DRONA AVIATION: Pluto Drone Swarm Challenge

Final Report (Team 41)

1 Introduction

Drona Aviation's Pluto Drone Swarm Challenge requires the participants to develop a python wrapper, which enables end users to communicate with and execute autonomous missions on Pluto 1.2. This report presents our technical approach to developing a python based wrapper to achieve the following tasks:

1. Enable the user to fly the drone using keyboard commands.
2. Enable the user to execute autonomous missions by passing a set of waypoints as inputs.
3. Establish a collaborative pipeline to communicate and control multiple drones autonomously.

2 Technical Approach

The approach presented in this report is broadly divided into three sections: **Communication**, **Computer Vision (CV)** and **Controls**. The communication module must transmit data over the radio to the drone's flight controller and receive state feedback from the onboard sensors using the **telnet** protocol. The computer vision module must estimate the drone's pose using **ArUco** markers and send this data to the controller. The control module receives the sensor feedback and the estimated pose from the vision module and uses a **PID** control law to send radio commands to the drone to execute the mission.

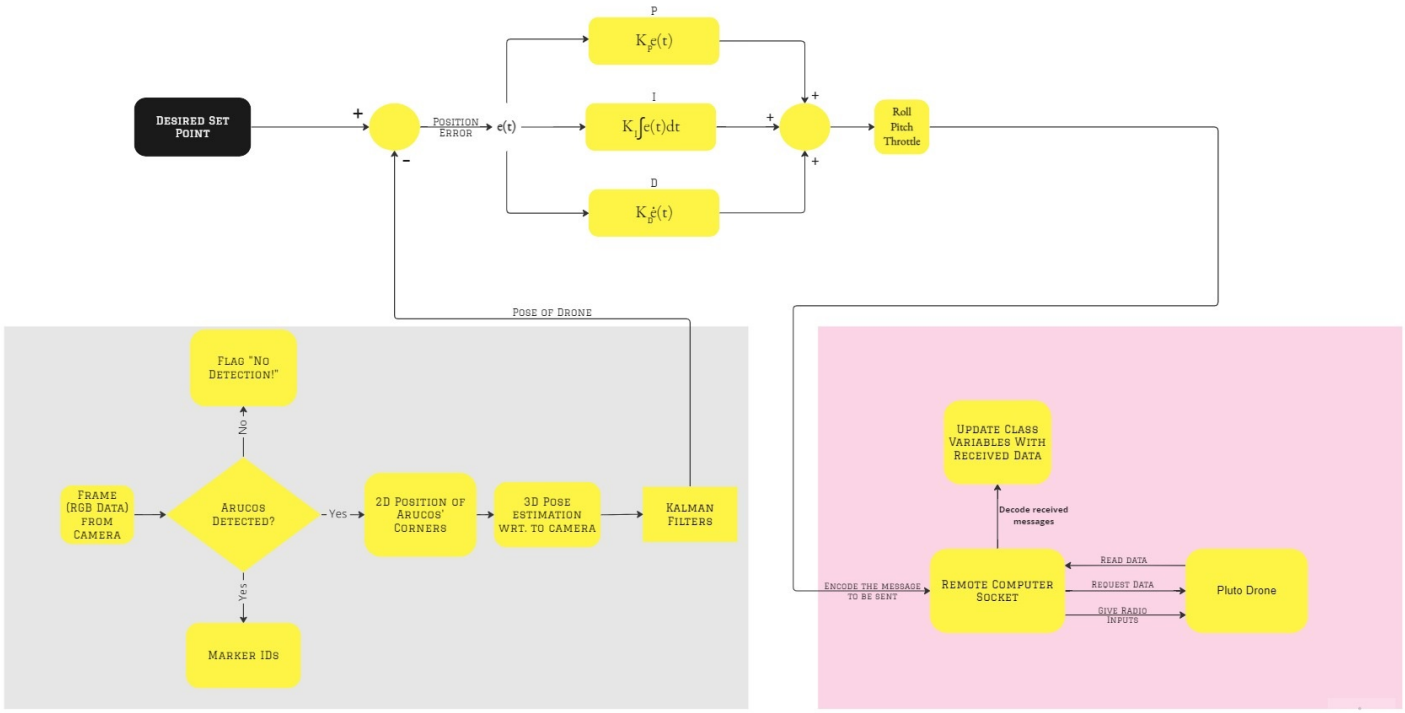

Figure 1: Overall Architecture

2.1 Communication

2.1.1 Objective

Multiple modules are needed to establish a communication pipeline between the drone and the remote computer over Wi-Fi to manoeuvre the Pluto drone autonomously. The first challenge is transmitting radio commands to fly the drone. The next challenge is to receive data from the drone as feedback to compare its current state with the desired state in the controller. This two-way communication protocol needs to run concurrently to avoid delay between the command and execution of an action. The sections below comprehensively describe the procedure to achieve this objective.

