INTERNSHIP REPORT
IIT GUWAHATI

Department: Electrical and Electronics Engineering

Objectives: Autonomous Drones

Ariyan Nath (2nd year)
Chandan Chetia (2nd year)
MD Sohail Ali (2nd year)

INTRODUCTION

Unmanned Aerial Vehicles (UAVs), commonly known as drones, have rapidly evolved from remote-controlled platforms into intelligent autonomous systems capable of performing complex tasks with minimal human intervention. Advances in embedded systems, flight controllers, sensors, and computer vision have enabled drones to be widely used in applications such as surveillance, mapping, agriculture, disaster management, and autonomous navigation.

This documentation presents the work carried out during a research internship at the Indian Institute of Technology (IIT) Guwahati, focused on the design, configuration, and development of autonomous drone systems. The primary objective of the internship was to gain practical, hands-on experience in assembling drone hardware, configuring professional-grade flight controllers, and implementing vision-based autonomy using onboard computing platforms.

The internship involved the assembly, wiring, and calibration of multi-rotor drones, followed by the configuration of Pixhawk and Cube Orange flight controllers using Mission Planner software. Emphasis was placed on understanding telemetry-based communication, sensor integration, and testing of various flight modes such as Loiter, Return-to-Launch (RTL), Auto-tune, and Stabilise. These activities helped in understanding the fundamentals of flight stability, control algorithms, and autonomous navigation.

In addition to flight control and hardware integration, the internship explored computer vision and real-time data processing using a Raspberry Pi 4 as an onboard companion computer. Live video streaming was implemented using MJPG-Streamer, and object detection was performed using the YOLO model in Python 3. This enabled the development of a vision-based drone-following-drone system, where one drone autonomously tracks and follows another.

Communication between the Raspberry Pi and the Cube Orange flight controller was established using GPIO-based serial communication (TX, RX, and GND) via the TELEM2 port, utilizing the MAVLink protocol. This integration allowed real-time exchange of navigation and control data, forming the backbone of autonomous behavior in the system.

Overall, this internship provided comprehensive exposure to UAV hardware integration, flight controller configuration, telemetry communication, and vision-based autonomy, bridging the gap between theoretical concepts and real-world drone system implementation. The documentation aims to systematically describe the methodology, challenges faced, solutions implemented, and outcomes achieved during the internship.

HARDWARE SETUP

Overview

The hardware setup forms the foundation of the autonomous drone system developed during this internship. It consists of a multi-rotor aerial platform integrated with professional-grade flight controllers, communication modules, etc. The objective of the hardware design was to ensure structural stability, reliable flight control, efficient sensor integration, and seamless communication between the vision-processing unit and the flight controller.

The overall system architecture follows a modular design approach, enabling ease of assembly, debugging, calibration, and future scalability.

Hardware Components Used

The major hardware components used in the system are listed below:

1. Drone Frame

A multi-rotor drone frame was used as the structural base to mount all electronic components, motors, and sensors. The frame provided mechanical stability and proper weight distribution for safe flight operations.

2. Brushless DC (BLDC) Motors and Propellers

BLDC motors were used to generate lift and thrust, paired with appropriately sized propellers to ensure efficient propulsion and stable flight dynamics.

3. Electronic Speed Controllers (ESCs)

ESCs were used to regulate motor speed based on control signals received from the flight controller. Each motor was connected to an individual ESC for precise thrust control.

4. Power Distribution Board (PDB) and Battery

A Li-Po battery was used as the primary power source. The Power Distribution Board distributed power uniformly to the ESCs, flight controller, and onboard electronics while maintaining voltage stability.

5. Flight Controllers

- Pixhawk Flight Controller
- Cube Orange Flight Controller

These controllers acted as the central processing units of the drone, handling sensor fusion, flight stabilisation, navigation, and control algorithms.

6. Telemetry Module

Telemetry radios were used for real-time communication between the drone and the Ground Control Station (GCS), enabling live monitoring and parameter tuning.

7. Raspberry Pi 4 (Companion Computer)

A Raspberry Pi 4 was used as an onboard companion computer for real-time video processing, object detection, and autonomous decision-making.

8. Camera Module

A webcam was connected to the Raspberry Pi to capture live video feed for object detection and tracking.

9. Communication Interfaces and Wiring

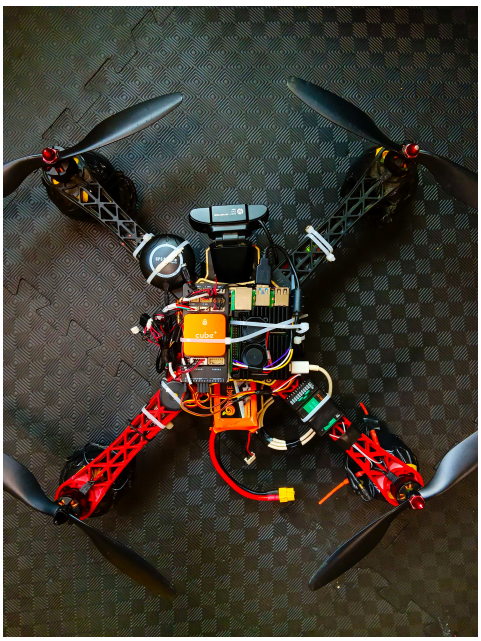
- GPIO pins of Raspberry Pi
- TX, RX, and GND connections
- TELEM2 port of Cube Orange

These interfaces enabled serial communication between the companion computer and the flight controller.

