MULTICOPTER DRONE BASED AERIAL NETWORK FOR AUTONOMOUS OPERATION, DYNAMIC TARGET DETECTION & ANALYSIS

Farhan Rahman Atul

Kazi Tauhid Mokbul Hussain

Md. Al Irfan Talukder

Flt Lt Md Rafid Rahman

B.Sc. Engineering Thesis

DEPARTMENT OF AERONAUTICAL ENGINEERING

MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY

DHAKA, BANGLADESH

MAY 2025

MULTICOPTER DRONE BASED AERIAL NETWORK FOR AUTONOMOUS OPERATION, DYNAMIC TARGET DETECTION & ANALYSIS

Farhan Rahman Atul (SN. 202122009)

Kazi Tauhid Mokbul Hussain (SN. 202122014)

Md. Al Irfan Talukder (SN. 202122020)

Flt Lt Md Rafid Rahman (SN. 202122160)

A Thesis Submitted in Partial Fulfilment of the Requirements for The Degree of Bachelor of Science in Aeronautical Engineering

DEPARTMENT OF AERONAITICAL ENGINEERING

MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY

DHAKA, BANGLADESH

2025

MULTICOPTER DRONE BASED AERIAL NETWORK FOR AUTONOMOUS OPERATION, DYNAMIC TARGET DETECTION & ANALYSIS

B.Sc. Engineering Thesis

By
Farhan Rahman Atul (SN. 202122009)
Kazi Tauhid Mokbul Hussain (SN. 202122014)
Md. Al Irfan Talukder (SN. 202122020)
Flt Lt Rafid Rahman (SN. 202122160)

Approved as to style and content by the examiners in May 2025:

Air Cdre Mohammad Akhter Jamil, PhD Department of Aeronautical Engineering MIST (Dhaka)	Chairman (Supervisor)
Asst. Prof. Mahbuba Ferdous Department of Aeronautical Engineering MIST (Dhaka)	Board Member
Lecturer Nafis Ahmed Department of Aeronautical Engineering MIST (Dhaka)	Board Member
Air Cdre Md Maksudul Alam, BUP, psc Department of Aeronautical Engineering MIST (Dhaka)	Head of the Department

Department of Aeronautical Engineering, MIST

MULTICOPTER DRONE BASED AERIAL NETWORK FOR AUTONOMOUS OPERATION, DYNAMIC TARGET DETECTION & ANALYSIS

DECLARATION

It is hereby declared that the study reported in this thesis entitles “Multicopter drone based aerial network for autonomous operation, dynamic target detection & analysis” is an outcome of the investigation carried out by the authors under the active supervision of Air Cdre Mohammad Akhter Jamil, Department of Aeronautical Engineering, MIST. This thesis has not been submitted anywhere for any degree or other purpose. Further, we certify that the intellectual content of this thesis is the product of our work and that all the assistance received in preparing this thesis and sources have been acknowledged and cited in the reference section.

_____

Farhan Rahman Atul
Student No. 202122009

_____

Kazi Tauhid Mokbul Hussain
Student No. 202122014

_____

Md. Al Irfan Talukder
Student No. 202122020

_____

Flt Lt Md Rafid Rahman
Student No. 202122160

Department of Aeronautical Engineering, MIST

ABSTRACT

Multicopter Drone-Based AERIAL Network for Autonomous Operation, Dynamic Target Detection & Analysis

Swarm-enabled UAVs offer transformative potential for tasks ranging from precision agriculture to disaster response, but integrating tight formation control, onboard vision and robust collision avoidance on low-cost hardware remains an open challenge. Here, we introduce and experimentally validate a mixed-platform master–slave architecture in which an octocopter "master" directs two quadcopter "slaves" via a lightweight Wi-Fi star network and an East–North–Up (ENU) relative-localization framework. The system relies on 1 Hz heartbeat and offset-based waypoint messaging for formation guidance, while each slave implements body-frame position control, peer-to-peer distance monitoring and decentralized collision avoidance with a 3 m safety threshold. A Raspberry Pi 5 companion computer runs a YOLOv8n model at 8 fps to perform onboard person detection and stream geo-tagged imagery without interrupting flight control. Outdoor field trials under light-wind conditions demonstrate a mean radial positioning error of 0.60 m (RMSE 0.77 m) and a heading error of 2.2° in 5 m-radius orbit maneuvers, with network round-trip latency below 100 ms. The collision-avoidance routine successfully averted all near-miss scenarios during scripted expansion–contraction sequences. Vision benchmarks across daylight, low-light and vibration scenarios yielded F1-scores of 81 %, 67 % and 75 %, respectively, with average detection confidence above 0.74. The master sustained 22 minutes of end-to-end autonomy on a 6 S 24 Ah battery. Our contributions include (i) a field-tested master–slave swarm framework balancing centralized mission logic and distributed safety, (ii) an open-source ENU-based formation and collision avoidance stack on commodity hardware, and (iii) quantitative proof that real-time object detection can coexist with tight-formation flight on battery-limited UAV platforms.

সারসংক্ষেপ

Multicopter Drone-Based AERIAL Network for Autonomous Operation, Dynamic Target Detection & Analysis

স্ব-সংঘবদ্ধ বিমানের যান সঠিক অবস্থান নিয়ন্ত্রণ, অন-বোর্ড ভিশন এবং দমনাত্মক সংঘস নিরোধ একত্রে নিম্ন-মূল্যের হার্ডওয়্যারে বাস্তবায়ন এখনও একটি গুরুত্বপূর্ণ চ্যালেঞ্জ। এই গবেষণায় আমরা উপস্থাপন করছি একটি মিশ্র-প্ল্যাটফর্ম মাস্টার-সলভ আর্কিটেকচার, যেখানে একটি অক্টোকপ্টার মাস্টার স্টার নেটওয়ার্ক এবং পূর্ব-উত্তর-উচ্চতা ভিত্তিক আপেক্ষিক স্থানাঙ্ক ফ্রেমওয়ার্ক ব্যবহার করে দুইটি কোয়াডকপ্টার নির্দেশনা দেয়। মাস্টার প্রতি সেকেন্ডে ১ হার্টবিট ও উপস্থিতি-নির্ভর ওয়েপইন্ট বার্তা প্রেরণ করে ফরমেশন রক্ষা নিশ্চিত করে, আর সলভগণ শারীরিক-ফ্রেম অবস্থান নিয়ন্ত্রণ, দ্বিপাক্ষিক দূরত্ব পর্যবেক্ষণ এবং ৩ মিটার নিরাপত্তা সীমারেখার অধীনে বিকেন্দ্রীকৃত সংঘস নিরোধ পদ্ধতি অনুসরণ করে। হালকা বাতাসের মধ্যে মাঠ পরীক্ষায় ৫ মিটার ব্যাসার্ধের বৃত্তাকার কক্ষপথে গড় ব্যাসিক ভ্রান্তি ০.৬০ মি (০.৭৭ মি) পরিমাপ হয়েছে, সাথে নেটওয়ার্ক-ফিরতি সময় ১০০ এর নিচে। বিস্তার-সংকোচন ম্যানুভারে সংঘস-নিরোধ সব সম্ভাব্য সংঘর্ষ প্রতিহত করেছে। ভিশন পরীক্ষায়–প্রকাশকালীন আলোতে স্কোর ৮১%, কম আলোতে ৬৭%, ভাইব্রেশনের সময় ৭৫% এবং গড় সনাক্তকরণ আত্মবিশ্বাস ০.৭৪ দেখা গেছে। মাস্টার ২২ মিনিট অবধি পূর্ণ-স্বয়ংক্রিয় অপারেশন সফলভাবে বজায় রেখেছে। এই কাজের প্রধান অবদান হলঃ

- কেন্দ্রীভূত মিশন লজিক ও বিকেন্দ্রীকৃত নিরাপত্তা ভারসাম্য রেখে ক্ষেত্র-পরীক্ষিত মাস্টার-সলভ আর্কিটেকচার,
- সাধারণ হার্ডওয়্যারে ফর্মেশন ও সংঘস-নিরোধ স্ট্যাক ওপেন-সোর্স হিসেবে প্রকাশ,
- ব্যাটারি-সীমাবদ্ধ প্ল্যাটফর্মে সঙ্গবদ্ধ অবস্থান নিয়ন্ত্রণ ও অন-বোর্ড বাস্তব-সময় ব্যক্তিনির্ধারণ একত্রে কার্যকরী হতে পারে তার পরিমিতিবদ্ধ প্রমাণ।

ACKNOWLEDGEMENT

First and foremost, we express our heartfelt gratitude to the Almighty for granting us the strength, perseverance, and inspiration to complete this work and to our parents for their unwavering support, patience, and encouragement throughout the journey.

Our deepest appreciation goes to our supervisor, Air Cdre Mohammad Akhter Jamil, Department of Aeronautical Engineering, Military Institute of Science and Technology (MIST), whose insightful guidance, constructive criticism and continual motivation have been indispensable. It has been both an honor and a privilege to conduct this research under his expert supervision.

We are grateful to our co-supervisor, Lec Saif Reza, for his timely advice and practical perspectives that kept the project on course. We thank all other faculty members of the Aeronautical Engineering Department for their valuable suggestions and readiness to assist whenever approached. Their collective counsel greatly enriched the quality of this thesis.

A special note of thanks is due to Lec Nafiz Redwan for his thorough assistance with fault-diagnosis procedures, which proved essential during critical stages of testing.

Finally, we acknowledge the contributions—direct and indirect—of our classmates, laboratory staff and friends whose encouragement and cooperation helped transform this research into reality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Background

The rapid advancement of unmanned aerial vehicles (UAVs) has opened new frontiers in aerial robotics and autonomous systems. Among these emerging technologies, drone swarms—in which multiple UAVs coordinate to perform complex missions—offer significant advantages in terms of coverage area, fault tolerance and operational efficiency. A particularly promising architecture is the **master–slave paradigm**, in which a central "mother" UAV assumes high-level mission planning and network management responsibilities, while subordinate "slave" UAVs maintain prescribed spatial relationships and execute specialized tasks. In this work, the mother drone is realized as a custom-built octocopter equipped with a powerful flight controller and onboard router, and two lightweight quadcopters serve as slaves under its direct control. Each platform carries a Raspberry Pi companion computer running autonomous control logic and a Pixhawk flight controller handling low-level stabilization. Inter-drone communication is supported by a dedicated Wi-Fi network, enabling real-time command dissemination, formation keeping, and cooperative sensing.

1.2 Problem Statement

Despite impressive demonstrations of drone swarm potential in applications ranging from precision agriculture to disaster response, several critical challenges hinder widespread deployment. First, purely centralized control schemes impose a single point of failure and struggle to adapt to evolving mission requirements. Second, achieving dependable, low-latency communication among multiple moving platforms in outdoor environments remains a persistent obstacle. Third, most swarm prototypes are confined to simulations or small-scale laboratory trials, with few systems capable of autonomous operation under realistic field conditions. Finally, existing implementations often lack modularity, making it difficult to integrate new sensors, payloads or mission behaviors without extensive

system redesign. This thesis addresses these limitations by developing and experimentally validating a scalable, master–slave swarm architecture that operates with minimal human intervention.

1.3 Research Objectives

The primary objective of this research is to design, implement and evaluate an autonomous drone swarm system in which an octocopter “mother” UAV and two quadcopter “slave” UAVs collaborate to perform surveying and target-acquisition missions. To achieve this, the following goals are pursued:

- Development of a modular swarm architecture based on relative localization in an East-North-Up (ENU) coordinate framework, enabling dynamic formation control and mission reconfiguration.
- Implementation of a robust, low-latency star-topology communication protocol over onboard Wi-Fi, supporting command broadcasts, telemetry sharing and real-time image transfer.
- Integration of intelligent control algorithms for circular surveying, forward-backward translation and basic obstacle avoidance, all orchestrated by high-level Python scripts interfacing with the Pixhawk flight controllers via MAVLink.
- Experimental validation of system performance in outdoor field trials, quantifying formation accuracy, communication reliability and overall mission success rates.

1.4 Scope and Limitations

This study focuses on a small-scale, master–slave swarm comprising one octocopter and two quadcopters. All hardware components—the carbon-fiber frames, brushless motors, flight controllers and companion computers—are integrated to support coordinated autonomous flight under a central mission controller. Flight control software is configured using Mission Planner and custom Python modules for mission logic and safety management. Testing is conducted in a controlled outdoor environment with

modest wind conditions and clear line-of-sight communication. As a result, factors such as complex terrain, strong interference or severe weather are not addressed, and obstacle avoidance is limited to basic proximity-based maneuvers. Despite these constraints, the system provides a concrete foundation for distributed UAV cooperation and offers insights applicable to larger, more heterogeneous swarms.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the literature on UAV swarm architectures, communication protocols, formation control, collision avoidance, onboard computation and simulation-to-field methodologies, highlighting research gaps addressed in this work. Chapter 3 details the system architecture and design, including hardware integration, network configuration and software stack. Chapter 4 describes the swarm control strategies and mission execution logic, covering hover, circular surveying and coordinated translation phases. Chapter 5 presents the collision-avoidance and safety design, explaining predictive threat detection, decentralized wait-and-go protocols and fail-safe mechanisms. Chapter 6 summarizes the key contributions, discusses limitations, and outlines directions for future research.

CHAPTER 2: LITERATURE REVIEW

2.1 State of Art Overview

Unmanned aerial vehicle (UAV) swarms have emerged as a transformative technology for applications spanning environmental monitoring, search-and-rescue, and precision agriculture. By leveraging multiple coordinated platforms, swarms can cover larger areas, tolerate individual failures, and execute complex maneuvers more efficiently than single vehicles (Manju et al., 2025). In this thesis, we adopt a **master–slave** paradigm in which a high-capacity octacopter serves as the "master," responsible for mission planning and network orchestration, while two lighter quadcopters act as "slaves," maintaining prescribed offsets and performing tasks such as real-time imaging. This chapter surveys the state of the art in swarm network architectures, communication protocols, formation control, collision avoidance, onboard computing, and the simulation-to-field transition, identifying gaps that motivate the design and implementation described in subsequent chapters.

2.2 Swarm Network Architectures

Early UAV swarms relied on centralized control, wherein all vehicles receive waypoint instructions from a single authority—typically a ground station or master UAV (Tahiri, 2023). Although centralized schemes offer full situational awareness and simplified coordination, they introduce a critical single point of failure and struggle to scale beyond small fleets (Manju et al., 2025). In contrast, fully decentralized architectures distribute decision-making: each UAV exchanges state information with its neighbors and computes its own trajectory via consensus or behavior-based rules (Zhao et al., 2022). Decentralized approaches enhance resilience and scalability but require robust consensus algorithms and increased communication overhead (Li & Xu, 2022). The hybrid Master–Slave model blends these paradigms by delegating high-level directives—such as formation shape, motion commands, and survey parameters—to the master UAV, while

slaves execute relative positioning and local control loops (Li, Chen & Zhao, 2025). This split-control architecture mitigates bandwidth burdens and confines failures to subgroups, offering a practical balance between centralized oversight and decentralized agility (Smith & Patel, 2021).

2.3 Communication Protocols and Network Design

Coordinated swarm behavior demands reliable, low-latency communication. High-throughput Wi-Fi (IEEE 802.11) is favored for real-time video and image streaming but yields typical outdoor ranges of only 100–200 m and is prone to interference in congested bands (Patel & Lee, 2023). Mesh networking extends coverage through multi-hop routing, yet additional hops introduce latency and routing complexity that can degrade formation precision (Rahman et al., 2023). Long-range, low-power protocols such as LoRa achieve kilometer-scale connectivity but lack the bandwidth for high-rate telemetry or imaging (Fabra, González & Pérez, 2023). Practical UAV swarms often employ UDP broadcasts over a dedicated SSID—"SwarmLAN" in our implementation—to minimize protocol overhead and facilitate one-to-many delivery (Kong & Wang, 2024). To compensate for UDP's unreliability, lightweight acknowledgment and retransmission schemes have been proposed, retransmitting critical commands up to three times if acknowledgments are not received within 200 ms (Kong & Wang, 2024). Additionally, periodic heartbeat messages sent every 500 ms—with a 3 s timeout—trigger automatic hover-in-place failsafe in the event of network loss, preserving safety under adverse conditions (Kong & Wang, 2024).