2.1.2 Communication Protocol (Telnet)

To establish robust communication with the drone, telnet protocol was used. The **telnet** library makes use of **sockets** as network endpoints to transmit and receive data over Wi-Fi. We can directly write or read information over the sockets using this library. The drone's IP address and the port number over which the communication is supposed to be established are fed as the input to the telnet library to allow for the reading and writing of sockets. Multiple sockets can be initialized and used concurrently for communicating with multiple drones. (See Figure **??** for the interaction diagram)

2.1.3 Message Encoding

Any input that needs to be given to the drone must be encoded in bytes before it can be sent over the telnet connection. Each message is part of an MSP packet with the following components -

1. Header - Each MSP message must start with "$M"
2. Direction - This is used to define if the message is being sent to the flight controller ("<") or is being received from the flight controller (">")
3. Message length - This defines the message length, which can be 16 bits, like for MSP_RAW_IMU or MSP_ATTITUDE, or 32 bits, like for MSP_ALTITUDE
4. Command - This is used to define the type of the packet
5. Payload - This is where the actual data is encompassed
6. Checksum - This value is used to check the integrity of the packet at the receiving end. An 'OUT' packet is only received when requested. This is done by creating and sending an 'IN' packet with the same command and an empty payload.

Every message sent to the drone is encoded in bytes using the pack function of the struct library in python. (Elaborate explanation in the API documentation)

2.1.4 Integration using multithreading

This module aimed to create a fully functional communication unit that could effectively exchange data with the drone calling a few functions. We used multi-threading to send and receive data from the drone simultaneously. We used three threads -

- First one repeatedly requests data from the drone by sending an empty message of the required type.
- The second is used to read the received data. This is done by reading the socket.
- The third is used to pass radio input data to the drone by writing to the socket.

Writer and reader functions are used to access these threads and perform the necessary actions.

- The writer function passes the radio input values from the protocol class to the drone via the socket.
- The reader function reads data from the socket and updates the protocol class variables. We then used these reader and writer functions to write functions for specific actions such as takeoff, altitude hold, set to developer mode, arm, and disarm that the user can directly call.
- Each action has a specified set of RC input values associated with it. We used the protocol class variables to change these RC values and then called the writer function to execute the action. So, the user would have to call one of these functions, and our code would internally determine the set of RC inputs required for the desired action to be executed and write them to the socket.

2.2 Computer Vision

2.2.1 Objective

Computer Vision tracks the drone using the OpenCV library to detect ArUco markers and calculate their orientation from the camera's point of view. The control system uses this information to calculate errors and apply PID control to execute autonomous missions.

2.2.2 Camera calibration

The calibration data (camera matrix, distortion coefficients, rotation and translation vectors) is important since it is used to correct the distortion in the images captured by the camera. Camera calibration is done using the chessboard method. The algorithm (Figure 2) is as follows:

- Images of a checkerboard pattern are captured and the corners of the pattern are used to calculate the camera matrix and distortion coefficients using the OpenCV library's CalibrateCamera module.
- If the checkerboard is detected, its corners are identified using OpenCV and the calibration data is computed using a set of 3D world points and their corresponding 2D image points.

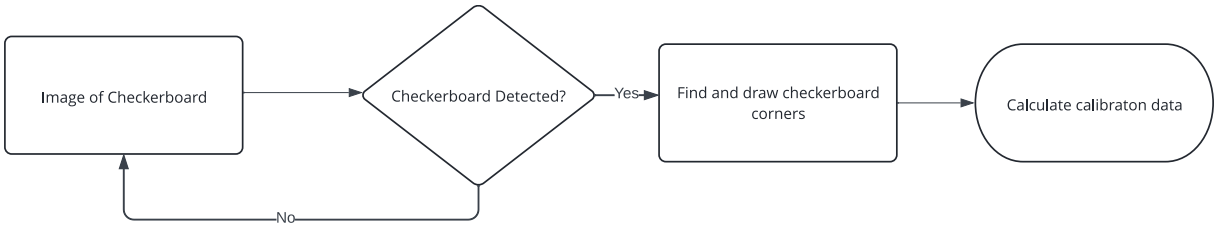

Figure 2: Camera Calibration method

2.2.3 Detection and Pose estimation

ArUco detection and pose estimation is done as follows: (See Figure **??** for the heuristics)

- The camera is initialized and its properties such as resolution, FPS, and codec are specified. We set the codec to 'MJPG', a video compression format to increase camera fps.
- We take frames from camera in real-time to detect and then track ArUco markers. We use OpenCV library ArUco Module which searches for given ArUco image against it's dictionary. If detected it returns a rotation and translation vector of marker concerning camera and corresponding id.
- To calculate the position and speed of markers, Kalman filters in the x, y & z directions are employed.
- These are used to reduce the noise in the directional estimates, the underlying constants and general equations we used are:

$$\hat{X}_k = K_k \cdot Z_k + (1 - K_k) \cdot \hat{X}_{k-1}$$

Where $\hat{X}_k$ is the state, Z_k is the sensor measurement and K_k is the Kalman Gain at the k^{th} iteration.