Hardware Assembly and Integration

The assembly process began with mounting the motors and ESCs onto the drone frame. The ESCs were connected to the motors and routed to the Power Distribution Board for power supply. Care was taken to ensure proper insulation and secure wiring to prevent electrical interference and mechanical damage.

The flight controller (Pixhawk / Cube Orange) was mounted at the centre of the frame using vibration-damping mounts to minimise sensor noise. Orientation alignment was maintained according to manufacturer specifications to ensure accurate attitude estimation.



Next, the battery and power module were integrated, supplying regulated power to the flight controller and peripheral components. All power connections were verified using a multimeter before powering the system.

The telemetry module was connected to the flight controller to enable bidirectional communication with the Ground Control Station. This allowed real-time monitoring of flight parameters through Mission Planner software.

https://youtu.be/9DsqzNu-zIA?si=UbUWp52ZNUwlT75l

For vision-based autonomy, the Raspberry Pi 4 was mounted securely on the drone frame. A webcam was attached and aligned to provide a clear forward-facing field of view. The Raspberry Pi was powered using a regulated power bank compatible with its voltage requirements.

Serial communication between the Raspberry Pi and the Cube Orange was established using the GPIO pins of the Raspberry Pi, connected to the TX, RX, and GND pins of the TELEM2 port on the Cube Orange flight controller. This connection

enabled MAVLink-based communication, allowing the companion computer to send navigation commands and receive telemetry data in real time.

Hardware Configuration and Validation

After assembly, all hardware components were validated through systematic testing. Connectivity checks, power integrity tests, and sensor detection were performed using Mission Planner. Motor direction, ESC calibration, and failsafe mechanisms were verified before proceeding to flight tests.

The hardware setup was designed to support both manual and autonomous flight modes, providing a reliable platform for experimentation with vision-based tracking and drone-to-drone coordination.

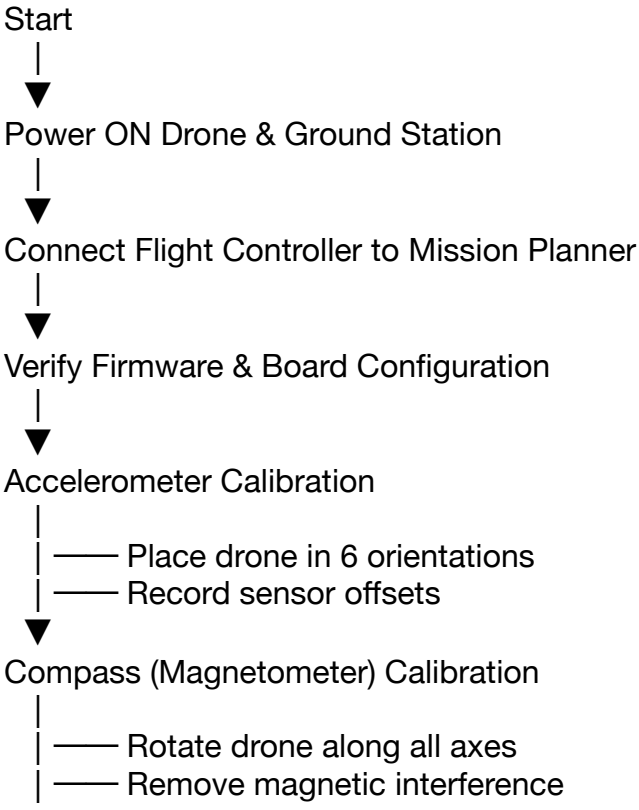
Drone Calibration

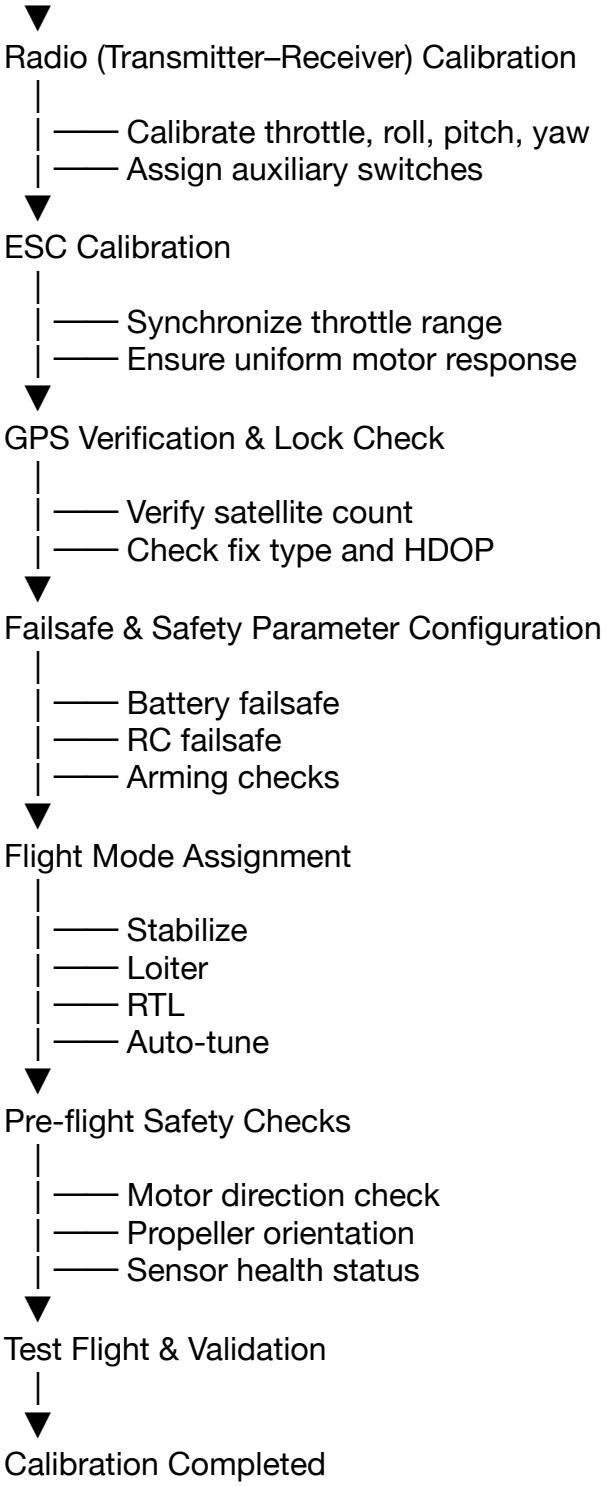
All calibration procedures were performed using Mission Planner, which served as the Ground Control Station (GCS) software. Mission Planner provides tools for configuring flight controller parameters, monitoring telemetry data, and performing systematic calibration of sensors and radio systems.