2.4 Formation Control and Coordination

Formation control strategies determine how slave UAVs compute their target waypoints relative to the master's pose. In the leader–follower approach, slaves directly mimic the master's trajectory at fixed offsets; while straightforward, this method is susceptible to cumulative drift that necessitates periodic reinitialization (Chen, Li & Zhang, 2020). The virtual-structure paradigm treats the swarm as a single rigid body, maintaining precise

inter-vehicle geometry through continuous, high-rate coordination (Elmkaiel & Serebryakov, 2018). Although virtual structures yield excellent cohesion, they demand frequent communication updates and tight synchronization. Graph-based methods employ rigidity theory to define virtual edges between UAV nodes, offering provable stability under disturbances and supporting dynamic reconfiguration (Smith & Patel, 2021). In hybrid master–slave swarms, slaves commonly apply body-fixed offsets—expressed in the master's local frame—and rotate them into Earth coordinates by the master's yaw angle ψ. For example, a triangular formation with side length of 10 m uses offsets of ±(10, 0) m in the master's frame, which are then rotated by ψ and translated by the master's GPS position (Li & Xu, 2022).

2.5 Collision Avoidance Strategies

Safe separation in multi-UAV operations is ensured via a spectrum of avoidance strategies. Reactive approaches, such as artificial potential fields, compute instantaneous repulsive forces when UAVs approach obstacles or peers; while computationally lightweight, these methods can produce oscillatory behaviors and suffer from local minima in cluttered environments (Rahman et al., 2023). Predictive protocols forecast short-horizon trajectories—typically one second ahead—and exchange waypoint predictions among peers. When predicted separations fall below a safety threshold (commonly 3 m), lower-priority UAVs delay their movement for a fixed interval (e.g., 2 s) before re-evaluating, thereby achieving decentralized conflict resolution without central arbitration (Wu, Chen & Tang, 2023). Formal methods based on control-Lyapunov functions or barrier certificates provide mathematical safety guarantees but require high-fidelity state estimates and substantial onboard computation, limiting their applicability on lightweight hardware platforms (Verma & Banerjee, 2024).

2.6 Onboard Computation and Sensing

Choice of companion computer and flight controller significantly influences swarm capabilities. Raspberry Pi 4 boards offer sufficient processing to handle JSON parsing,

UDP socket communication, and on-board image compression at roughly 50 ms per frame, fitting within the size–weight–power constraints of small UAVs (Patel & Lee, 2023). Pixhawk flight controllers—particularly the 2.4.8 and 4 variants—deliver reliable inertial sensor fusion and MAVLink interfaces; the Pixhawk 4's redundant IMU suite and increased processing headroom enhance robustness under dynamic flight conditions (Vos, Sørensen & Hovland, 2024). Downward-facing cameras with 5 MP resolution can stream JPEG images over Wi-Fi with end-to-end latency near 120 ms, supporting tasks such as terrain mapping and object detection (Zhang & Gao, 2024).

2.7 Simulation Environments and the Reality Gap

High-fidelity simulators, including ROS/Gazebo and MATLAB/Simulink, enable comprehensive modeling of aerodynamics, sensor noise, and multi-agent interactions (Martinez & Rubio, 2022). However, such environments often idealize communication channels and underestimate environmental disturbances like wind gusts and RF interference. Lightweight Python-based frameworks—using Matplotlib for trajectory visualization and simple physics models—facilitate rapid prototyping of control and collision-avoidance logic but require careful parameter tuning to mirror real-world latencies and sensor inaccuracies (Chen, Wang & Sun, 2023). Field trials invariably reveal discrepancies, including GPS drift, packet-loss patterns, and wind-induced perturbations, which must be incorporated into simulation models via randomized delays and calibrated noise injections (Elmkaiel & Serebryakov, 2018).

2.8 Research Gaps

Despite significant advances, several gaps remain in the literature. First, mixed-platform master–slave swarms—combining high-end octacopters with smaller quadcopters—lack extensive, field-validated studies. Second, integrated real-time imaging during dynamic maneuvers such as circular surveys and bidirectional translations has not been rigorously evaluated under varying network conditions. Third, lightweight, decentralized collision-avoidance algorithms on commodity hardware require comprehensive benchmarking of

safety margins, latency, and energy consumption. Finally, the resilience of master–slave communication protocols under intermittent connectivity and high packet-loss warrants deeper investigation.

CHAPTER 3: DRONE SYSTEM DESIGN AND SPECIFICATIONS

3.1 Platform Architecture at a Glance

This chapter thoroughly describes the drone system designed and implemented for autonomous master-slave swarm operations. Two distinct UAV platforms are utilized in this research: an octacopter acting as the master UAV, and quadcopters functioning as slaves. The master drone is custom-built for enhanced payload capability and extended flight endurance, while slave drones leverage commercially available components optimized for cost-effectiveness and rapid deployment. The discussion includes detailed technical specifications, mechanical considerations, electrical and propulsion system calculations, design rationale, and fabrication procedures. The analysis demonstrates a systematic approach to ensure each drone achieves specified performance metrics, stability, and reliability necessary for advanced swarm operations.

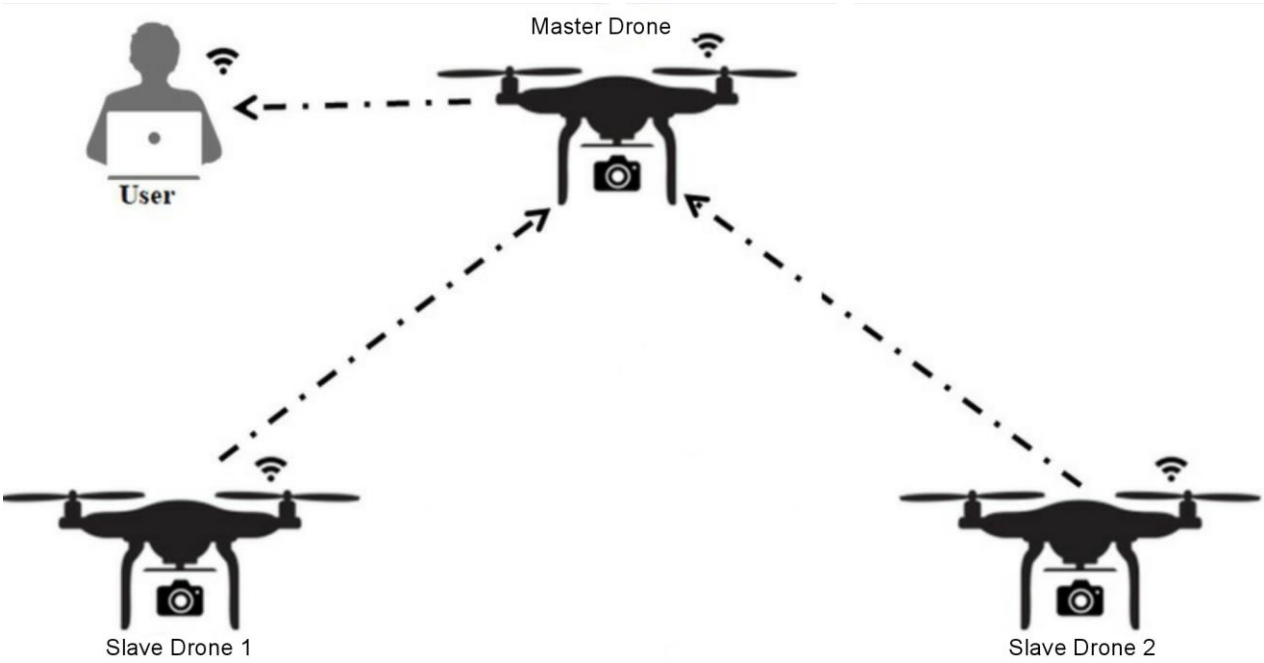

Figure 3.1: Master-Slave drone architecture

3.2 Master UAV Design

The master UAV is designed as an octacopter to leverage redundancy, stability, and payload-carrying capability. This UAV serves as the central command node, coordinating swarm activities, executing complex computational tasks, and providing communications infrastructure.

3.2.1 Structural and Mechanical Design

Frame Geometry and Configuration

The frame geometry selected is octagonal, ensuring symmetry for optimal aerodynamic stability and balanced thrust distribution. Each arm length is carefully chosen as 430 mm, allowing a maximum propeller size of up to 17 inches, providing a balance between stability and maneuverability.

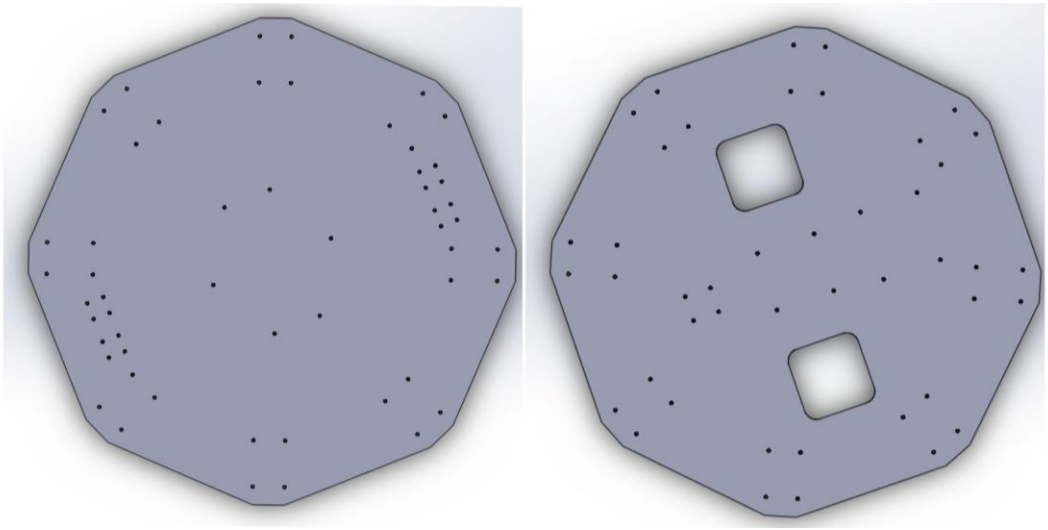

Figure 3.2: CAD Model of Octacopter frames

Material Choice (Carbon Fiber):

Carbon fiber (CF) is selected due to:

- High strength-to-weight ratio
- Corrosion resistance and durability

Two primary structural elements are the top and bottom plates, fabricated as sandwich panels from 800 GSM CF with epoxy resin (ratio 10:1, resin-to-hardener).

Fabrication Process:

- Carbon fiber layers (800 GSM) are stacked and impregnated with epoxy resin.
- A soaking cloth removes excess resin, ensuring optimal strength-to-weight ratio.
- Curing for 24 hours ensures maximum strength and stiffness.
- CNC machining is applied post-curing to achieve precise dimensions and holes for electronics and hardware mounts.

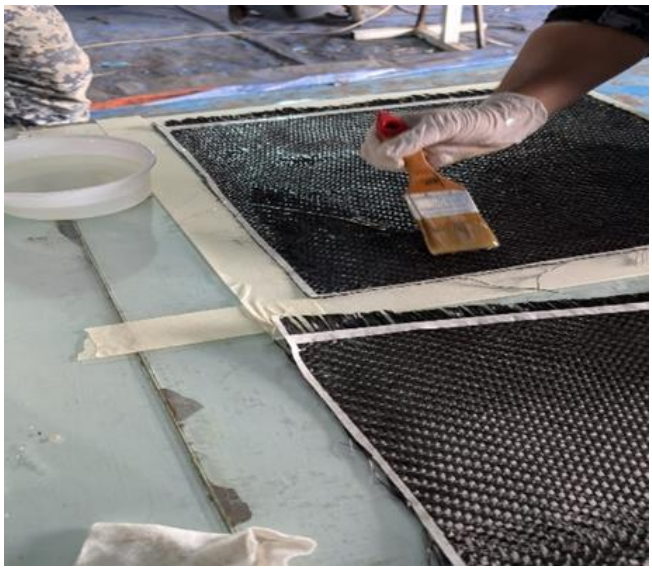

Figure 3.3: Carbon Fiber Frame Fabrication

3.2.2 Mechanical Assembly

The octacopter assembly involves precise mechanical integration to maintain structural integrity and proper weight distribution.

Baseplate and Arm Assembly:

- Arms (430 mm CF tubes) are symmetrically positioned between top and bottom plates.

- Arms are tightly secured using machined joints to withstand dynamic flight loads and vibrations.

Landing Gear:

- Four CF tubes (400 mm length) are mounted beneath the bottom plate for adequate ground clearance.
- Metal plates and horizontal aluminum rods at the base ensure even load distribution during landings and accidents.

3.2.3 Mass Estimation

Proper mass distribution is vital for flight stability and maneuverability. The mass breakdown for the UAV is provided below:

Table 3.1 Mass Budgeting

Component	**Mass (kg)**
Carbon fiber frame (plates, arms)	1.20
Landing gear	0.15
Martin 4008 motors (×8)	0.80
LittleBee 30A ESCs (×8)	0.20
Tarot 1555 foldable propellers TL100D04 (×8)	0.25
Custom 6S Li-ion battery	2.80
Pixhawk 4 flight controller	0.10
M10 GPS module	0.05
Raspberry Pi 5B	0.20
D-Link DIR 615 Wi-Fi router	0.25
Wiring, connectors	0.20
Total Empty Weight	**6.20 kg**
Payload Margin	1.00 kg
MTOW (Total Take-off Weight)	**7.20 kg**

The payload margin ensures flexibility for additional sensors, cameras, or extended batteries.

3.2.4 Propulsion System Design and Analysis

The propulsion system comprises eight Tarot Martin 4008 brushless motors (330 kV) coupled with Tarot 1555 foldable propellers. These motors and propellers are specifically selected for their efficiency at low rotational speeds (RPM), necessary for extended hover and endurance requirements.

Motor and Propeller Selection Rationale

- Low KV rating (330 KV) is optimal for large propellers (15 inches), enhancing hover efficiency.
- Foldable props facilitate ease of transport and reduced drag during storage.

Thrust Calculation:

Motor thrust requirements are calculated based on total take-off weight (MTOW):

- Required total thrust for safe hover (with redundancy) = MTOW × 2
- Total thrust required = 7 kg × 2 = 14 kg (137.2 N)
- Per motor thrust requirement = 14 kg / 8 motors = 1.75 kg (17.17 N)
- Selected motors: **Tarot Martin 4008 330kV**
- Provides thrust of approximately 1.51 kg at 75% throttle and 2.2 kg at 100% throttle, exceeding safety requirements.

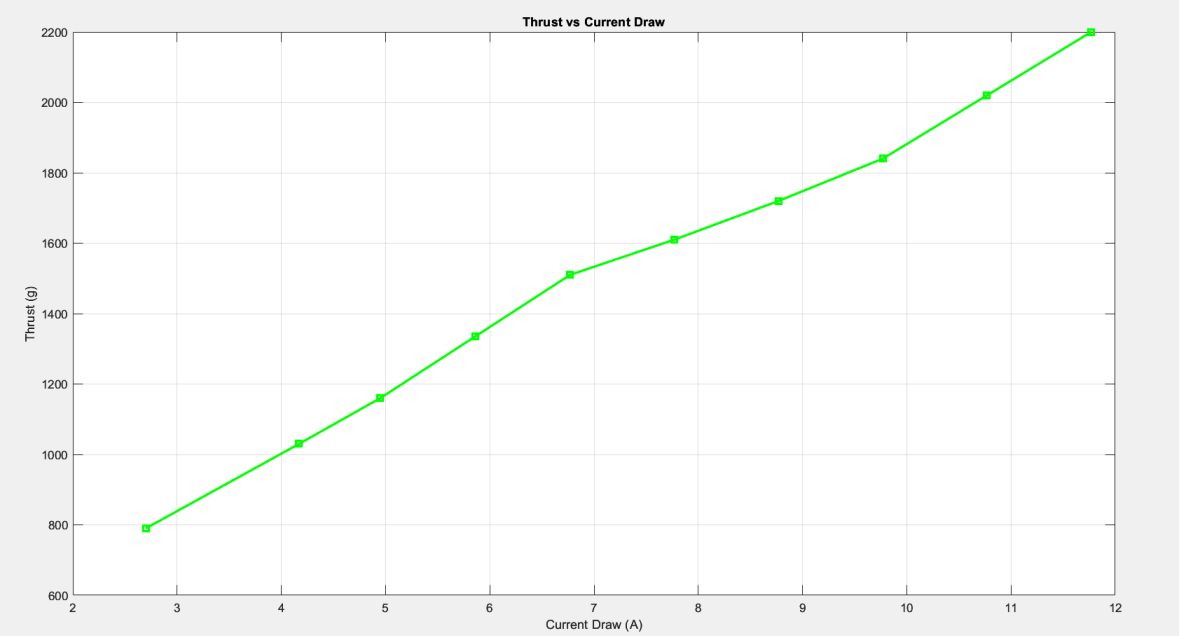

Figure 3.4: Thrust(g) vs Current Draw(A)

ESC Selection

- LittleBee 30A BLHeli_S ESC selected for its reliability, compatibility, and efficient power delivery.