2.2.4 Issues Faced & Solutions

Cameras must be calibrated across their entire field of view to ensure accurate distance estimation. This is necessary because the camera's measurement and image quality may vary depending on the location within the field of view. Calibrating the camera over the whole field of view can account for any discrepancies, leading to improved accuracy of distance estimation.

Solution - The calibration procedure involves capturing images of a checkerboard pattern from different positions, as shown in Fig 4, in terms of roll, pitch, yaw, and at varying heights to guarantee full coverage of every pixel within the camera's field of view. This approach considers potential distortions and variations, resulting in a more accurate and comprehensive calibration.

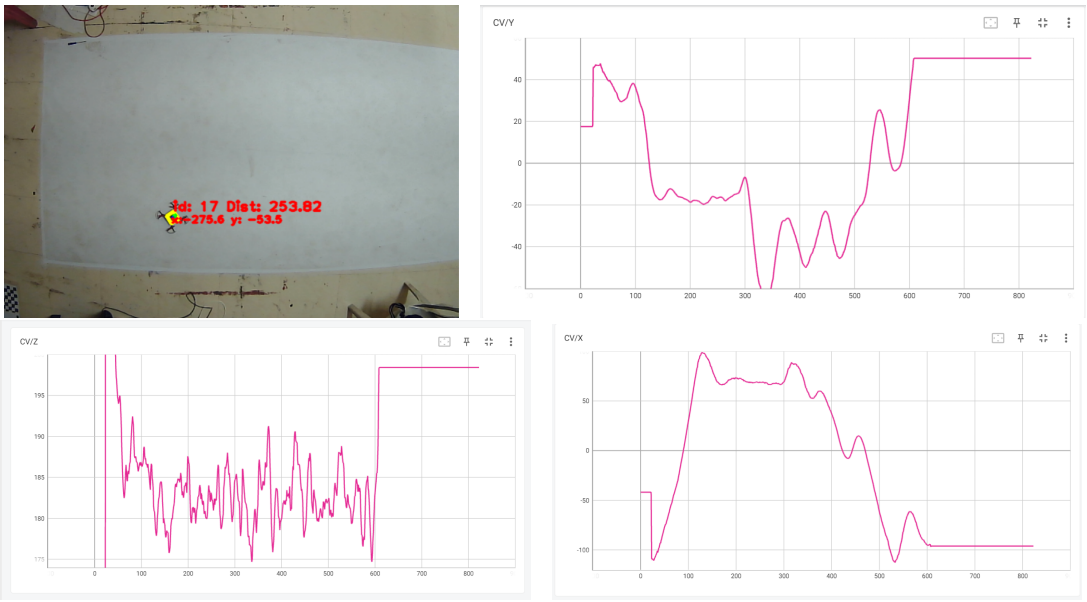

Figure 3: ArUco detection and its X,Y,Z (cm) errors

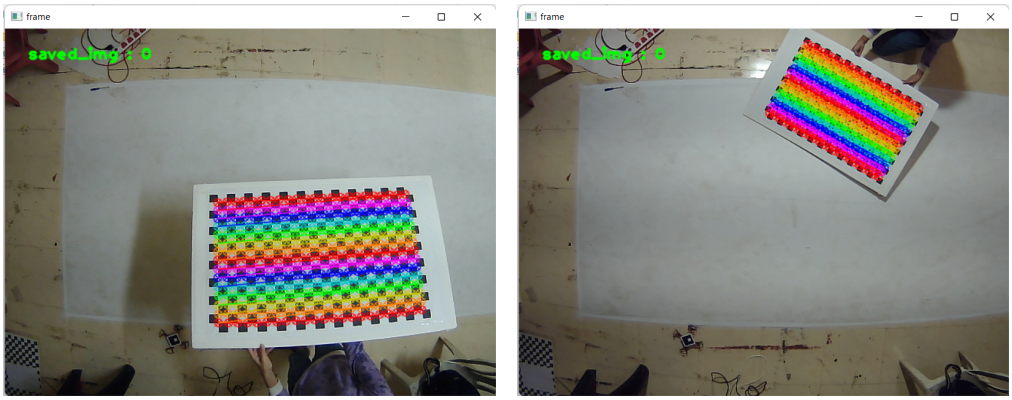

Figure 4: Calibrating camera at different heights

2.3 Controls

2.3.1 Objective

The primary objective of this challenge is to develop a sophisticated control system for the Pluto drone. The drone is equipped with an **ArUco** tag and the vision module is tasked to detect this tag, the system will leverage this information to estimate the drone's position and orientation in real-time with visual odometry. Based on the drone's pose, the system will then navigate the drone to specific waypoints, utilizing a path planning algorithm to generate a trajectory that will allow the drone to form a rectangle of 2 meters in length and 1 meter in width while maintaining a constant altitude of approximately 1.066 m. The control system will consist of a feedback control loop that will continuously adjust the control inputs to the drone to ensure that it follows the desired trajectory.

2.3.2 PID Based Model Free Approach

To control the movements of the Pluto drone, we implement a model-free, sensor-based control strategy. Specifically, we utilize a Proportional-Integral-Derivative (PID) controller, a widely used control method in feedback control systems. It continuously monitors the error between the desired state and the system's current state and generates control signals to reduce the error. This approach

allows for quick and accurate real-time adjustments to the drone's movements and the ability to fine-tune the controller's performance using the controller's gains.

a) PID Control: A PID controller is a feedback control loop mechanism that continuously calculates an error value e(t) as the difference between a desired set-point and the current position by applying a correction based on proportional, integral, and derivative terms. The control law for a PID controller is:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{d}{dt} e(t) \tag{1}$$

Where $u(t)$: control input, $e(t)$: error, K_p: proportional gain, K_i: integral gain, and K_d: derivative gain. The proportional term of the controller adjusts the drone's actions based on the current error, the integral term accounts for any accumulated error over time, and the derivative term predicts future error based on the rate of change of the current error.