The drone was connected to Mission Planner via a telemetry link or USB connection during the calibration process.

Successful flight testing confirmed that the drone was properly calibrated and ready for advanced autonomous experiments.

https://youtu.be/HwY4WCNErV8?si=WRqdfoBVoMu2PbUO



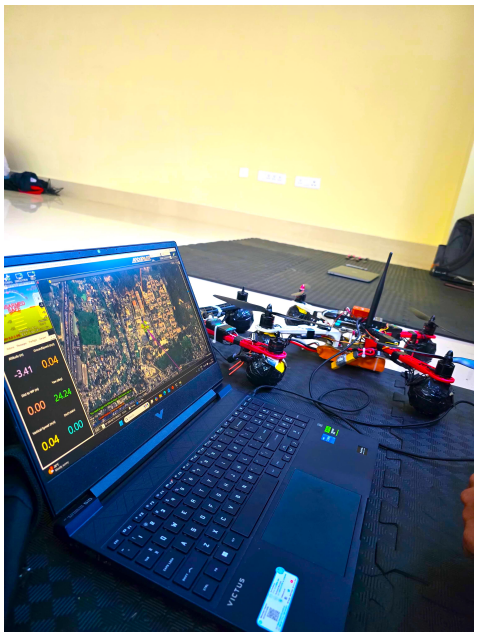

Radio (Transmitter–Receiver) Calibration
Calibrate throttle, roll, pitch, yaw
Assign auxiliary switches
ESC Calibration
Synchronize throttle range
Ensure uniform motor response
GPS Verification & Lock Check
Verify satellite count
Check fix type and HDOP
Failsafe & Safety Parameter Configuration
Battery failsafe
RC failsafe
Arming checks
Flight Mode Assignment
Stabilize
Loiter
RTL
Auto-tune
Pre-flight Safety Checks
Motor direction check
Propeller orientation
Sensor health status
Test Flight & Validation
Calibration Completed

PROBLEMS FACED AND CHALLENGES ENCOUNTERED

During the course of the research internship, several technical and operational challenges were encountered while working with the drone platform, flight controllers, and onboard computing systems. These challenges played a crucial role in enhancing practical understanding, troubleshooting skills, and system-level thinking. The major problems faced are discussed below.

Challenges Related to Drone Hardware and Flight Controller

One of the primary difficulties encountered was the lack of comprehensive tutorial resources, particularly for the Cube Orange flight controller. While limited documentation and videos were available, most resources did not explain small but critical configuration parameters. This made the initial understanding of certain flight controller settings challenging and required extensive experimentation and reference to official documentation.



Another major issue was related to Mission Planner connectivity. At times, Mission Planner failed to establish a stable connection with the flight controller. In some instances, the software lagged or stopped responding during calibration procedures, which disrupted the setup process and required repeated reconnections and restarts.

During radio calibration, the receiver failed to respond correctly to input signals. This issue was traced back to improper binding between the transmitter and receiver. Re-binding the receiver resolved the issue and allowed the calibration process to be completed successfully.

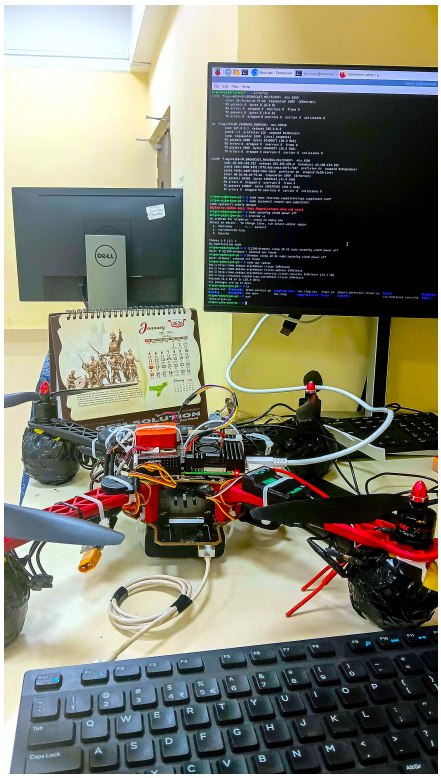
Several challenges were also faced during ESC calibration, where improper motor response and synchronisation issues were observed. In addition, GPS detection failures, sensor health warnings, and parameter-related errors were frequently encountered. These issues required repeated verification of wiring, recalibration of sensors, and careful adjustment of flight controller parameters to ensure stable and safe operation.