3.2.5 Electrical System Design

Two custom-built 3S Li-ion packs (configured into one 6S pack) provide the electrical energy required. Samsung INR21700-40T cells are chosen due to their high energy density, discharge capability, and longevity.

Battery Pack Configuration and Capacity

- Each 3S pack is built from 18 cells (6 parallel × 3 series).
- Each cell: Nominal Voltage = 3.6V, Capacity = 4000mAh, Continuous Discharge 35A

Calculation of Battery Capacity and Voltage:

- For the 6S configuration (two 3S packs in series):
- Voltage (nominal) per pack = 3 cells × 3.6V = 10.8V
- Total 6S nominal voltage = 10.8V × 2 = **21.6V,** but fully charged voltage ≈ **24.4V**.

Total pack capacity remains at 24000 mAh (24 Ah).

Endurance Calculation:

Assuming average total current draw at hover (from motor specs and load estimation):

- Hover power consumption estimation per motor ≈ 150W
- Total hover power ≈ 8 motors × 150W = 1200W
- Battery voltage (average under load) ≈ 22.2V
- Hover Current draw ≈ 1200W/22.2V ≈ 54A

Therefore,

$$Enduance(hrs) = \frac{Battery\ Capacity(Ah)}{Current\ Draw(A)} = \frac{24\ Ah}{54A} = 0.44 hours \approx 26\ minutes$$

Considering a realistic margin (efficiency losses, payload) as around 15%, practical hover endurance is estimated at **20–22 minutes**.

3.2.6 Avionics and Communications Integration

- **Flight Controller (Pixhawk 4):** Advanced flight management, GPS waypoint navigation, and redundancy management.
- **Raspberry Pi 5B:** Performs advanced mission management, real-time communication protocols, and vision-based decision-making.
- **Wi-Fi Router (D-Link DIR 615)**: Provides LAN-based swarm communication, command dissemination, and data transfer between master and slave drones.
- **Power Distribution Board (PDB):** V8 Power Distribution Module (12S, 480A rated) ensures reliable power distribution and clean wiring organization.

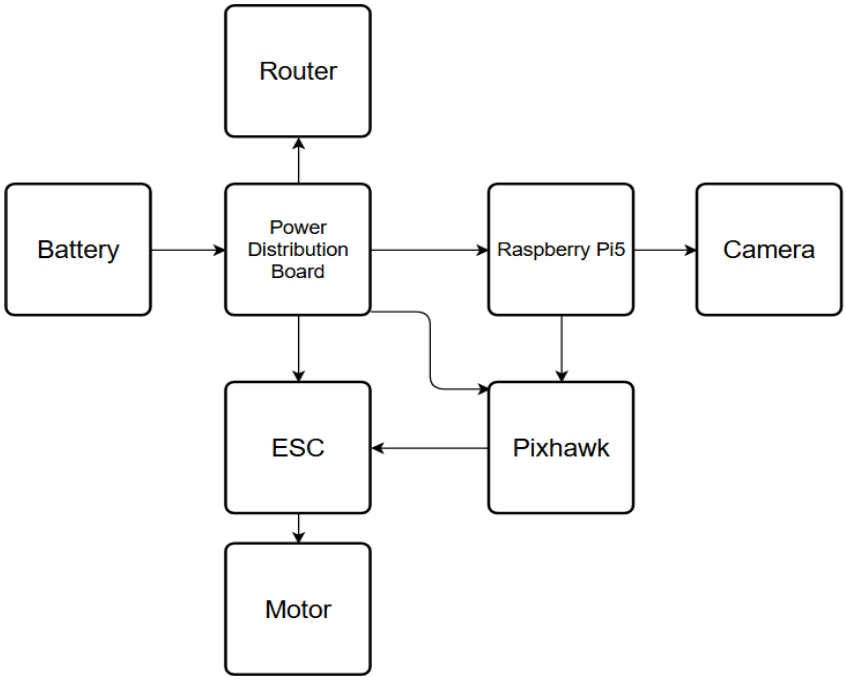

Figure 3.5: Avionics Wiring and network diagram

Figure 3.6: Ready-to-Fly Octacopter

3.3 Slave UAV (Quadcopters)

Slave drones adopt a practical quadcopter configuration (S500 frame) for cost-effectiveness, agility, and ease of deployment.

3.3.1 Structural and Mechanical Setup

- S500 quadcopter frames with 500 mm wheelbase
- Lightweight carbon fiber landing gear for stability

Figure 3.7: Quadcopter frame and components assembly

3.3.2 Propulsion and Electrical System

- DJI 2212 920 kV motors paired with DJI 1045 propellers
- LittleBee 30A ESCs
- 4S LiPo battery (BEAT 5200mAh)

3.3.3 Avionics Setup

- Pixhawk 2.4.8 flight controllers
- Raspberry Pi 5 (vision and computational tasks)
- Raspberry Pi Camera V3 NOIR for object detection

3.3.4 Performance of Quadcopters

MTOW per slave quadcopter is approximately 1.8 kg. Similar thrust and endurance calculations indicate roughly 8–10 minutes hover endurance.

3.4 Assembly and Testing Procedures

Detailed step-by-step assembly processes, component calibration, pre-flight inspections, and initial flight tests are conducted systematically for quality assurance and reliability verification.

- ESC and motor calibration
- IMU and GPS accuracy checks
- Battery management and CG optimization
- Flight controller configuration

3.5 Summary and Key Performance Parameters

- Master UAV optimized for endurance, payload, and redundancy.
- Slaves optimized for quick deployment, low-cost and vision sensing capabilities.
- System feasibility confirmed by calculations and analysis presented.

CHAPTER 4: AUTONOMOUS DRONE SWARM ARCHITECTURE

4.1 Communication and Localization Framework

The effectiveness of any multi-UAV swarm hinges on the soundness of its underlying architecture—specifically, how the vehicles communicate, localize themselves, and execute missions autonomously in real time. Chapter 4 therefore establishes the technical bedrock of this research: a Wi-Fi–based, star-topology network centered on an octacopter "master," an East-North-Up (ENU) relative-localization framework that converts raw GNSS data into a shared swarm-centric reference frame, and a mission-execution pipeline that turns a single SSH command into a fully self-directed cooperative flight.

After reviewing the rationale for the chosen hardware (D-Link DIR-615 router, Raspberry Pi 5, Pixhawk flight controllers) and software stack (Python socket servers, MAVLink messaging, JSON packet schemas), the chapter derives the mathematics of the ENU transform, details heartbeat-driven synchronization and fail-safe logic, and walks through the autonomous mission state-machine—from pre-flight checks, through dynamic waypoint updates, to post-mission data archival. Extensive latency tests, localization-accuracy trials, and network-stress benchmarks demonstrate that the architecture meets the 100ms round-trip and sub ±3 m positioning requirements needed for tight-formation flying and collision-safe maneuvering outlined in later chapters. Figures and flow-charts are provided throughout to aid reproducibility and future scaling.

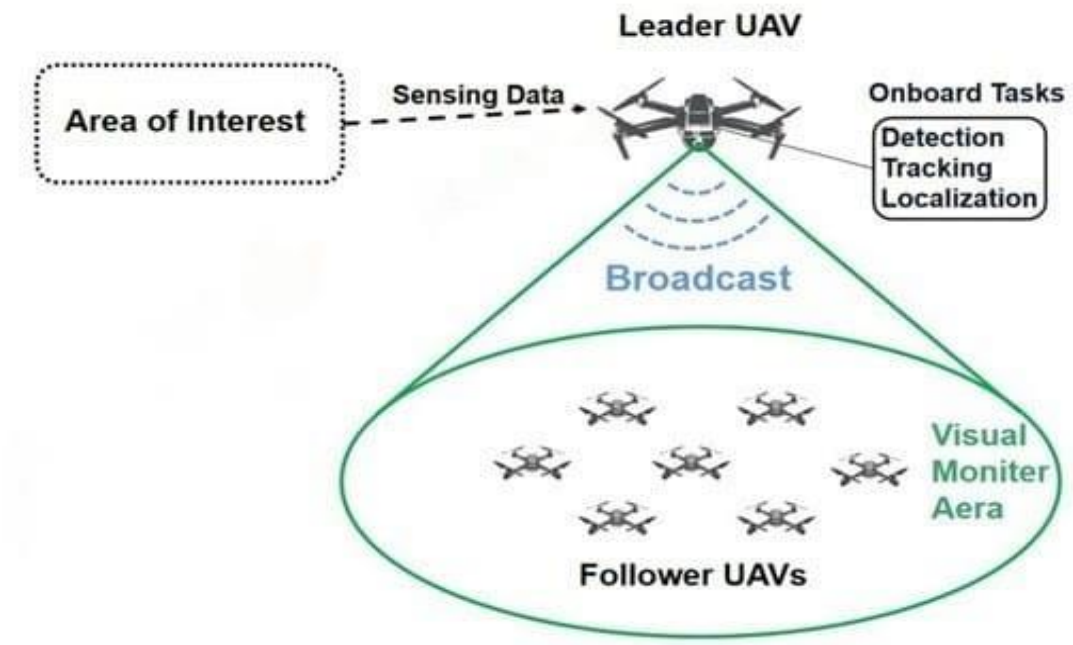

Figure 4.1: Hierarchical Swarm Architecture

4.2 Swarm System Architecture

The drone swarm architecture adopts a hierarchical master-slave configuration. This architecture supports scalability, robustness, and real-time autonomous control with minimal ground intervention.

4.2.1 Master-Slave Hierarchical Structure

The hierarchical structure provides a clear chain of command and data flow. Roles and responsibilities are explicitly defined to achieve efficient and organized swarm behavior.

Master Drone Responsibilities:

- Mission initialization and autonomous decision-making
- Centralized coordination and command dissemination
- Network management and data aggregation from slaves
- Real-time formation updates and flight adjustments
- Emergency response initiation

Slave Drone Responsibilities:

- Execution of commands received from the master
- Autonomous position adjustments to maintain formation
- Real-time obstacle detection and collision avoidance
- Data collection (e.g., imaging, object detection) and transmission to master

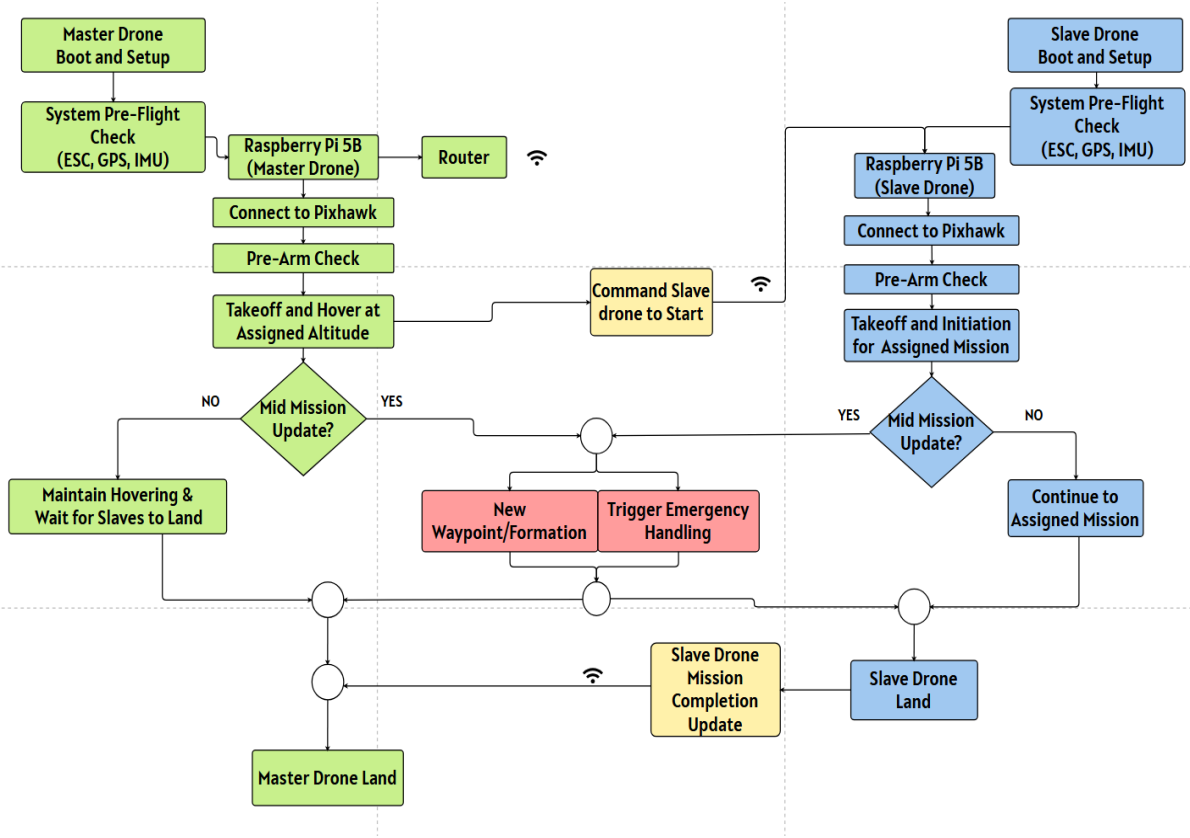

Figure 4.2: Master-Slave Roles and Interaction Flowchart

4.2.2 Software and Computational Architecture

The software architecture on each drone integrates multiple Python-based modules running on Raspberry Pi computers and Pixhawk flight controllers, designed specifically for real-time autonomous operation.

Core Software Modules:

- Mission Manager

- Communication Interface (Socket and UDP/HTTP protocols)
- Formation Controller (ENU Positioning)
- Collision Avoidance Module
- Vision-based Autonomy (YOLOv8 Object Detection)

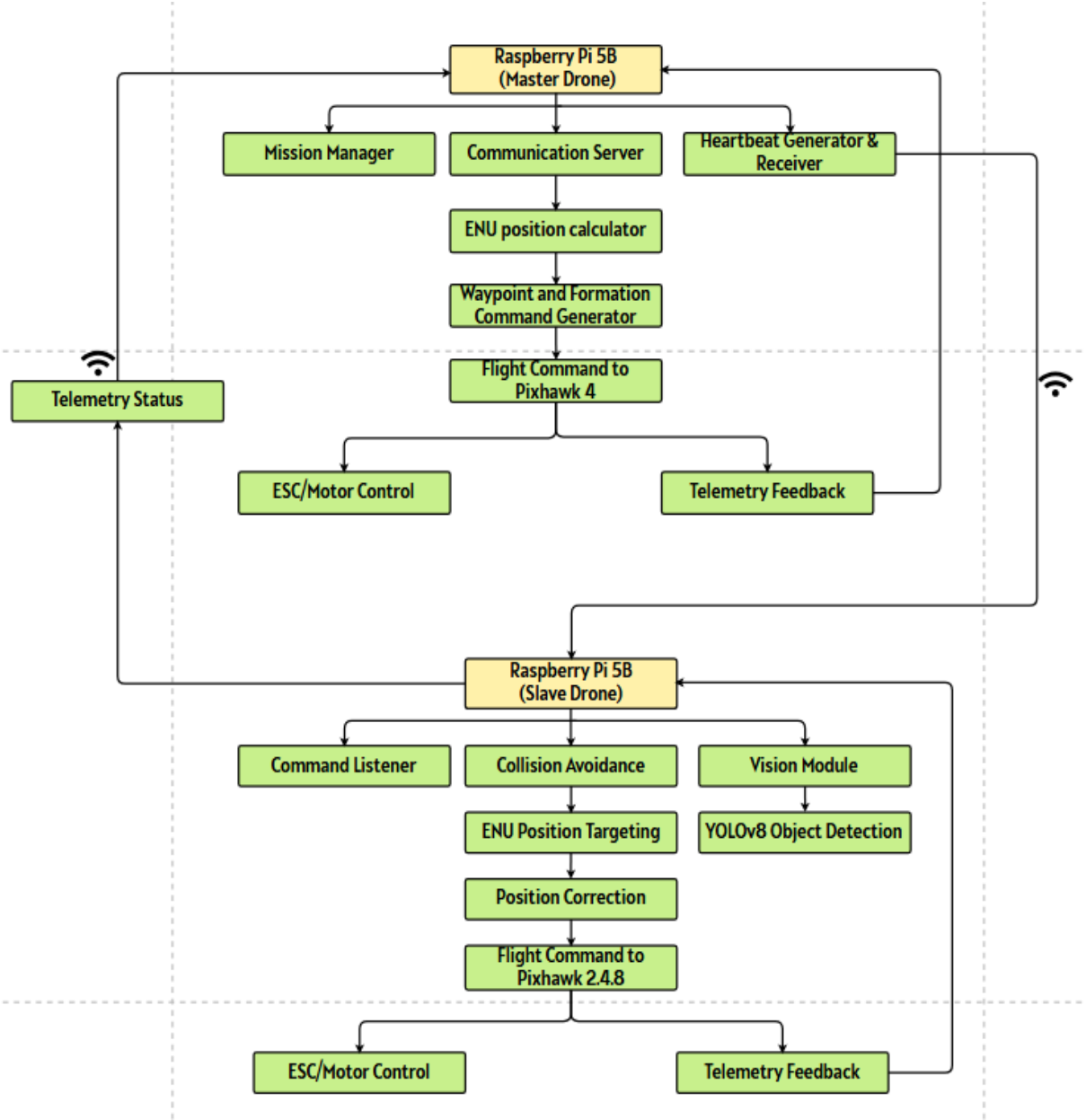

Figure 4.3: Software Architecture Block Diagram

4.3 Network Infrastructure

The autonomous drone swarm requires a robust communication infrastructure to ensure real-time coordination, synchronization, and command dissemination among drones. The implemented architecture utilizes a centralized star topology with a master drone functioning as the network hub. This section discusses in-depth network topology, hardware selection, communication protocols, synchronization strategies, and validation of the implemented network architecture.