Challenges Related to Raspberry Pi and Remote Access

While working with the Raspberry Pi 4 as the companion computer, multiple challenges were encountered, especially during the initial setup phase. Initially, remote access to the Raspberry Pi was not possible due to lack of familiarity with remote connectivity methods.

This issue was resolved by learning and implementing Secure Shell (SSH) and Raspberry Pi Connect, which enabled remote access to the Raspberry Pi from a laptop terminal. Access was achieved by identifying the IP address of the Raspberry Pi using the command:

```
ping hostname.local
```



It was also observed that both the Raspberry Pi and the laptop must be connected to the same network in order to establish a successful SSH connection.

Another recurring issue was related to network connectivity. In certain instances, when the Raspberry Pi was powered on using a USB-C cable, it failed to remember the Wi-Fi network credentials that were assigned during the operating system installation. As a result, the Raspberry Pi was unable to connect to the internet, which prevented SSH access.

Initially, the operating system was reinstalled multiple times to reconfigure the network settings; however, this approach was inefficient and time-consuming. After discussing the issue with the mentor, a more effective solution was identified. By removing the password from the target Wi-Fi network and configuring it as an open network, the Raspberry Pi was able to connect reliably without authentication issues. This approach significantly improved accessibility and reduced setup time.

LEADER FOLLOWER SETUP

To enable one drone to autonomously follow another, a Leader–Follower architecture was implemented. In this setup, two drones operate simultaneously with distinct roles:

- Leader Drone: Operated manually by a pilot
- Follower Drone: Operates autonomously by tracking the leader using vision processing and flight controller commands

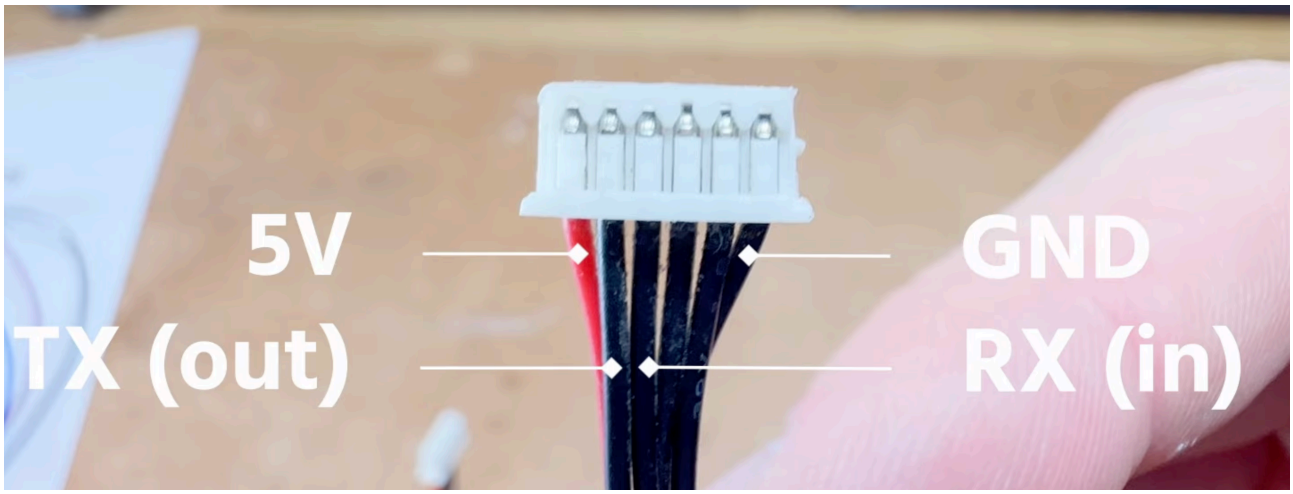
The leader drone provides a dynamic target, while the follower drone processes visual data and navigation information to maintain relative positioning. This approach enables real-time autonomous following without direct human control of the follower drone.

Physical Connection Between Cube Orange and Raspberry Pi

To achieve autonomous control and real-time communication, the Cube Orange flight controller was physically interfaced with a Raspberry Pi 4, which acted as the companion computer responsible for vision processing and decision-making.

https://www.youtube.com/watch?v=nluoCYauW3s

Communication between the Cube Orange and Raspberry Pi was established using the TELEM2 port of the Cube Orange flight controller and the GPIO pins of the Raspberry Pi.



Wiring Methodology

The connection was implemented as follows:

- The TELEM2 cable of the Cube Orange was cut and prepared for custom wiring
- Three female jumper wires were soldered to the corresponding TELEM2 lines
- Only the required serial communication lines were used (TX, RX and GND)

Pin Connections

Cube Orange (TELEM2)	Raspberry Pi GPIO
GND	GPIO Pin 9
TX	GPIO Pin 10
RX	GPIO Pin 8

This configuration enabled UART-based serial communication between the flight controller and the companion computer, which is essential for MAVLink message exchange.

Software Environment and Dependency Installation

After establishing the physical connection, the required software dependencies were installed on the Raspberry Pi to enable communication with the Cube Orange flight controller.All operations were performed by accessing the Raspberry Pi remotely via SSH.

System Update and Python Environment Setup

```
sudo apt-get update
sudo apt-get install python3-pip
sudo apt-get install python3-dev python3-opencv python3-wxgtk4.0 python3-matplotlib python3-lxml
libxml2-dev libxslt-dev
sudo apt update
sudo apt install python3-venv python3-full
```

A Python virtual environment was created to isolate MAVLink-related dependencies:

```
python3 -m venv ~/mavproxy_venv
source ~/mavproxy_venv/bin/activate
```

The package manager was upgraded and MAVProxy was installed:

```
pip install --upgrade pip
pip install mavproxy
```

MAVLink Communication Verification

After completing the hardware and software setup, the communication link between the Raspberry Pi and the Cube Orange flight controller was verified using MAVProxy.

The following command was used to establish the MAVLink connection:

mavproxy.py --master=/dev/serial0 --baudrate 57600 —out=udp:127.0.0.1:14550

Successful execution of this command confirmed:
- Proper UART communication over TELEM2
- Correct baud rate configuration
- Reliable MAVLink message exchange

This step validated that the Raspberry Pi could receive telemetry data and send control commands to the Cube Orange flight controller, forming the backbone of the autonomous leader–follower drone system.

Preliminary Object Detection Using YOLO

Before implementing the complete Leader–Follower drone system, it was necessary to validate the object detection pipeline independently. This step ensured that the onboard companion computer (Raspberry Pi 4) was capable of performing real-time object detection reliably before integrating it with the flight controller for autonomous control.

The objective of this phase was to implement basic object detection using a YOLO-based model in Python 3, verify camera input, and ensure proper software dependency management within a controlled virtual environment.

Software Environment Preparation

All required dependencies were installed inside the previously created Python virtual environment to avoid conflicts with system-wide packages. The virtual environment was activated before executing the following commands.

To ensure compatibility between MAVProxy, pymavlink, and other Python libraries, existing installations were first removed and reinstalled in a controlled manner.

```
pip uninstall mavproxy pymavlink -y
pip install --upgrade pip setuptools wheel
pip install mavproxy --no-cache-dir
pip uninstall mavproxy pymavlink -y
pip install git+https://github.com/ArduPilot/MAVProxy.git
```

This ensured the latest stable version of MAVProxy compatible with the ArduPilot ecosystem.

Installation of Object Detection Dependencies

The following system-level and Python-level libraries were installed to support computer vision and object detection.

sudo apt update
sudo apt install -y python3-opencv python3-pip python3-numpy python3-requests libopencv-dev

These packages provide OpenCV bindings, numerical computation support, and HTTP utilities required for model handling.Downloading Pre-trained Object Detection Models.

To perform object detection, a pre-trained MobileNet-SSD model trained on the COCO dataset was used. The required configuration files, class labels, and model weights were downloaded as follows:

wget https://raw.githubusercontent.com/opencv/opencv/master/samples/dnn/
object_detection_classes_coco.txt -O classes.txt
wget https://github.com/opencv/opencv/raw/master/samples/dnn/ssd_mobilenet_object_detection.dnn.cfg
-O ssd_mobilenet_coco.cfg
wget http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v1_coco_2017_11_17.tar.gz

The downloaded model archive was extracted and organised:

tar -xvf ssd_mobilenet_v1_coco_2017_11_17.tar.gz
mv ssd_mobilenet_v1_coco_2017_11_17/frozen_inference_graph.pb model.pb
rm ssd_mobilenet_v1_coco_2017_11_17.tar.gz
rm -rf ssd_mobilenet_v1_coco_2017_11_17

This resulted in a clean directory structure containing:

- Class label file (classes.txt)
- Model configuration file (ssd_mobilenet_coco.cfg)
- Pre-trained inference graph (model.pb)

Installation of YOLO Framework

In addition to the MobileNet-SSD model, the Ultralytics YOLO framework was installed to support YOLO-based object detection using Python 3.

python3 -m pip install ultralytics

This framework enables efficient object detection and provides flexibility for experimentation with different YOLO variants.

Creation of Object Detection Script

After completing all dependency installations, a Python script was created to implement object detection and video streaming.

A new Python file was created using the Nano text editor:

nano object_detection_stream.py

After the script editor opens, paste the python script given below:

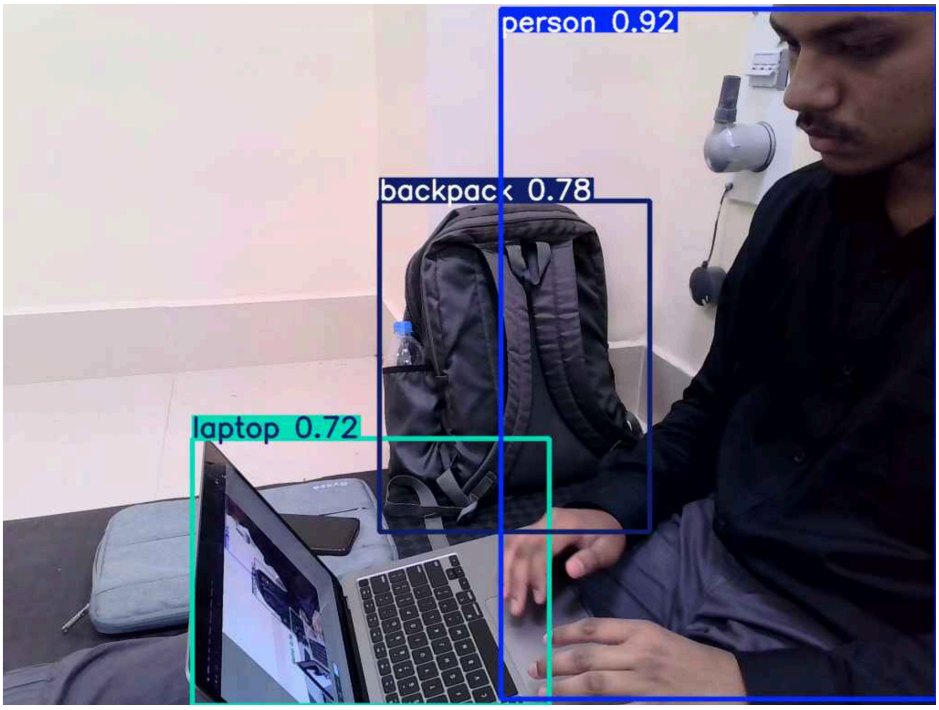
```
import cv2
from ultralytics import YOLO
import socketserver
import http.server
import time
import sys

# ——— Configuration

PORT = 8000
WIDTH, HEIGHT = 640, 480
YOLO_INPUT_SIZE = 256
JPEG_QUALITY = 35                    # 40-60 range recommended for speed
DETECTION_INTERVAL = 0.30            # seconds between detections (≈5-6 fps detection)
CONFIDENCE_THRESHOLD = 0.50

# ——— Load model

print("Loading YOLOv8n model...")
model = YOLO("yolov8n.pt")
print("Model loaded.")

# ——— Open webcam

print("Opening webcam...")
cap = cv2.VideoCapture(0, cv2.CAP_V4L2)

if not cap.isOpened():
    print("ERROR: Cannot open webcam /dev/video0")
    print("Try changing to cv2.VideoCapture(1) or (2) etc.")
    sys.exit(1)

cap.set(cv2.CAP_PROP_FRAME_WIDTH, WIDTH)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, HEIGHT)
cap.set(cv2.CAP_PROP_FPS, 15)
cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)          # minimize internal queue
cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc('M','J','P','G'))
```

```
actual_w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
actual_h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
print(f"Camera opened at {actual_w}x{actual_h}")

# ── HTTP Handler

class LowLagMJPEGHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path != '/stream.mjpg':
            self.send_error(404)
            self.end_headers()
            return

        self.send_response(200)
        self.send_header("Content-type", "multipart/x-mixed-replace; boundary=frame")
        self.send_header("Cache-Control", "no-cache")
        self.end_headers()

        last_detection_time = 0
        last_annotated = None

        try:
            while True:
                # ── Grab & drop old frames aggressively ──
                for _ in range(2):          # adjust 1-4 depending on lag
                    cap.grab()

                ret, frame = cap.retrieve()
                if not ret:
                    time.sleep(0.02)
                    continue

                now = time.time()

                # Run detection only at desired interval
                if now - last_detection_time >= DETECTION_INTERVAL:
                    results = model(frame,
                                    imgsz=YOLO_INPUT_SIZE,
                                    conf=CONFIDENCE_THRESHOLD,
                                    verbose=False)
                    annotated = results[0].plot() if results[0].boxes else frame.copy()
                    last_annotated = annotated
```

```
                last_detection_time = now
            else:
                # Reuse last annotated frame (cheap)
                annotated = last_annotated if last_annotated is not None else frame

            # Encode fast
            success, jpeg = cv2.imencode('.jpg', annotated,
                                         [cv2.IMWRITE_JPEG_QUALITY, JPEG_QUALITY])
            if not success:
                continue

            data = jpeg.tobytes()

            try:
                self.wfile.write(b"--frame\r\n")
                self.send_header("Content-type", "image/jpeg")
                self.send_header("Content-length", len(data))
                self.end_headers()
                self.wfile.write(data)
                self.wfile.write(b"\r\n")
            except (BrokenPipeError, ConnectionResetError):
                print("Client disconnected")
                break

        except Exception as e:
            print(f"Stream error: {e}")
        finally:
            print("Connection closed")

# ——— Threaded Server

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    allow_reuse_address = True
    daemon_threads = True

# ——— Main

if __name__ == "__main__":
    server_address = ('', PORT)
    httpd = ThreadedTCPServer(server_address, LowLagMJPEGHandler)

    print(f"\nLow-lag YOLOv8 MJPEG stream running on port {PORT}")
```

```
print(f"Open in browser or VLC:  http://<your-pi-ip>:{PORT}/stream.mjpg")
print("Press Ctrl+C to stop\n")

try:
    httpd.serve_forever()
except KeyboardInterrupt:
    print("\nShutting down...")
    cap.release()
    httpd.server_close()
    print("Done.")
```

Run it by just giving the command: *python3 object_detection_stream.py*
Then paste this on your web browser: *http://<your-IP-address>:8000/stream.mjpg*



You must see a result like this.
*(The webcam must be connected to the raspberry pi via the USB A port)

Autonomous Drone Following Implementation

After validating the object detection pipeline using YOLO, the next stage involved implementing an autonomous drone-following system, where a follower drone tracks and follows a manually piloted leader drone. This capability forms the central functionality of the Leader–Follower architecture, enabling vision-based autonomous navigation.

The objective of this stage was to integrate computer vision, MAVLink communication, and flight control logic to allow the follower drone to respond dynamically to the position of the leader drone in real time.