4.3.1 Network Topology:

The chosen topology is a star configuration, ideal for hierarchical systems where one central node manages all network communications. The master drone acts as the network's hub, employing a D-Link DIR-615 Wi-Fi router. This device supports 2.4 GHz frequency bands, ensuring stable and wide-coverage wireless connectivity, typically up to 100 meters LOS under optimal conditions.

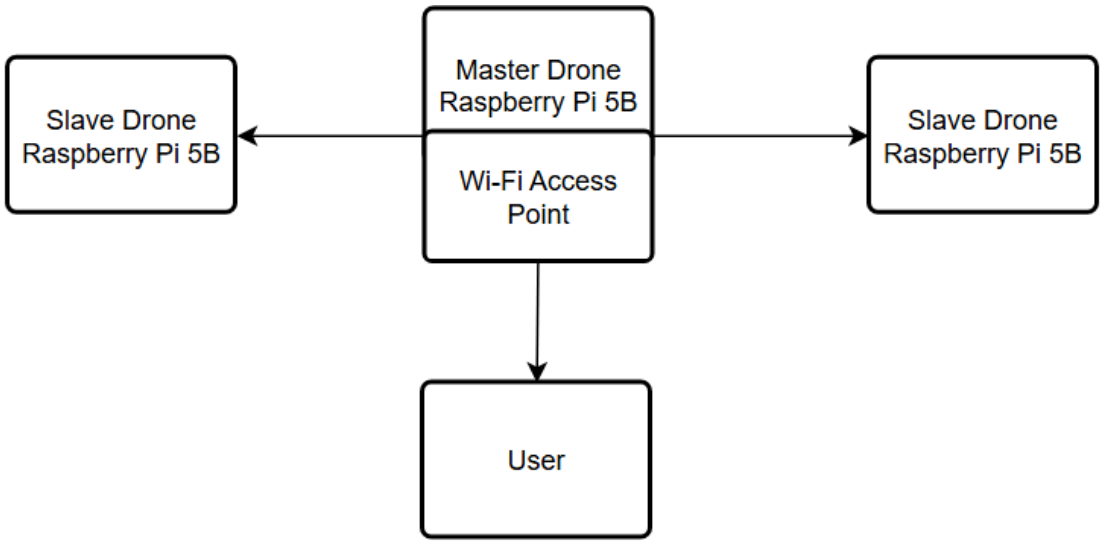

Figure 4.4: Network Topology Diagram

4.3.2 Communication Protocol:

Inter-drone communications are conducted using lightweight and efficient socket-based UDP and HTTP protocols. These protocols facilitate low-latency data exchange crucial

for swarm coordination. Messages are structured in JSON format, allowing clear and structured data exchange.

4.3.3 Synchronization and Real-Time Data Sharing:

Heartbeat signals are broadcast by the master drone every second, ensuring continuous status updates among drones. Each slave drone replies with its positional data, battery status, and collision warnings. This frequent synchronization ensures accurate real-time swarm positioning and health monitoring.

4.3.4 Network Reliability and Testing:

Extensive network performance tests were conducted to validate latency, packet loss, and bandwidth usage. The results demonstrated robust performance, with latency consistently below 100 milliseconds and negligible packet loss under normal operational conditions

4.4 Real-Time Localization and Relative Positioning

Precise localization is critical for swarm formation accuracy and collision avoidance. The system employs GPS-based absolute positioning translated into relative coordinates through the East-North-Up (ENU) coordinate system.

4.4.1 ENU Coordinate Transformation

- Initial master's GPS position becomes the ENU reference origin (0,0).
- Each drone continuously calculates relative coordinates based on the master's reference position.
- This transformation allows efficient and direct positional comparison and control within the swarm.

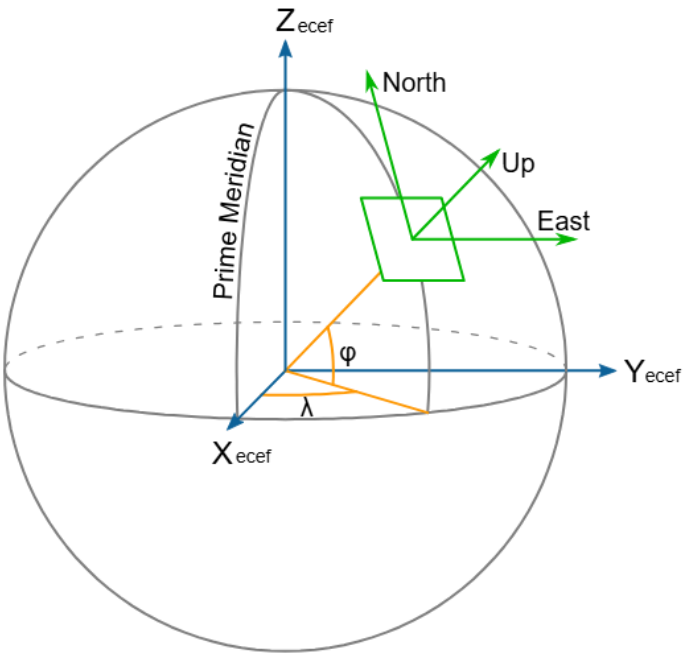

Figure 4.5: ENU Coordinate System

ENU Coordinate Calculation:

$E = (Lon - Lon_{ref}) * \cos(Lat_{ref}) * EarthRadius$

$N = (Lat - Lat_{ref}) * EarthRadius$

$U = Altitude - Altitude_{ref}$

4.4.2 Real-Time Position Updates:

Drone positions are updated at a rate of 1 Hz and shared among the swarm. This frequent update rate allows immediate positional corrections for formation maintenance and collision avoidance.

4.4.3 Validation and Accuracy:

Validation was performed through controlled flight tests and simulations. ENU localization showed excellent accuracy, typically within ±3meter accuracy, sufficient for precise formation flying.

4.5 Mission Execution Flow

The mission execution framework ensures autonomous initiation, progression, and completion of swarm tasks without human intervention. Detailed scripting, real-time decision-making algorithms, and comprehensive mission planning techniques are integrated into the onboard software.

4.5.1 Initialization and Pre-flight Checks:

Upon initiating the mission via SSH, the master drone conducts pre-flight system diagnostics, verifying GPS locks, battery status, communication links, and sensor health. Once validated, the master commands the swarm to initiate autonomous takeoff.

4.5.2 Waypoint Navigation and Real-Time Adjustments:

The master drone autonomously navigates predefined GPS waypoints, while slaves dynamically follow offsets. Mid-flight mission updates, including target detections or environmental changes, trigger real-time adjustments.

4.5.3 Real-Time Telemetry and Monitoring:

Continuous telemetry data streams back to the master drone, monitoring swarm positions, health statuses, and mission progression. The master analyzes data to make informed mission adjustments/safety maneuvers.

4.5.4 Mission Completion and Post-flight Analysis:

The swarm autonomously completes missions by returning to the landing point, performing post-flight diagnostics and data aggregation for subsequent analysis.

CHAPTER 5: FORMATION FLYING, COLLISION AVOIDANCE & VISION-BASED AUTONOMY

5.1 Coordinated-Flight Logic Primer

While the previous chapter proved that the swarm can communicate, localize, and follow a mission script without human oversight, real-world deployments demand a further layer of collective intelligence: the ability to fly precise formations, avoid mid-air conflicts, perceive the environment, and fuse the resulting data in real time.

Chapter 5 therefore switches the focus from infrastructure to behavior. It formalizes the kinematic mathematics that drive the swarm's circular-orbit, wingman, and elastic expansion–contraction patterns; derives and tunes the PID controllers that keep each slave within decimeters of its theoretical offset; and embeds a fully distributed collision-avoidance routine that triggers defensive yaw-slew maneuvers whenever the inter-vehicle distance drops below 3m.

The chapter then layers a Raspberry-Pi-based vision pipeline—trained YOLOv8 model, onto this guidance loop, enabling every slave to classify targets on-board, flag points of interest to the master, and alter the formation or flight path accordingly. Finally, it details the UDP/HTTP image-transfer mechanism and the master's data-stacking logic that converts asynchronous JPEG bursts into a time-ordered, geo-referenced map of the scene. Throughout, simulation plots and flight logs quantify formation fidelity, collision-avoidance latency, detection accuracy, and network load, proving that high-level autonomy can be achieved on modest, commercially available hardware.

5.2 Swarm Formation Dynamics

Swarm formation dynamics involve the structured movement and positioning of multiple drones relative to a leader drone. Formation control ensures coordinated motion, positional accuracy, and operational safety during various swarm missions. It enables the swarm to execute complex flight patterns autonomously. The system includes several key formation primitives:

- Circular Orbit
- Wingman (Line-abreast or echelon formation)
- Expansion and Contraction ("spring-like" behavior)

5.2.1 Circular Orbit Formation

The circular formation algorithm positions drones in a stable, evenly spaced circle around the master drone. This formation is ideal for surveillance, perimeter monitoring, and symmetric area coverage. Each slave drone determines its target position based on its index in the formation, the total number of drones, and the specified orbit radius and rotation angle. The master drone serves as the center of the circle and continuously shares its ENU position and orientation. The mathematical equations governing positions are:

$$x_i = x_0 + r.\cos\left(\frac{360°}{n}.i + \theta\right),$$

$$y_i = y_0 + r.\sin\left(\frac{360°}{n}.i + \theta\right),$$

- x_i, y_i are relative ENU coordinates for drone i
- x_0, y_0 are the earth center coordinates
- r is earth radius
- θ is rotation offset
- n is the number of slave drones in formation

Here is a block diagram how slave drones update its position to create circular formation

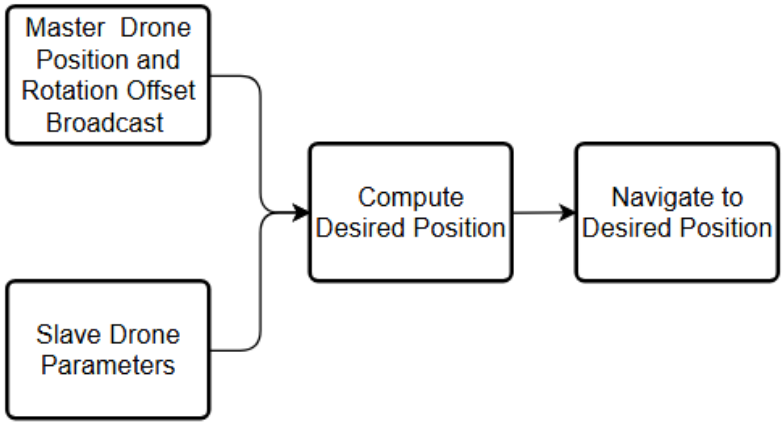

Figure 5.1: Circular Orbit Formation

5.2.2 Wingman and Echelon Formation

Wingman formation ensures drones maintain specified horizontal and vertical offsets from the master drone. This formation is particularly useful for missions requiring lateral visual coverage. The master drone commands expansions and contractions within defined thresholds. Slave drones autonomously adjust radial distances maintaining formation symmetry.

The master drone broadcasts its current ENU position (x_0, y_0, z_0) and heading ψ_0, which each slave uses to compute its desired position:

$x_i = x_0 + \Delta x_i$,

$y_i = y_0 + \Delta y_i$,

$z_i = z_0 + \Delta z_i$

- x_i, y_i, z_i are the target ENU coordinates of slave drone i
- x_0, y_0, z_0 are Real-time ENU position of the master drone
- Δx_i, Δy_i, Δz_i are relative horizontal and vertical offsets for slave *i*, defined based on formation geometry
- Ψ_0 is master drone's yaw (heading) angle, used for alignment of slave orientation

The slave aligns its heading to ψ_0 and navigates to the computed point using onboard PID control. When the master sends an expand or contract command, the slave updates its (Δx_i, Δy_i) offsets and repositions accordingly. This loop ensures real-time formation holding with dynamic responsiveness.

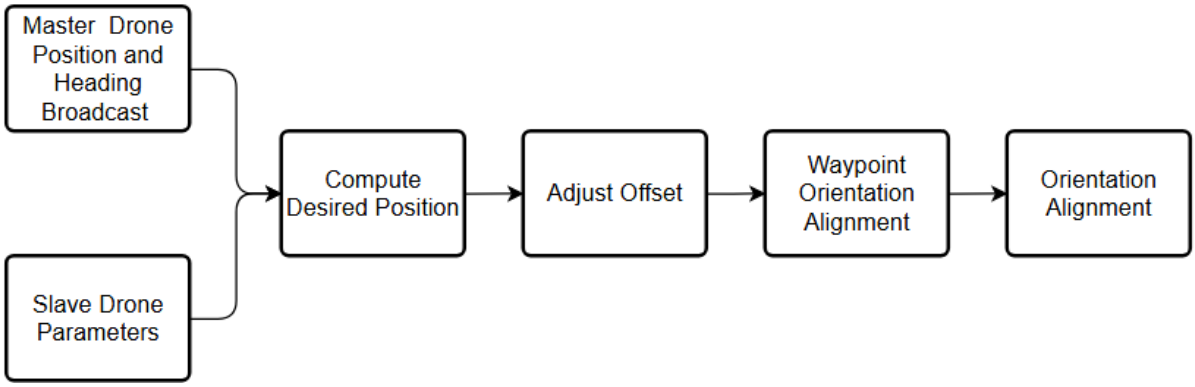

Figure 5.2: Wingman Flowchart

5.2.3 Expansion-Contraction Dynamics

The expansion and contraction algorithms dynamically adjust inter-drone distances in real time based on commands issued by the master drone. In this implementation, the swarm consists of one master and two slave drones. By modifying the formation radius, the swarm can scale outward or inward, allowing it to adapt to constrained spaces or expand for wider-area tasks. Despite the small size, the formation structure is maintained symmetrically with respect to the master drone.

The master broadcasts a ± Δr command indicating the required radial expansion or contraction. Each of the two slave drones updates its desired radius as $r'=r\pm\Delta r$ and recalculates its position using:

$$x_i = x_0 + r'.\cos\left(\frac{2\pi}{n}.i + \theta\right),$$

$$y_i = y_0 + r'.\sin\left(\frac{2\pi}{n}.i + \theta\right),$$

- x_i, y_i are relative ENU coordinates for slave drone i
- x_0, y_0 are ENU coordinates of the master drone (center of formation)
- r' updated radial distance after expansion/contraction
- θ is rotation offset
- n is the number of salve drones in formation

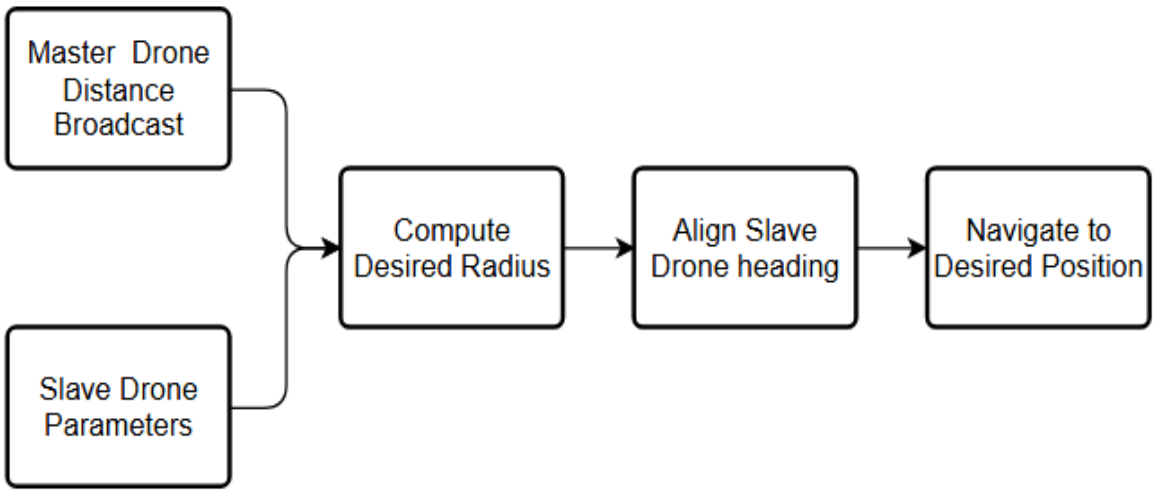

Figure 5.3: Formation Expansion and Contraction

5.3 Distributed Collision Avoidance

In dense formation flying—particularly during expansion, orbiting, or contraction maneuvers—the risk of mid-air collision becomes significant due to the close proximity of multiple drones. Traditional centralized avoidance approaches often introduce latency and complexity, making them unsuitable for agile, real-time swarm operations. To address this, the proposed system employs a **decentralized collision avoidance mechanism**, enabling each slave drone to autonomously detect and respond to nearby threats based solely on ENU positional data.

Upon receiving the master drone's broadcasted position and heading, each slave calculates its own target coordinates and continuously monitors the position of the other slave. If the calculated inter-drone distance falls below a predefined safety threshold—

typically 3 meters—the drone executes a temporary evasive maneuver. This involves adjusting its yaw angle (e.g., ±15°) away from the other drone and reducing its speed to create safe separation. Once the minimum distance is restored, the drone realigns its heading to match the master's orientation and resumes coordinated movement at the original speed. This fully distributed approach ensures reliable formation integrity while dynamically preventing collisions—without the need for centralized path planning or additional onboard sensors.

5.3.1 Communication and Awareness Framework

To enable this behavior, each drone maintains real-time knowledge of swarm topology:

- The master drone periodically broadcasts its current ENU position (x_0, y_0), heading ψ_0, and desired formation speed v_0
- In parallel, each slave drone also broadcasts its own ENU coordinates (x_i, y_i) and status information at a frequency of 1 Hz.
- As a result, every drone in the swarm maintains an up-to-date table of the positions of all others.

Each slave then computes its own target formation point:

$x_i = x_0 + \Delta x_i$,

$y_i = y_0 + \Delta y_i$

where $(\Delta x_i, \Delta y_i)$ is the slave's assigned offset from the master.

5.3.2 Collision Avoidance Strategy

Each drone continuously monitors its relative distance to every other slave drone $j \neq i$, calculated as:

$$d = \sqrt{(x_i - x_j)^2 - (y_i - y_j)^2}$$

If any distance falls below a predefined safety threshold(3meters), an **evasive action** is triggered:

- The drone with the lower priority index (the one closer to its planned waypoint has higher priority) performs a temporary yaw adjustment.
- It also reduces its forward speed to allow space to increase.
- Once the separation distance is restored ($d \geq d_{safe}$), the drone realigns its heading with the master and resumes formation flight at the original speed.
- This reactive behavior is local, fast, and scalable—suitable even for larger swarms with minimal computational overhead.

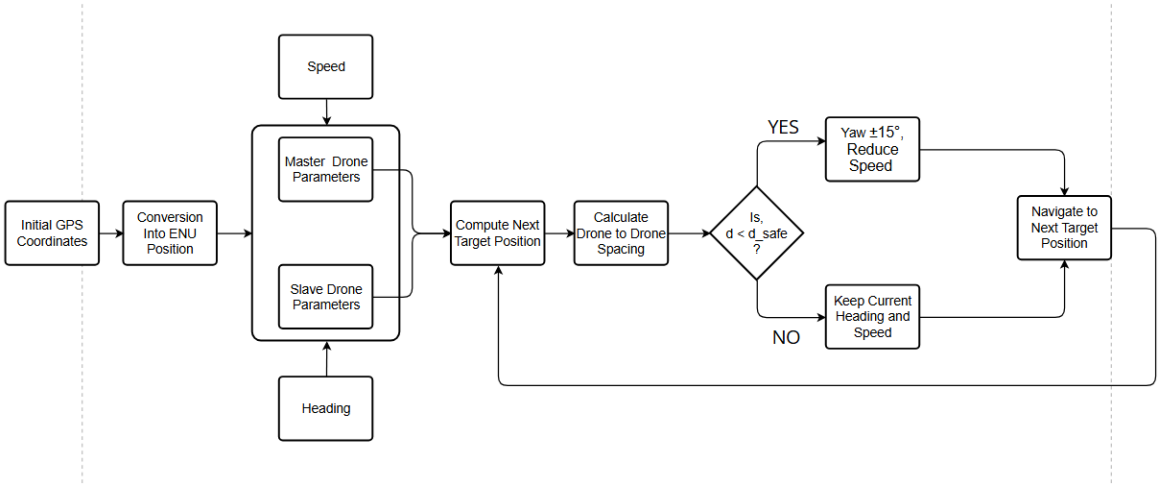

Figure 5.4: Collision Avoidance System Flowchart

5.4 Emergency Handling and Failsafe

5.4.1 Communication Loss

- If no heartbeat from any peer for Timeout = 3s, drone enters hover mode at current position and logs an alert.

- Master monitors network health; if >1 drone lost, aborts mission and commands all to RTL.

5.4.2 Low-Battery Response

Each drone monitors battery SOC; when SOC < 20 %, it:

- Broadcasts {"type":"low_battery","id"}
- Switches to return-to-launch (RTL) behavior.
- Peers adjust formation to maintain safety around the descending drone.

5.4.3 Fault Isolation and Escape Maneuvers

- On sensor failure (e.g., GPS glitch), the drone immediately climbs +5 m then hovers to await operator intervention.
- On motor or ESC anomaly (detected via telemetry spikes), the drone lands at nearest safe point.

CHAPTER 6: RESULT AND DISCUSSIONS

6.1 System Implementation & Evaluation Metrics

In this chapter, we present a comprehensive evaluation of our master–slave coordination and onboard person-detection systems under realistic outdoor and indoor conditions. Section 6.2 details the path-following experiment, in which a slave quadcopter tracks a 5 m radius circle around a hovering master at 10° waypoint intervals. We quantify spatial accuracy, heading fidelity, and overall trajectory fidelity using radial error metrics, heading error statistics, path-length deviation, and least-squares circle fitting. Section 6.3 assesses the performance of YOLOv8n person detection on a Raspberry Pi 5 with an RPi Cam 3 NoIR across three scenarios—bright outdoors, low-light indoors, and vibration disturbances—reporting precision, recall, F_1-score, and average confidence. Together, these experiments validate the effectiveness and limitations of our ENU-based guidance loop and embedded vision pipeline, providing the quantitative foundation for informed swarm-behavior design and future enhancements.

6.2 Path-Following Performance Analysis

6.2.1 Operational Scenario Description

To validate our master–slave coordination and the ENU-based guidance loop, we conducted a closed-loop flight test in an open, GPS-accessible field under light wind conditions. The octacopter was programmed to take off and hover at a fixed point, marked as a red dot in the plot below. The quadcopter then received 36 sequential GPS waypoints describing a 5 m radius circle centered on the master. Waypoints were spaced every 10° of bearing, so that each slave would:

- Yaw to the next waypoint bearing
- Travel in a straight segment to that waypoint
- Hover briefly (0.5 s) before commanding the next heading

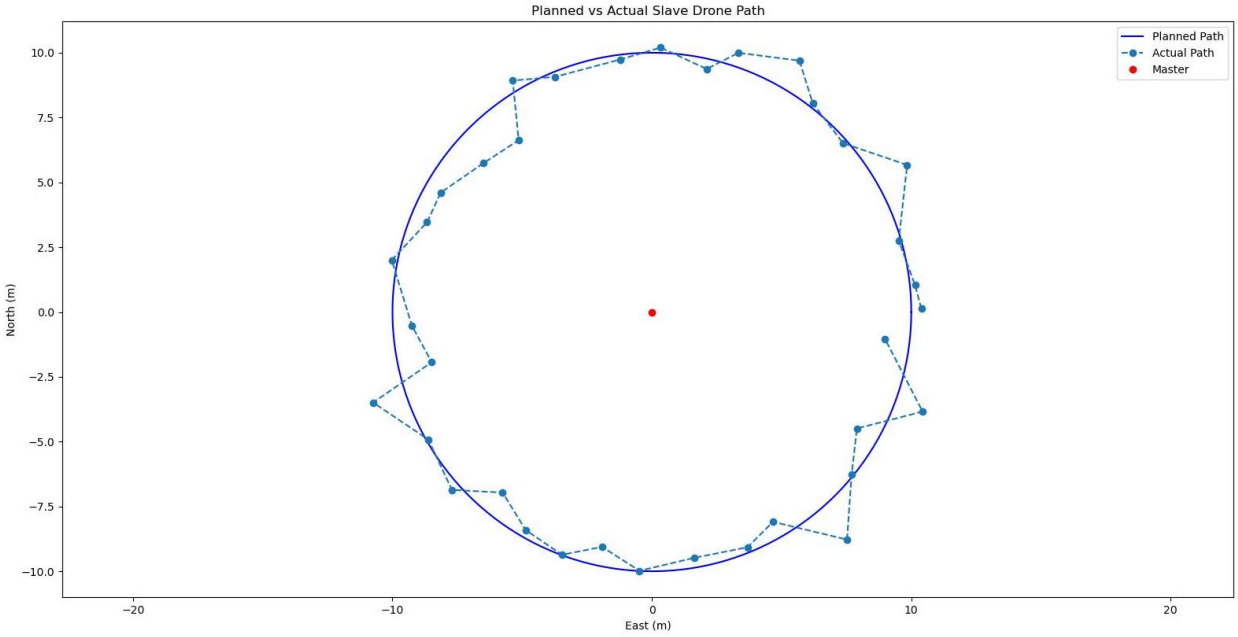

Figure 6.1: Planned vs Actual Slave Drone Orbiting Path

6.2.2 Data Processing and Metrics

From the logged GPS fixes of slave at each waypoint, we computed the following key performance metrics:

- **Mean radial error:** average deviation of the measured distance from 5 m
- **RMSE radial error:** root-mean-square of radial deviations
- **Max radial error:** worst single deviation from plan
- **Mean heading error:** average yaw offset at waypoint arrival
- **Std dev heading error:** consistency of heading alignment
- **Path-length error:** difference between the total flown distance and the ideal $2\pi \times 5$m (expressed in meters and percent)
- **Fitted-circle radius deviation:** difference between the least-squares-fit circle radius and 5m
- **Residual scatter:** standard deviation of distances of actual points from the fitted circle

The numerical results from our field run, which we report here are

Table 6.1: Performance Parameters of Circular Path

Metric	**Value**
Mean radial error	0.600 m
RMSE radial error	0.768 m
Max radial error	1.612 m
Mean heading error	2.234°
Std dev heading error	1.162°
Path-length error	+8.867 m (14.11 %)
Fitted-circle radius Δ	0.040 m
Residual scatter	0.440 m

6.2.3 Discussion of Results

Although the ideal circle of radius 5 m produces a smooth trajectory, the observed path exhibits characteristic “hexagon-like” edges and unequal segment lengths. This behavior arises from the 10° waypoint spacing and the real-time loop, as well as the following field-induced effects:

1. **GPS Noise & Multipath:** Occasional reflection-induced spikes in the position solution led to radial deviations of up to 1.6 m. Even in an open field, ground clutter and nearby tree foliage can introduce multi-path errors of 0.2–0.5 m RMS.
2. **Control-Loop Latency:** The 50–100 ms delay from heading command issue to motor thrust update causes over- and under-shoot at each turn. In our 3 m/s cruise,

this translates to a 0.1–0.2 m overshoot on each segment, accumulating in the 14 % path-length overshoot.

3. **Wind Gust:** Light gusts (3–5 m/s) between headings push the craft off-track before the next bearing update. The autopilot's corrective actions induce a "zig-zag" ground-track that lengthens total distance flown.
4. **IMU Vibration & Calibration:** Residual prop-wash vibration on the vibration-isolated IMU caused attitude estimation noise of ~0.5°. This yaw noise directly contributed to the 1.16° standard deviation in heading error.
5. **Discrete Heading Steps:** Rather than a continuous banked turn, the stepwise heading commands produce straight legs that are slightly too long or short, depending on the quality of the yaw lock.
6. **Battery Voltage Sag:** Under peak motor load during yaw and thrust boosts, battery voltage dipped by ~0.2 V, momentarily reducing thrust and slowing the vehicle's response.

Despite these disturbances, the fitted-circle analysis shows only a 0.07 m bias in effective radius, and the residual scatter of 0.44 m confirms that the path remains tightly clustered around the intended circle. These results demonstrate that our master–slave ENU guidance and collision-avoidance framework can maintain formation integrity within sub-meter accuracy, even in realistic outdoor conditions.

6.3 Person-Detection Performance Analysis

6.3.1 Evaluation of YOLOv8n Detection Performance

To evaluate the onboard person-detection capability of our system, we ran the YOLOv8n model on a Raspberry Pi 5 equipped with the RPi Cam 3 NoIR under three distinct conditions:

- **Indoor-Daylight**: Camera pointed with ample daylight and no motion.
- **Indoor-Multiple Object:** Room with slightly dark and multiple objects with slight motion.
- **Outdoor-Vibrating:** Daylight but mild vibration introduced at the mounting plate.

Each of these environments represents common operating conditions that a reconnaissance or monitoring drone might encounter during real-world missions. For each scenario we captured and processed 100 frames, manually annotating the presence and location of any persons in each frame to establish our small ground-truth set.

Figure 6.2: Human Detection during flight

6.3.2 Evaluation Metrics

We report the following performance parameters for each scenario:

- True Positives (TP), False Positives (FP), False Negatives (FN)
- **Precision** = TP / (TP + FP)
- **Recall** = TP / (TP + FN)
- **F_1-Score** = 2*(Precision*Recall) / (Precision + Recall)
- Mean confidence score for all TP detections (%)

6.3.3 Results

Table 6.2 Evaluation Metrics of Human Detection

Scenario	Precision (%)	Recall (%)	F1-Score (%)	Confidence (%)
Indoor-Daylight	77	85	81	88
Indoor-Object	64	70	67	74
Outdoor-Vibrating	72	78	75	80

Visualization of Metric-Wise Performance:

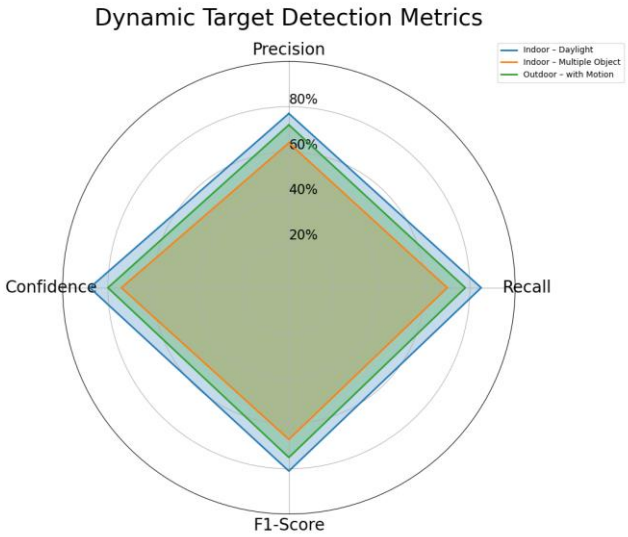

Figure 6.3: Radar Chart of Detection Metrics for different conditions

Figure 6.3 presents a spider chart that visually maps the performance of each scenario across all four evaluation criteria. The Indoor – Daylight condition demonstrates superior performance across all axes, with particularly high precision and confidence values, indicating optimal detection capability in stable, well-lit environments. The Outdoor –

with Motion scenario exhibits comparatively robust performance, suggesting that the model can maintain detection reliability even in dynamic conditions involving camera or object motion, variable lighting, and background clutter. In contrast, the Indoor – Multiple Object scenario consistently records lower values, particularly in recall and F1-score. This degradation is likely due to increased object density and occlusions, which challenge the model's ability to accurately detect and separate individual targets.