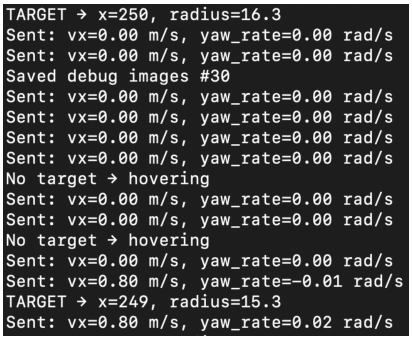
Software Dependencies for Drone Following

To enable real-time vision processing and MAVLink-based communication with the flight controller, additional software dependencies were installed on the Raspberry Pi. All commands were executed after ensuring the system was updated.

sudo apt update
sudo apt upgrade -y
sudo apt install python3-pip python3-opencv libatlas-base-dev -y

The following Python libraries were installed to support MAVLink communication and image processing:

pip3 install pymavlink opencv-contrib-python imutils



Creation of Drone Following Script

After successful installation of all required dependencies, a separate Python script was created to implement the drone-following logic.

The script was created using the Nano text editor:

nano drone_pursuit.py

After the script editor opens, paste the python script given below:

```
import cv2
import imutils
import time
from pymavlink import mavutil

# ==================== CONFIG ====================
BAUD              = 57600
MIN_VX            = 0.15
MAX_VX            = 0.50
MIN_FOLLOW_DIST   = 6.0
TARGET_REAL_WIDTH  = 0.40         # â MEASURE your target drone width in meters
YAW_GAIN          = 2.2
VZ_CLIMB          = 0.60          # m/s upward
VZ_HOVER          = 0.00
VZ_BACKOFF        = -0.25
```

```
TAKEOFF_ALT       = 3.5          # meters

print("Drone Pursuit - Auto-Takeoff Version Starting...")

# ==================== MAVLink ====================
conn = mavutil.mavlink_connection('/dev/serial0', baud=BAUD, dialect='ardupilotmega')
print("Waiting for heartbeat...")
conn.wait_heartbeat(timeout=15)
print(f"Connected â sys {conn.target_system} comp {conn.target_component}")
conn.target_component = 1

# ==================== Arm ====================
def arm_drone():
    print("Arming...")
    conn.mav.command_long_send(conn.target_system, conn.target_component,
        mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM, 0, 1, 0, 0, 0, 0, 0, 0)
    time.sleep(4)
    print("Arm sent")

# ==================== Set GUIDED_NOGPS ====================
def set_guided_nogps():
    print("Setting GUIDED_NOGPS...")
    for i in range(8):
        conn.mav.set_mode_send(conn.target_system,
            mavutil.mavlink.MAV_MODE_FLAG_CUSTOM_MODE_ENABLED, 20)
        time.sleep(3)
        msg = conn.recv_match(type='HEARTBEAT', blocking=True, timeout=3)
        if msg and msg.custom_mode == 20:
            print("â GUIDED_NOGPS active")
            return True
    print("GUIDED_NOGPS not confirmed")
    return False

# ==================== Send velocity ====================
def send_velocity(vx=0.0, vy=0.0, vz=0.0, yaw_rate=0.0):
    vx = max(vx, MIN_VX)
    if abs(yaw_rate) < 0.03:
        yaw_rate = 0.02 if yaw_rate >= 0 else -0.02

    conn.mav.set_position_target_local_ned_send(
        0, conn.target_system, conn.target_component,
        mavutil.mavlink.MAV_FRAME_BODY_OFFSET_NED,
        0b0000111111000111,
        0,0,0, vx,vy,vz, 0,0,0, 0,yaw_rate
    )
    print(f"â vx={vx:.2f} vz={vz:.2f} yaw={yaw_rate:.3f}")

# ==================== Webcam ====================
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
```