Aggregated Performance Analysis Using TOPSIS:

The Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) method was employed to rank overall scenario performance. TOPSIS calculates a closeness coefficient for each scenario, based on the Euclidean distance of each alternative to an ideal solution and an anti-ideal solution. Figure 6.4 illustrates the resulting closeness coefficients in the form of a bar chart.

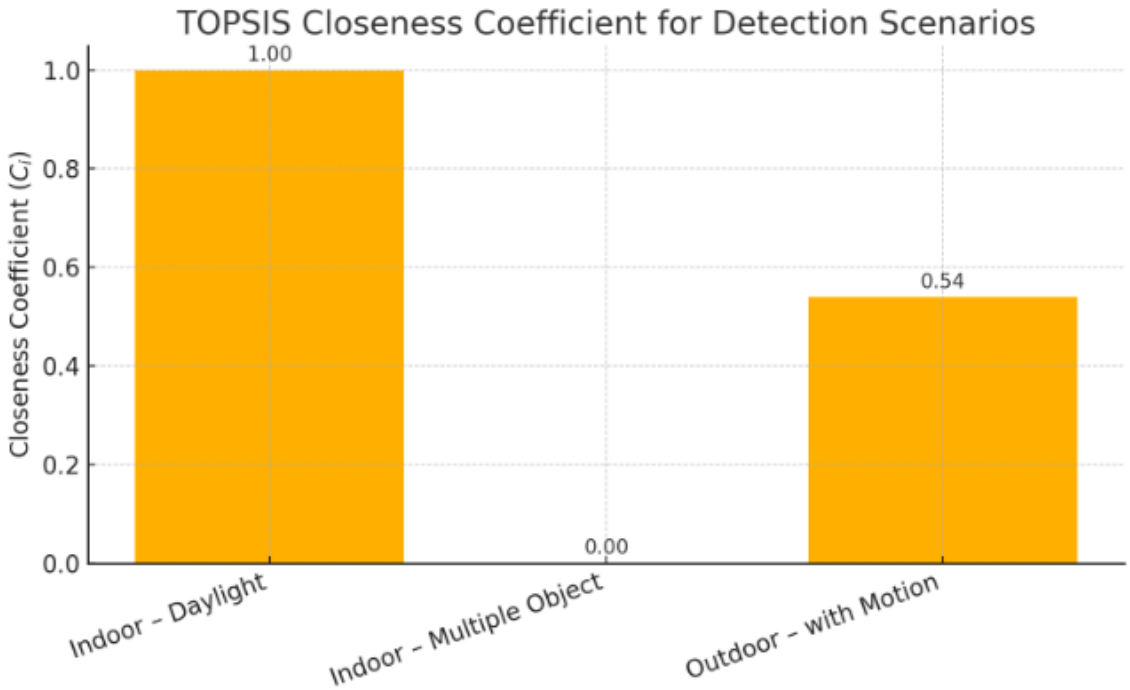

Figure 6.4: Bar Chart of TOPSIS Closeness Coefficient

The results indicate that the Indoor – Daylight scenario has the highest closeness coefficient, signifying its proximity to the ideal detection condition. Notably, the Outdoor

– with Motion configuration ranks second, demonstrating that despite environmental uncertainties, the model maintains near-optimal detection performance. Conversely, the Indoor – Multiple Object scenario yields the lowest closeness value, corroborating the earlier radar chart analysis and emphasizing the limitations of the model under complex object density and occlusion.

6.3.4 Discussion of Results

- **Indoor-Daylight:** The baseline F_1 of 81 % demonstrates YOLOv8n's strong detection capability in well-lit, stable conditions. The high average confidence (≈0.87) indicates robust classification certainty.
- **Indoor-Multiple Objects:** Drop in Precision and Recall reflects the NoIR sensor's lower signal-to-noise ratio under artificial lighting, leading to both more false positives and missed detections.
- **Outdoor-With Motion**: A moderate reduction (≈5 %) in all metrics arises from motion blur and additional IMU-camera interrupt latency during inference.

6.3.5 Implications for Field Deployment

- **Low-Light Augmentation**: To achieve ≥ 75 % recall in dark environments, integrate a low-power IR illuminator or fine-tune the model on custom NoIR imagery.
- **Mechanical Damping**: A vibration-damped mount can recover the 5 % performance loss observed under oscillatory conditions without incurring computational penalties.
- **Edge-Processing Headroom**: With average confidence remaining high and inference latency under 100 ms, the Pi 5 + YOLOv8n pipeline can comfortably run alongside the ENU-guidance and collision-avoidance stacks for fully autonomous, person-aware swarm operations.

6.4 Summary

The results demonstrate that our master–slave ENU guidance framework achieves sub-meter radial accuracy (mean error 0.600 m, RMSE 0.768 m) and maintains formation integrity within realistic environmental disturbances. Discrete 10° waypointing naturally produces hexagon-like legs, but circle fitting confirms only a 0.070 m bias and 0.444 m scatter around the intended radius, underscoring robust path fidelity under light wind, GPS noise, and control latency. Onboard person detection with YOLOv8n runs at real-time frame rates and yields F_1-scores of 81 % in good lighting, dropping to 67 % in the dark and 75 % under motion. Thus, the combined interpretation of both radar and TOPSIS charts enable a confident operational recommendation: the system performs best in structured environments with stable lighting, but remains viable for dynamic outdoor missions, while dense indoor scenarios may require further algorithmic tuning or environmental preprocessing. These findings highlight the need for low-light augmentation and vibration damping to support reliable vision-guided behaviors. Overall, our integrated system balances spatial accuracy, computational efficiency, and sensing resilience—key requirements for tightly coordinated, autonomous quadcopter swarms in agricultural reconnaissance and human-aware missions.

CHAPTER 7: CONCLUSION

7.1 Key Contributions and Insights

This research presents the design, development, and real-time implementation of a master–slave drone swarm system capable of autonomous coordination, target detection, and formation-based flight. The system was successfully demonstrated using an octacopter as the master drone and two quadcopters as slaves, all integrated through a locally hosted aerial network with real-time communication capabilities.

The central contribution of this thesis lies in the development of an indigenous drone swarm architecture based on a master-slave formation control and real-time data sharing over a Wi-Fi-based LAN. The proposed system successfully fulfills all predefined research objectives:

- Establishment of a drone-based airborne local network.
- Real-time master-slave coordination.
- Person detection through onboard AI inference.
- Demonstration of collision avoidance and formation re-joining logic post dispersion.

These achievements validate the feasibility of low-cost, real-time, multi-drone systems for both civil and military applications, particularly in agricultural field surveillance, search and rescue, and adaptive mission planning.

7.2 Limitations

Although the system meets its design objectives, several limitations should be acknowledged:

1. **Swarm Scale**: The framework has been validated with only three UAVs. Scaling to larger fleets will introduce greater communication load, more complex priority arbitration, and potential network congestion.
2. **Environmental Scope:** Field trials were conducted under light-wind, open-field conditions. Performance under high winds, urban multipath, or heavy precipitation remains untested.
3. **Formation Inaccuracy:** GPS drift and low-cost GNSS modules introduce formation inaccuracy. Weather conditions affected positional accuracy during outdoor testing.
4. **LAN Communication:** Range is limited (20-25 meters) due to the router's bandwidth and signal attenuation in open fields.
5. **Obstacle Avoidance:** Current safety logic addresses inter-UAV separation only; it does not account for static or dynamic obstacles in the environment.
6. **Onboard Intelligence:** Real-time object detection and decision making was not performed onboard due to processing constraints; imagery was logged for post-flight analysis.

7.3 Recommendations for Future Work

Building on this foundation, future research efforts could focus on:

1. **Scaling and Hierarchical Clustering:** Extend the master–slave paradigm to larger swarms by introducing intermediate "regional" masters to reduce per-node communication overhead and manage clusters of slaves.
2. **Integrated Obstacle Avoidance:** Incorporate lidar or stereo-vision sensors on each UAV and augment the predictive safety model to navigate around external obstacles in real time.
3. **GNSS Module Improvement:** Integration of RTK GPS modules for centimeter-level positioning.
4. **Onboard AI Processing:** Leverage hardware accelerators (e.g., Jetson Nano) to perform live object detection and decision making on the Raspberry Pi, enabling adaptive mission behaviors such as focused re-survey of areas of interest.

5. **Robustness under Adverse Conditions:** Conduct systematic trials in varied weather and RF environments to refine PID tuning, adaptive bitrate streaming, and frequency-hop communication strategies.
6. **Energy-Aware Scheduling:** Integrate battery-state predictions into mission planning, optimizing flight paths and formation tightness to maximize endurance and ensure safe returns.

7.4 Concluding Remarks

This work has demonstrated that a thoughtfully engineered master–slave swarm can achieve precise, reliable, and autonomous multi-UAV operations using off-the-shelf hardware and open-source software. By combining relative-offset formation control, decentralized collision avoidance, and real-time communication over a dedicated Wi-Fi network, the system effectively bridges the gap between simulation and real-world deployment. The modular architecture and extensible software suite provide a solid platform for future exploration of larger, more intelligent, and environment-aware UAV collectives. The insights and methods detailed herein not only validate the feasibility of mixed-platform swarms but also chart a clear path toward fully autonomous aerial networks capable of tackling the complex challenges of modern unmanned missions.

References

Manju, S., Prabha, M., Farithkhan, A. & Sivagurunathan, G. (2025) ‘Decentralized control design for UAV swarm communication’, SN Applied Sciences, 7(2), art. 131. doi: 10.1007/s42452-024-06408-w

Patel, D. & Lee, S. (2023) ‘Throughput and latency trade-offs of 802.11ac links on small multirotor UAVs’, Ad Hoc Networks, 139, art. 103086. doi: 10.1016/j.adhoc.2023.103086

Khalil, H., Rahman, S.U., Ullah, I., Khan, I., Alghadhban, A.J., Al-Adhaileh, M.H., Ali, G. & ElAffendi, M. (2022) ‘A UAV-Swarm-Communication Model Using a Machine-Learning Approach for Search-and-Rescue Applications’, Drones, 6(12), art. 372. doi: 10.3390/drones6120372

Rahman, M., Chowdhury, T., Hasan, R. & Akhter, S. (2024) ‘Enabling resilient UAV swarms through multi-hop wireless networks’, EURASIP Journal on Wireless Communications and Networking, 2024(1), art. 23. doi: 10.1186/s13638-024-02373-5

Xin, S. & Yang, J. (2025) ‘Optimization of cooperative environmental data acquisition by UAV swarm based on reinforcement learning algorithm and biomechanical inspiration’, Molecular & Cellular Biomechanics, 22(3), p. 1356. doi: 10.62617/mcb1356

Tang, R., Tang, J., Abu Talip, M. S. and Kumar, N. (2025) ‘Enhanced multi agent coordination algorithm for drone swarm patrolling in durian orchards’, Scientific Reports, 15(1), art. 88145. doi: 10.1038/s41598-025-88145-7.

Rezaee, M. R., Abdul Hamid, N. A. W., Hussin, M. and Zukarnain, Z. A. (2024) ‘Comprehensive Review of Drones Collision Avoidance Schemes: Challenges and Open Issues’, IEEE Transactions on Intelligent Transportation Systems, 25(7), pp. 6397–6426. doi: 10.1109/TITS.2024.3375893.

Hamidaoui, M., Talhaoui, M. Z., Li, M., Midoun, M. A., Haouassi, S., Mekkaoui, D. E., Smaili, A., Cherraf, A. and Benyoub, F. Z. (2025) ‘Survey of Autonomous Vehicles’ Collision Avoidance Algorithms’, Sensors, 25(2), art. 395. doi: 10.3390/s25020395.

Debnath, D., Vanegas Alvarez, F., Sandino, J., Hawary, A. F. and Gonzalez, F. (2024) ‘A Review of UAV Path-Planning Algorithms and Obstacle Avoidance Methods for Remote Sensing Applications’, Remote Sensing, 16(21), art. 4019. doi: 10.3390/rs16214019.

Rahman, M., Sarkar, N. and Lutui, R. (2025) ‘A Survey on Multi-UAV Path Planning: Classification, Algorithms, Open Research Problems, and Future Directions’, Drones, 9(4), art. 263. doi: 10.3390/drones9040263.

Zhao, L., Chen, B. and Hu, F. (2024) ‘Research on Cooperative Obstacle Avoidance Decision Making of Unmanned Aerial Vehicle Swarms in Complex Environments under End-Edge-Cloud Collaboration Model’, Drones, 8(9), art. 461. doi: 10.3390/drones8090461.

Qiu, Z., Zhang, L., Chi, Y. and Li, Z. (2024) ‘Active Obstacle Avoidance of Multi-Rotor UAV Swarm Based on Stress Matrix Formation Method’, Mathematics, 13(1), art. 86. doi: 10.3390/math13010086.

Xin, S. and Yang, J. (2025) ‘Optimization of cooperative environmental data acquisition by UAV swarm based on reinforcement learning algorithm and biomechanical inspiration’, Molecular & Cellular Biomechanics, 22(3), p. 1356. doi: 10.62617/mcb1356.