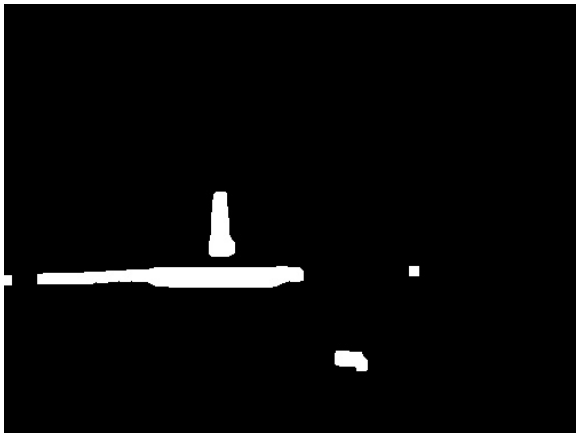
```
# VALID HSV (red example)
lower_color = (0, 100, 100)
upper_color = (10, 255, 255)

frame_count = 0
takeoff_sent = False
last_vx = MIN_VX
last_vz = 0.0
last_yaw_rate = 0.0

# ==================== MAIN ====================
arm_drone()
set_guided_nogps()

print("\nWaiting for target detection â will auto-takeoff to 3.5 m")
print("Debug images every 12 frames")

try:
    while True:
        ret, frame = cap.read()
        if not ret: continue

        frame_count += 1
        resized = imutils.resize(frame, width=500)
        blurred = cv2.GaussianBlur(resized, (11,11), 0)
        hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

        mask = cv2.inRange(hsv, lower_color, upper_color)
        mask = cv2.erode(mask, None, 2)
        mask = cv2.dilate(mask, None, 4)

        if frame_count % 12 == 0:
            cv2.imwrite(f"debug_frame_{frame_count}.jpg", resized)
            cv2.imwrite(f"debug_mask_{frame_count}.jpg", mask)

        contours = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[-2]

        if contours:
            c = max(contours, key=cv2.contourArea)
            ((x, y), radius) = cv2.minEnclosingCircle(c)

            if radius > 15:
                center_x = int(x)
                frame_center = resized.shape[1] // 2
                yaw_error = (center_x - frame_center) / frame_center
                yaw_rate = yaw_error * YAW_GAIN

                distance_m = TARGET_REAL_WIDTH / (radius / resized.shape[1])

                # === AUTO TAKEOFF ON FIRST DETECTION ===
                if not takeoff_sent:
```

```
                print(f"TARGET DETECTED! Sending TAKEOFF to {TAKEOFF_ALT} m...")
                conn.mav.command_long_send(
                    conn.target_system, conn.target_component,
                    mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0,
                    0, 0, 0, 0, 0, 0, TAKEOFF_ALT
                )
                takeoff_sent = True
                time.sleep(12)  # wait for takeoff
                print("Takeoff command sent - waiting for altitude...")

            # Normal chase logic
            if distance_m > MIN_FOLLOW_DIST + 3:
                vx = 0.45
                vz = VZ_CLIMB
            elif distance_m > MIN_FOLLOW_DIST:
                vx = 0.30
                vz = VZ_HOVER
            else:
                vx = 0.0
                vz = VZ_BACKOFF if distance_m < MIN_FOLLOW_DIST - 1 else 0.0

            vx = min(vx, MAX_VX)
            vz = max(vz, -0.5)

            send_velocity(vx, 0, vz, yaw_rate)
            last_vx, last_vz, last_yaw_rate = vx, vz, yaw_rate

        else:
            send_velocity(last_vx * 0.7, 0, 0.30, last_yaw_rate * 0.6)

    else:
        send_velocity(MIN_VX, 0, 0.25, 0.02)

    conn.mav.heartbeat_send(mavutil.mavlink.MAV_TYPE_GCS,
                            mavutil.mavlink.MAV_AUTOPILOT_INVALID, 0, 0, 0)

    time.sleep(0.065)

except KeyboardInterrupt:
    print("\nStopped by user")
finally:
    send_velocity(0, 0, 0, 0)
    time.sleep(1)
    conn.mav.command_long_send(conn.target_system, conn.target_component,
        mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM, 0, 0, 0, 0, 0, 0, 0, 0)
    cap.release()
    print("Done.")
```

Run it by just giving the command: *python3 drone_pursuit.py*

*(Make sure that the raspberry pi is physically connected to the flight controller via the TELEM2 port)

RESULTS



These are some images that the webcam captured when a drone was detected.

Video Gallery:

https://drive.google.com/drive/folders/1old65274a7uVKUZfbL-ukDCEAQpRBQ4R?usp=sharing

Photo Gallery:

https://drive.google.com/drive/folders/1HSGzTEuW3mnSMV13_brZm7Uf3fvJ2qJY?usp=sharing

CONCLUSION

This project successfully demonstrated the design, configuration, and implementation of an autonomous leader–follower drone system through a combination of UAV hardware integration, flight controller configuration, telemetry communication, and computer vision techniques. The primary objective of gaining hands-on experience with professional drone platforms and autonomous flight systems was effectively achieved during the course of this research internship.

The work began with the systematic assembly and calibration of a multi-rotor drone, followed by the configuration of Pixhawk and Cube Orange flight controllers using Mission Planner. Proper calibration of sensors, radio systems, ESCs, and GPS modules ensured stable and reliable flight performance, forming a strong foundation for autonomous operations. Various flight modes, including Stabilise, Loiter, Return-to-Launch (RTL), and Auto-tune, were tested and validated.

A significant contribution of this project was the integration of a Raspberry Pi 4 as a companion computer to enable onboard computation and real-time perception. Object detection was implemented using the YOLO object detection algorithm in Python 3, with live video streaming achieved through MJPG-Streamer. This vision system enabled the detection and tracking of a target drone in real time.

The successful establishment of MAVLink-based communication between the Raspberry Pi and the Cube Orange flight controller via the TELEM2 interface enabled seamless data exchange and control. This integration formed the backbone of the autonomous drone-following-drone (leader–follower) system, where the follower drone autonomously tracked and followed a manually piloted leader drone based on visual feedback.

Throughout the project, several technical challenges related to hardware setup, software configuration, connectivity, and calibration were encountered. Addressing these challenges significantly enhanced troubleshooting skills, practical understanding of UAV systems, and system-level problem-solving abilities. The project effectively bridged theoretical concepts in robotics and computer vision with real-world UAV implementation.

In conclusion, this work demonstrates the feasibility of implementing vision-based autonomous drone following using low-cost embedded computing platforms and open-source flight control software. The outcomes of this project provide a strong foundation for future research in multi-drone coordination, swarm robotics, and advanced autonomous navigation, and contribute valuable practical insights into the development of intelligent UAV systems.

I would like to express my sincere gratitude to **Professor Chayan Bhawal** for his valuable guidance and support throughout the internship. I am also thankful to our supervisor, **PhD student of IIT G Sir Paraj**, for his constant encouragement and technical guidance. Finally, I would like to thank **IIT Guwahati** for providing the opportunity and resources to carry out this internship successfully.