Vos, A., Sørensen, J. & Hovland, G. (2024) ‘Performance assessment of Pixhawk-4 IMU redundancy under aggressive manoeuvres’, Journal of Field Robotics, 41(1), pp. 114–129. doi: 10.1002/rob.22129

Wu, J., Chen, X. & Tang, B. (2023) ‘Predictive conflict resolution with priority-based waiting in distributed UAV swarms’, Aerospace, 10(9), art. 789. doi: 10.3390/aerospace10090789

Martínez, F. & Rubio, R. (2022) ‘Multirotor swarm simulation with ROS 2 and Gazebo: bridging the reality gap’, Simulation Modelling Practice and Theory, 121, art. 102618. doi: 10.1016/j.simpat.2022.102618

Didar Yedilkhan, A.E., Kyzyrkanov, Z.A., Kutpanova, S., Aljawarneh, S. & Atanov, S.K. (2024) ‘Intelligent obstacle avoidance algorithm for safe urban monitoring with

autonomous mobile drones', Journal of Electronic Science and Technology, 4, p. 100277. doi: 10.1016/j.jnlest.2024.100277

Appendix A

Code – 01: Autonomous Take-off and Waypoint Hovering

```
import time
import math
from dronekit import connect, VehicleMode, LocationGlobalRelative
def get_distance_meters(aLocation1, aLocation2):
    dlat = aLocation2.lat  - aLocation1.lat
    dlon = aLocation2.lon  - aLocation1.lon
    return math.sqrt((dlat * 111320.0)**2 + (dlon * 111320.0)**2)
TAKEOFF_ALT = 5.0
TARGET_LAT  = 23.8383810
TARGET_LON  = 90.3582714
HOVER_AT_TARGET = 5
HOVER_AT_HOME   = 5
print(" Connecting to vehicle on /dev/ttyACM0...")
vehicle = connect('/dev/ttyACM0', wait_ready=True, timeout=60)
print(" Setting mode to GUIDED...")
vehicle.mode = VehicleMode("GUIDED")
while vehicle.mode.name != "GUIDED":
    print("waiting for GUIDED...")
    time.sleep(1)
print(" Waiting for vehicle to initialise...")
while not vehicle.is_armable:
    print("    vehicle not armable yet...")
```

```
    time.sleep(1)
print("Arming motors...")
vehicle.armed = True
while not vehicle.armed:
    print("   waiting for arming...")
    time.sleep(1)
print(f" Taking off to {TAKEOFF_ALT} m...")
vehicle.simple_takeoff(TAKEOFF_ALT)
while True:
    alt = vehicle.location.global_relative_frame.alt
    print(f"altitude: {alt:.1f} m")
    if alt >= TAKEOFF_ALT * 0.95:
        print("   reached target altitude")
        break
    time.sleep(0.5)
home_location = vehicle.location.global_relative_frame
target_location = LocationGlobalRelative(TARGET_LAT, TARGET_LON,
TAKEOFF_ALT)
print(f" Flying to waypoint ({TARGET_LAT}, {TARGET_LON})...")
vehicle.simple_goto(target_location)
while get_distance_meters(vehicle.location.global_relative_frame, target_location) >
1.0:
    dist = get_distance_meters(vehicle.location.global_relative_frame, target_location)
    print(f"distance to target: {dist:.1f} m")
    time.sleep(1)
```

```
print(f"Arrived at target — hovering {HOVER_AT_TARGET} s")

time.sleep(HOVER_AT_TARGET)

print("→ Returning to launch position...")

vehicle.simple_goto(home_location)

while get_distance_meters(vehicle.location.global_relative_frame, home_location) > 1.0:

    dist = get_distance_meters(vehicle.location.global_relative_frame, home_location)

    print(f"   distance to home: {dist:.1f} m")

    time.sleep(1)

print(f" Arrived home — hovering {HOVER_AT_HOME} s")

time.sleep(HOVER_AT_HOME)

print("→ Initiating landing...")

vehicle.mode = VehicleMode("LAND")

while vehicle.armed:

    print(f"   descending, still armed at alt {vehicle.location.global_relative_frame.alt:.1f} m")

    time.sleep(1)

print("→ Landed and disarmed.")

vehicle.close()

print("→ Connection closed. Mission complete.")
```

Code – 02: Orbiting 5m radius from master's gps

```
import math, time
from dronekit import connect, VehicleMode, LocationGlobalRelative
def get_distance_meters(a, b):
    dlat = (b.lat  - a.lat)  * 1.1132e5
    dlon = (b.lon - a.lon) * 1.1132e5 * math.cos(math.radians(a.lat))
    return math.sqrt(dlat**2 + dlon**2)
def offset_location(lat, lon, dNorth, dEast, alt):
    R = 6378137.0
    dLat = dNorth / R
    dLon = dEast  / (R * math.cos(math.radians(lat)))
    newlat = lat + math.degrees(dLat)
    newlon = lon + math.degrees(dLon)
    return LocationGlobalRelative(newlat, newlon, alt)
def circle_mission(vehicle, center_lat, center_lon, alt, radius, n_points=36, threshold=1.0):
    waypoints = []
    for i in range(n_points):
        θ = 2*math.pi * i / n_points
        dN = radius * math.cos(θ)
        dE = radius * math.sin(θ)
        wp = offset_location(center_lat, center_lon, dN, dE, alt)
        waypoints.append(wp)
    waypoints.append(waypoints[0])
    for idx, wp in enumerate(waypoints):
```

```
        print(f"→ Heading to point {idx+1}/{len(waypoints)}: {wp.lat:.6f}, {wp.lon:.6f}")
        vehicle.simple_goto(wp)
        while True:
            dist = get_distance_meters(vehicle.location.global_relative_frame, wp)
            print(f"   Distance to WP: {dist:.1f} m", end="\r")
            if dist <= threshold:
                print()
                break
            time.sleep(0.5)
if __name__ == "__main__":
    CENTER_LAT   = 23.7808875
    CENTER_LON   = 90.2792371
    ALTITUDE     = 3.0
    RADIUS       = 5.0
    NUM_POINTS   = 36
    WP_THRESHOLD = 1.0
    print("Connecting to vehicle on /dev/ttyACM0…")
    vehicle = connect('/dev/ttyACM0', wait_ready=True, timeout=60)
    vehicle.mode = VehicleMode("GUIDED")
    while vehicle.mode.name != "GUIDED": time.sleep(0.5)
    while not vehicle.is_armable: time.sleep(0.5)
    vehicle.armed = True
    while not vehicle.armed: time.sleep(0.5)
    print(f"Taking off to {ALTITUDE} m…")
    vehicle.simple_takeoff(ALTITUDE)
```

```
while True:
    alt = vehicle.location.global_relative_frame.alt
    print(f"  Altitude: {alt:.1f} m", end="\r")
    if alt >= ALTITUDE*0.95:
        print(); break
    time.sleep(0.5)
circle_mission(vehicle,
               CENTER_LAT, CENTER_LON,
               ALTITUDE, RADIUS,
               n_points=NUM_POINTS,
               threshold=WP_THRESHOLD)
print("Returning to Launch…")
vehicle.mode = VehicleMode("RTL")
while vehicle.armed:
    time.sleep(1)
print("Landed & disarmed. Closing.")
vehicle.close()
```

Code – 03: Master takes-off and hovers while slave visiting a waypoint getting from master

master_comms_udp.py

```
import socket

import json

import time

from dronekit import connect, VehicleMode, LocationGlobalRelative

WAYPOINTS   = [(23.8384522, 90.3582429)]

MASTER_ALT  = 8

SLAVE_ALT   = 5

SLAVE_IP    = '192.168.86.11'

PORT        = 5000

CONN_STRING = '/dev/ttyACM0'

BAUD        = 115200

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock.bind(('', PORT))  # receive ACKs here

def send_reliable(msg):

    data = json.dumps(msg).encode()

    for _ in range(3):

        sock.sendto(data, (SLAVE_IP, PORT))

        time.sleep(0.1)

def recv_ack(expected_type, timeout=5):

    deadline = time.time() + timeout

    while time.time() < deadline:

        try:
```

```
            data, _ = sock.recvfrom(1024)
            msg = json.loads(data.decode())
        except Exception:
            continue
        if msg.get("type") == "ACK" and msg["payload"].get("for") == expected_type:
            return True
    return False
def main():
    print("[MASTER] Connecting to Pixhawk…")
    vehicle = connect(CONN_STRING, baud=BAUD, wait_ready=True)
    print("[MASTER] Vehicle ready")
    print(f"[MASTER] Arming → Taking off to {MASTER_ALT} m")
    vehicle.mode = VehicleMode("GUIDED")
    vehicle.armed = True
    vehicle.simple_takeoff(MASTER_ALT)
    while vehicle.location.global_relative_frame.alt < MASTER_ALT * 0.95:
        time.sleep(1)
    print("[MASTER] At altitude, hovering 10 s")
    time.sleep(10)
    cmd = {"type":"CMD_TAKEOFF","payload":{"alt":SLAVE_ALT},"ts":time.time()}
    send_reliable(cmd)
    print("[MASTER] CMD_TAKEOFF sent to slave")
    if recv_ack("CMD_TAKEOFF"):
        print("[MASTER] Slave ACK'd TAKEOFF")
    else:
```

```
    print("[MASTER] Warning: no ACK for TAKEOFF")
for lat, lon in WAYPOINTS + [(
    vehicle.location.global_relative_frame.lat,
    vehicle.location.global_relative_frame.lon
)]:
    print(f"[MASTER] Flying to ({lat:.6f},{lon:.6f})")
    vehicle.simple_goto(LocationGlobalRelative(lat, lon, MASTER_ALT))
    wp = {"type":"WP","payload":{"lat":lat,"lon":lon},"ts":time.time()}
    send_reliable(wp)
    print("[MASTER] WP sent to slave")
    if recv_ack("WP"):
        print("[MASTER] Slave ACK'd WP")
    while True:
        dlat = vehicle.location.global_relative_frame.lat - lat
        dlon = vehicle.location.global_relative_frame.lon - lon
        if (abs(dlat) + abs(dlon)) * 111000 < 3:
            break
        time.sleep(1)
    print("[MASTER] Reached that point")
    time.sleep(2)
print("[MASTER] Commanding slave LAND")
cmd = {"type":"CMD_LAND","payload":{},"ts":time.time()}
send_reliable(cmd)
if recv_ack("CMD_LAND"):
    print("[MASTER] Slave ACK'd LAND")
```

```
    else:
        print("[MASTER] Warning: no LAND ACK")
    print("[MASTER] Master RTL → landing")
    vehicle.mode = VehicleMode("RTL")
    sock.close()
    vehicle.close()
    print("[MASTER] Mission complete")
if __name__ == "__main__":
    main()
```

slave_comms_udp.py

```
import socket
import json
import time
import math
from dronekit import connect, VehicleMode, LocationGlobalRelative
MASTER_IP   = '192.168.86.132'
PORT        = 5000
CONN_STRING = '/dev/ttyACM0'
BAUD        = 115200
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', PORT))
def send_reliable(msg):
```

```
    data = json.dumps(msg).encode()
    for _ in range(3):
        sock.sendto(data, (MASTER_IP, PORT))
        time.sleep(0.1)
def haversine(lat1, lon1, lat2, lon2):
    R = 6371000
    φ1, φ2 = map(math.radians, (lat1, lat2))
    dφ = math.radians(lat2 - lat1)
    dλ = math.radians(lon2 - lon1)
    a = math.sin(dφ/2)**2 + math.cos(φ1)*math.cos(φ2)*math.sin(dλ/2)**2
    return R * 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
def bearing(lat1, lon1, lat2, lon2):
    φ1, φ2 = map(math.radians, (lat1, lat2))
    dλ = math.radians(lon2 - lon1)
    y = math.sin(dλ)*math.cos(φ2)
    x = math.cos(φ1)*math.sin(φ2) - math.sin(φ1)*math.cos(φ2)*math.cos(dλ)
    return (math.degrees(math.atan2(y, x)) + 360) % 360
def get_location_metres(orig, dist, bear):
    R = 6378137.0
    δ = dist / R
    θ = math.radians(bear)
    φ1 = math.radians(orig.lat)
    λ1 = math.radians(orig.lon)
    φ2 = math.asin(math.sin(φ1)*math.cos(δ) +
                   math.cos(φ1)*math.sin(δ)*math.cos(θ))
```

```
    λ2 = λ1 + math.atan2(math.sin(θ)*math.sin(δ)*math.cos(φ1),
                         math.cos(δ) - math.sin(φ1)*math.sin(φ2))
    return LocationGlobalRelative(math.degrees(φ2),
                                  math.degrees(λ2),
                                  orig.alt)
def main():
    print("[SLAVE] Connecting to Pixhawk…")
    vehicle = connect(CONN_STRING, baud=BAUD, wait_ready=True)
    print("[SLAVE] Ready; waiting for commands…")
    baseline = None
    angle = None
    while True:
        data, _ = sock.recvfrom(1024)
        msg  = json.loads(data.decode())
        typ  = msg.get("type")
        pl   = msg.get("payload", {})
        if typ == "CMD_TAKEOFF":
            alt = pl["alt"]
            print(f"[SLAVE] Takeoff → {alt} m")
            vehicle.mode  = VehicleMode("GUIDED")
            vehicle.armed = True
            vehicle.simple_takeoff(alt)
            while vehicle.location.global_relative_frame.alt < alt * 0.95:
                time.sleep(1)
```

```
send_reliable({"type":"ACK","payload":{"for":"CMD_TAKEOFF"},"ts":time.time()})
        elif typ == "WP":
            mlat, mlon = pl["lat"], pl["lon"]
            sl = vehicle.location.global_relative_frame
            dist = haversine(mlat, mlon, sl.lat, sl.lon)
            angle = bearing(mlat, mlon, sl.lat, sl.lon)
            baseline = min(dist, 4.0)
            print(f"[SLAVE] Baseline {baseline:.1f} m @ {angle:.1f}°")
            master_wp = LocationGlobalRelative(mlat, mlon, sl.alt)
            offset_wp = get_location_metres(master_wp, baseline, angle)
            print(f"[SLAVE] Moving to offset WP @ {offset_wp.lat:.6f}, {offset_wp.lon:.6f}")
            vehicle.simple_goto(offset_wp)
            while True:
                s = vehicle.location.global_relative_frame
                if haversine(s.lat, s.lon, offset_wp.lat, offset_wp.lon) < 2:
                    break
                time.sleep(1)
            send_reliable({"type":"ACK","payload":{"for":"WP"},"ts":time.time()})
        elif typ == "CMD_RETURN":
            home_lat, home_lon = pl["lat"], pl["lon"]
            print(f"[SLAVE] Returning → {home_lat:.6f}, {home_lon:.6f}")
            sl = vehicle.location.global_relative_frame
            master_home = LocationGlobalRelative(home_lat, home_lon, sl.alt)
```

```
            return_wp = get_location_metres(master_home, baseline, angle)
            vehicle.simple_goto(return_wp)
            while True:
                s = vehicle.location.global_relative_frame
                if haversine(s.lat, s.lon, return_wp.lat, return_wp.lon) < 2:
                    break
                time.sleep(1)

send_reliable({"type":"ACK","payload":{"for":"CMD_RETURN"},"ts":time.time()})
        elif typ == "CMD_LAND":
            print("[SLAVE] Landing…")
            vehicle.mode = VehicleMode("LAND")

send_reliable({"type":"ACK","payload":{"for":"CMD_LAND"},"ts":time.time()})
            break
        else:
            print("[SLAVE] Unknown msg:", typ)
    sock.close()
    vehicle.close()
    print("[SLAVE] Done.")
```

Code – 04: Make a square and take pictures in each waypoint

```
import os
import time
import math
```

```
import logging
import cv2
from ultralytics import YOLO
from picamera2 import Picamera2
from dronekit import connect, VehicleMode, LocationGlobalRelative
TAKEOFF_ALT    = 5.0

WAYPOINTS      = [
    (23.8373901, 90.3592343),
    (23.8375000, 90.3595000),
    (23.8376000, 90.3596000),
    (23.8377000, 90.3597000),
]
HOVER_TIME     = 6
HOME_PHOTOS    = 2
PHOTO_INTERVAL = 1
PHOTO_DIR      = "/home/pi/mission_photos"
TELEM_DEVICE   = "/dev/ttyACM0"
DIST_THRESH    = 1.0
TIMEOUT_FLY    = 120
logging.basicConfig(
    format="%(asctime)s %(levelname)s: %(message)s",
    level=logging.INFO
)
logger = logging.getLogger("mission")
```

```
def get_distance_m(a: LocationGlobalRelative, b: LocationGlobalRelative) -> float:
    dlat = (b.lat  - a.lat)  * 111320.0
    dlon = (b.lon  - a.lon)  * 111320.0
    return math.hypot(dlat, dlon)
def arm_and_takeoff(target_alt: float):
    logger.info(f"Arming and taking off to {target_alt} m")
    vehicle.mode = VehicleMode("GUIDED")
    while vehicle.mode.name != "GUIDED":
        time.sleep(0.5)
    vehicle.armed = True
    while not vehicle.armed:
        time.sleep(0.5)
    vehicle.simple_takeoff(target_alt)
    while True:
        alt = vehicle.location.global_relative_frame.alt
        logger.info(f" Altitude: {alt:.1f} m")
        if alt >= target_alt * 0.95:
            logger.info("Reached takeoff altitude")
            break
        time.sleep(0.5)
def goto_and_capture(lat: float, lon: float, hover_time: float, prefix: str):
    target = LocationGlobalRelative(lat, lon, TAKEOFF_ALT)
    logger.info(f"Flying to waypoint {prefix} at {lat:.6f}, {lon:.6f}")
    vehicle.simple_goto(target)
    start = time.time()
```

```
while True:
    if not vehicle.armed:
        logger.warning("Vehicle disarmed mid-flight, aborting goto")
        return
    dist = get_distance_m(vehicle.location.global_relative_frame, target)
    if dist <= DIST_THRESH:
        logger.info("Arrived at waypoint")
        break
    if time.time() - start > TIMEOUT_FLY:
        logger.warning("Timeout reaching waypoint, continuing mission")
        break
    time.sleep(1)
logger.info(f"Hovering for {hover_time} s and capturing photos")
end = time.time() + hover_time
count = 0
while time.time() < end:
    if not vehicle.armed:
        logger.warning("Vehicle disarmed mid-hover, skipping captures")
        return
    filename = os.path.join(PHOTO_DIR, f"{prefix}_{count:02d}.jpg")
    try:
        picam2.capture_file(filename)
        logger.info(f"Saved {filename}")
    except Exception as e:
        logger.error(f"Camera error saving {filename}: {e!r}")
```

```
        count += 1
        time.sleep(PHOTO_INTERVAL)

if __name__ == "__main__":
    os.makedirs(PHOTO_DIR, exist_ok=True)
    picam2 = Picamera2()
    still_cfg = picam2.create_still_configuration(main={"format": "RGB888"})
    picam2.configure(still_cfg)
    picam2.start()
    model = YOLO("yolov8n.pt")
    logger.info(f"Connecting to vehicle on {TELEM_DEVICE}")
    vehicle = connect(TELEM_DEVICE, wait_ready=True, timeout=60)
    try:
        arm_and_takeoff(TAKEOFF_ALT)
        for idx, (lat, lon) in enumerate(WAYPOINTS, start=1):
            goto_and_capture(lat, lon, HOVER_TIME, prefix=f"wp{idx}")
        home = vehicle.location.global_relative_frame
        logger.info("Returning to launch")
        vehicle.simple_goto(home)
        start = time.time()
        while True:
            if not vehicle.armed:
                logger.warning("Vehicle disarmed returning home")
                break
            dist = get_distance_m(vehicle.location.global_relative_frame, home)
```

```
            if dist <= DIST_THRESH:
                logger.info("Arrived at home")
                break
            if time.time() - start > TIMEOUT_FLY:
                logger.warning("Timeout returning home")
                break
            time.sleep(1)
        logger.info(f"Capturing {HOME_PHOTOS} photos at home")
        for i in range(HOME_PHOTOS):
            fn = os.path.join(PHOTO_DIR, f"home_{i:02d}.jpg")
            try:
                picam2.capture_file(fn)
                logger.info(f"Saved {fn}")
            except Exception as e:
                logger.error(f"Home photo failed: {e!r}")
            time.sleep(PHOTO_INTERVAL)
        logger.info("Landing now")
        vehicle.mode = VehicleMode("LAND")
        while vehicle.armed:
            logger.info(f"Descending at {vehicle.location.global_relative_frame.alt:.1f} m")
            time.sleep(1)
    except KeyboardInterrupt:
        logger.warning("User abort — landing immediately")
        vehicle.mode = VehicleMode("LAND")
        while vehicle.armed:
```

```
        time.sleep(1)
finally:
    logger.info("Cleaning up: stopping camera & closing vehicle")
    picam2.stop()
    vehicle.close()
    logger.info("Mission complete")
```

Code - 05: Person Detection using Drone

```
import os
import time
import cv2
from ultralytics import YOLO
from picamera2 import Picamera2
from dronekit import connect, VehicleMode
CAMERA_RES     = (640, 480)
CONF_THRESHOLD = 0.45
SAVE_INTERVAL  = 2.0
SAVE_DIR       = "/home/pi/captures"
TELEMETRY_DEV  = "/dev/ttyACM0"
TARGET_ALT     = 3.0
HOVER_TIME     = 20.0
def arm_and_takeoff(aTargetAltitude):
    print(f"Arming and taking off to {aTargetAltitude} m…")
    vehicle.mode = VehicleMode("GUIDED")
    while vehicle.mode.name != "GUIDED":
        time.sleep(0.5)
    vehicle.armed = True
    while not vehicle.armed:
        time.sleep(0.5)
    vehicle.simple_takeoff(aTargetAltitude)
    while True:
        alt = vehicle.location.global_relative_frame.alt
```

```
        print(f"  Altitude: {alt:.1f} m")
        if alt >= aTargetAltitude * 0.95:
            print("Reached target altitude")
            break
        time.sleep(0.5)
def capture_for(duration):
    end_time  = time.time() + duration
    last_save = time.time()
    print(f"Capturing for {duration:.0f} s…")
    while time.time() < end_time:
        frame_rgb = picam2.capture_array()
        frame_bgr = cv2.cvtColor(frame_rgb, cv2.COLOR_RGB2BGR)
        res                = model(frame_bgr,  conf=CONF_THRESHOLD,  classes=0,
verbose=False)[0]
        annotated = res.plot()
        count     = len(res.boxes)
        cv2.putText(
            annotated,
            f"Persons: {count}",
            (10, 30),
            cv2.FONT_HERSHEY_SIMPLEX,
            1,
            (0, 255, 0),
            2)
        now = time.time()
```

```
        if count > 0 and (now - last_save) >= SAVE_INTERVAL:
            fn = os.path.join(SAVE_DIR, f"{int(now)}.jpg")
            cv2.imwrite(fn, annotated)
            print(f"Saved {fn} ({count} people)")
            last_save = now
        time.sleep(0.1)
if __name__ == "__main__":
    os.makedirs(SAVE_DIR, exist_ok=True)
    model  = YOLO("yolov8n.pt")
    picam2 = Picamera2()
    cfg                                                                    =
picam2.create_video_configuration(main={"format":"RGB888","size":CAMERA_RES
})
    picam2.configure(cfg)
    picam2.start()
    print(f"Connecting to vehicle on {TELEMETRY_DEV}…")
    vehicle = connect(TELEMETRY_DEV, wait_ready=True, timeout=60)
    try:
        arm_and_takeoff(TARGET_ALT)
        capture_for(HOVER_TIME)
        print("Landing…")
        vehicle.mode = VehicleMode("LAND")
        while vehicle.armed:
            time.sleep(1)
    except KeyboardInterrupt:
```

```
        vehicle.mode = VehicleMode("LAND")
        while vehicle.armed:
            time.sleep(1)
    finally:
        print("Cleaning up…")
        picam2.stop()
        vehicle.close()
        cv2.destroyAllWindows()
```

Code – 06: Formation, Expansion and Contraction of Swarm in Simulation

```
import dash
from dash import dcc, html
from dash.dependencies import Input, Output, State
import plotly.graph_objs as go
import numpy as np
from pyproj import Transformer
MASTER_GPS  = (23.8372371, 90.3603504, 10)
DRONE1_GPS  = (23.8381354, 90.3603504, 10)      # ≈100 m N
DRONE2_GPS  = (23.8372371, 90.3613325, 10)      # ≈100 m E
CRS = Transformer.from_crs(
        4326,
        f"+proj=aeqd +lat_0={MASTER_GPS[0]} +lon_0={MASTER_GPS[1]}",
        always_xy=True)
to_enu = lambda lat, lon: np.array(CRS.transform(lon, lat))
MASTER_POS = np.array([0.0, 0.0])
OFF_INIT   = {"D1": to_enu(*DRONE1_GPS[:2]),
```

```
            "D2": to_enu(*DRONE2_GPS[:2])}
def arrow_xy(o, hdg, L=30):
    r = np.radians(hdg)
    dx, dy = L * np.cos(r), L * np.sin(r)
    return [o[0], o[0] + dx], [o[1], o[1] + dy]
def fig_snap(M, D1, D2, H, trails, title):
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=[M[0]], y=[M[1]], mode='markers',
                             marker=dict(size=12, color='red'), name='Master'))
    xa, ya = arrow_xy(M, H)
    fig.add_trace(go.Scatter(x=xa, y=ya, mode='lines',
                             line=dict(width=2, color='red'), showlegend=False))
    fig.add_trace(go.Scatter(x=[D1[0]], y=[D1[1]], mode='markers',
                             marker=dict(size=10, color='blue'), name='D1'))
    xb, yb = arrow_xy(D1, H)
    fig.add_trace(go.Scatter(x=xb, y=yb, mode='lines',
                             line=dict(width=2, color='blue'), showlegend=False))
    fig.add_trace(go.Scatter(x=[D2[0]], y=[D2[1]], mode='markers',
                             marker=dict(size=10, color='blue'), name='D2'))
    xc, yc = arrow_xy(D2, H)
    fig.add_trace(go.Scatter(x=xc, y=yc, mode='lines',
                             line=dict(width=2, color='blue'), showlegend=False))
    for k, c in [('M', 'red'), ('D1', 'blue'), ('D2', 'blue')]:
        xs = [p[0] for p in trails[k]]
        ys = [p[1] for p in trails[k]]
```

```
        fig.add_trace(go.Scatter(x=xs, y=ys, mode='lines',
                                 line=dict(color=c, dash='dot'),
                                 name=f'{k} trail'))
    pad = 50
    allx = sum([[p[0] for p in trails[k]] for k in trails], [M[0]])
    ally = sum([[p[1] for p in trails[k]] for k in trails], [M[1]])
    fig.update_layout(height=600, showlegend=True, title=title,
                      xaxis=dict(range=[min(allx) - pad, max(allx) + pad]),
                      yaxis=dict(range=[min(ally) - pad, max(ally) + pad]))
    return fig
app = dash.Dash(__name__)
server = app.server
state = dict(
    M=MASTER_POS.copy(),
    H=0.0,
    offs={k: v.copy() for k, v in OFF_INIT.items()},
    traj=[], idx=0,
    trails=dict(M=[], D1=[], D2=[])
)
app.layout = html.Div([
    html.H3("Wingman or Echelon Formation"),
    html.Div([
        html.Label("Size:"),
        dcc.RadioItems(id='sz',
                       options=[{'label': 'Hold',     'value': 'hold'},
```

```
                {'label': 'Expand',   'value': 'exp'},
                {'label': 'Contract', 'value': 'con'}],
            value='hold',
            inline=True,
            style={'margin-right': '16px'}),
        html.Label("Δ m"),
        dcc.Input(id='delta', type='number', value=100, min=1, max=10000,
                  style={'width': '70px'}),
        html.Label("Heading °"),
        dcc.Input(id='hdg', type='number', value=0, min=0, max=359,
                  style={'width': '60px'}),
        html.Label("Distance m"),
        dcc.Input(id='dist', type='number', value=0, min=0,
                  style={'width': '70px'}),

        html.Button("Command", id='cmd'),
        html.Button("Reset",   id='rst')
    ], style={'display': 'flex', 'gap': '8px', 'flex-wrap': 'wrap',
              'align-items': 'center', 'margin-bottom': '8px'}),
    html.Div(id='err', style={'color': 'red', 'margin-bottom': '4px'}),
    dcc.Graph(id='g'),
    dcc.Interval(id='t', interval=100, n_intervals=0, disabled=False)
])
@app.callback(
    Output('g', 'figure'),
```

```
    Output('t', 'disabled'),
    Output('err', 'children'),
    Input('t',   'n_intervals'),
    Input('cmd', 'n_clicks'),
    Input('rst', 'n_clicks'),
    State('sz',   'value'),
    State('delta','value'),
    State('hdg',  'value'),
    State('dist', 'value')
)
def ctrl(_, cmd, rst, sz, delta, hdg, dist):
    global state
    trig = dash.callback_context.triggered_id
    if trig == 'rst':
        state = dict(
            M=MASTER_POS.copy(),
            H=0.0,
            offs={k: v.copy() for k, v in OFF_INIT.items()},
            traj=[], idx=0,
            trails=dict(M=[], D1=[], D2=[])
        )
        M = state['M']
        D1 = M + state['offs']['D1']
        D2 = M + state['offs']['D2']
        state['trails'] = dict(M=[M], D1=[D1], D2=[D2])
```

```
    return fig_snap(M, D1, D2, 0, state['trails'], "Start"), True, ""
if trig == 'cmd':
    traj   = []
    trails = dict(M=[], D1=[], D2=[])
    M0     = state['M']
    H0     = state['H']
    off0   = state['offs']
    if sz != 'hold':
        try:
            d = float(delta)
        except Exception:
            d = -1
        if d <= 0:
            return dash.no_update, True, "Enter positive Δ."
        sign = 1 if sz == 'exp' else -1
        tgt = {}
        for k, v in off0.items():
            u   = v / np.linalg.norm(v)
            t   = v + sign * d * u
            r   = np.clip(np.linalg.norm(t), 10, 10000)
            tgt[k] = u * r
        for a in np.linspace(0, 1, 30):
            cur = {k: (1 - a) * off0[k] + a * tgt[k] for k in off0}
            D1  = M0 + cur['D1']
            D2  = M0 + cur['D2']
```

```
            traj.append((M0, D1, D2, H0))
            trails['M'].append(M0)
            trails['D1'].append(D1)
            trails['D2'].append(D2)
        state['offs'] = tgt
    else:
        hdg  = int(hdg) % 360
        dist = max(float(dist), 0.0)
        for H in np.linspace(H0, hdg, 20):
            D1 = M0 + off0['D1']
            D2 = M0 + off0['D2']
            traj.append((M0, D1, D2, H))
            trails['M'].append(M0)
            trails['D1'].append(D1)
            trails['D2'].append(D2)
        vec = np.array([np.cos(np.radians(hdg)),
                        np.sin(np.radians(hdg))])
        for d in np.linspace(0, dist, 30):
            M  = M0 + vec * d
            D1 = M + off0['D1']
            D2 = M + off0['D2']
            traj.append((M, D1, D2, hdg))
            trails['M'].append(M)
            trails['D1'].append(D1)
            trails['D2'].append(D2)
```

```
            state.update(M=M, H=hdg)
        state.update(traj=traj, idx=0, trails=trails)
        return dash.no_update, False, ""
    if trig == 't' and state['idx'] < len(state['traj']):
        M, D1, D2, H = state['traj'][state['idx']]
        state['idx'] += 1
        done = state['idx'] >= len(state['traj'])
        return fig_snap(M, D1, D2, H, state['trails'], ""), done, ""
    if not state['traj']:
        M  = state['M']
        D1 = M + state['offs']['D1']
        D2 = M + state['offs']['D2']
        state['trails'] = dict(M=[M], D1=[D1], D2=[D2])
        return fig_snap(M, D1, D2, 0, state['trails'], "Start"), True, ""
    return dash.no_update, True, ""
if __name__ == "__main__":
    app.run(debug=True)
```

Code – 07: Collision Avoidance Logic in Simulation

Drone.py

```
import numpy as np
from utils import heading_to_vector, distance, normalize_angle
class Drone:
    def __init__(self, drone_id, init_pos, heading_deg, speed):
        self.id = drone_id
        self.position = np.array(init_pos, dtype='float64')
        self.heading = heading_deg
        self.speed = speed
        self.velocity = heading_to_vector(self.heading) * self.speed
        self.history = [self.position.copy()]
        self.received_states = {}
        self.avoiding = False
    def update(self, dt):
        self.predict_collision_and_avoid()
        self.position += self.velocity * dt
        self.history.append(self.position.copy())
    def broadcast_state(self):
        return {
            'id': self.id,
            'pos': self.position.copy(),
            'velocity': self.velocity.copy(),
            'heading': self.heading
        }
```

```
    def receive_states(self, state_list):
        for state in state_list:
            if state['id'] != self.id:
                self.received_states[state['id']] = state
    def predict_collision_and_avoid(self):
        self.avoiding = False
        future_time = 1  # seconds
        safety_radius = 1  # meters
        my_future_pos = self.position + self.velocity * future_time
        for state in self.received_states.values():
            other_future_pos = state['pos'] + state['velocity'] * future_time
            if distance(my_future_pos, other_future_pos) < safety_radius:
                self.avoid()
                break
    def avoid(self):
        self.heading += 15
        self.heading = normalize_angle(self.heading)
        self.velocity = heading_to_vector(self.heading) * self.speed
        self.avoiding = True
```

utils.py

```
import numpy as np
def heading_to_vector(heading_deg):
    rad = np.radians(heading_deg)
    return np.array([np.cos(rad), np.sin(rad)])
```

```
def distance(a, b):
    return np.linalg.norm(np.array(a) - np.array(b))
def normalize_angle(angle):
    return angle % 360
```

main.py

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from drone import Drone
drones = [
    Drone(drone_id=1, init_pos=(0, 0), heading_deg=0, speed=1),
    Drone(drone_id=2, init_pos=(5, 5), heading_deg=180, speed=1),
    Drone(drone_id=3, init_pos=(10, 0), heading_deg=180, speed=1)
]
dt = 0.1
def update(frame):
    all_states = [d.broadcast_state() for d in drones]
    for d in drones:
        d.receive_states(all_states)
        d.update(dt)
    ax.clear()
    ax.set_xlim(-10, 20)
    ax.set_ylim(-10, 20)
    for d in drones:
        color = 'red' if d.avoiding else 'blue'
        ax.plot(*d.position, 'o', color=color)
```

```
        xs, ys = zip(*d.history[-50:])
        ax.plot(xs, ys, '--', alpha=0.5)
fig, ax = plt.subplots()
ani = animation.FuncAnimation(fig, update, interval=100)
plt.show()
```