b) Control Strategy: The control module takes the desired position and the drone's current position as an input and calculates the roll, pitch and throttle values required to achieve a desired setpoint. These roll, pitch and throttle commands are then sent to the drone's flight controller, where a low-level controller calculates the appropriate motor voltages. Noise in estimation creates a noisy output, therefore, Kalman Filter is applied to account for the noisy fluctuations in the estimated values; this smoothens the feedback data as well as ensures control continuity which ensures that the motors do not receive any jerky inputs. The radio channel of roll and pitch of the Pluto drone has a mean value of 1500 units and a range of -100 to +100 units, allowing for values between 1400 to 1600 units. The yaw angle has an unrestricted range of values from 2100 to 1000. The throttle control, which adjusts the propulsion system's thrust output, has a mean value of 1680 units and a range of -400 to +400 units, allowing for values between 1280 to 2080 units. The motion along the x-axis is achieved using pitch control, along the y-axis is achieved using roll control, and along the z-axis is achieved using throttle control.

c) Tuning: To fine-tune the PID controller, a real-time tuning interface was implemented using the Tkinter library to create a **graphical user interface (GUI)** that includes **sliders** for each PID gain along each of the axes. The GUI runs on a separate thread utilizing Python's threading library to allow for concurrent adjustment of the parameters while the drone is in flight. The values of the sliders were also saved for future use by serializing the data using **pickle** library, which allows for loading the previously set values without manually adjusting the sliders.

2.3.3 Position Control

In order to reach the desired point, we first start by tuning the altitude with the objective of achieving a stable altitude hold. The initial adjustment was made by modifying the gains and closely monitoring the resulting changes in response.

- Proportional gain (K_p) adjusted the system's response to the current error. Increasing the proportional gain caused the system to respond more aggressively to the error, resulting in a faster approach to the target value. However, if the K_p was set too high, the system becomes unstable and oscillates.
- Integral gain (K_i) eliminates the steady-state error. Even if the error was constant over time, the integral gain caused the system to continue adjusting its output until it was eliminated. Increasing the integral gain increased the system's ability to eliminate steady-state error, but it also increased the overshoot and oscillations of the system if set too high.
- Derivative gain (K_d) reduced the overshoot and oscillations of the system. It measured the rate of change of the error and used this information to adjust the system's output. Increasing the derivative gain caused the system to respond more quickly to changes in the error, resulting in less overshoot and oscillations. However, if the derivative gain is set too high, the system becomes unstable and oscillates.

After successfully attaining a stable altitude hold in the Z direction, the next phase involved fine-tuning the controller for the X-Y direction. With the above-mentioned tuning ideology, the roll and pitch control was tuned to obtain the position hold finally.

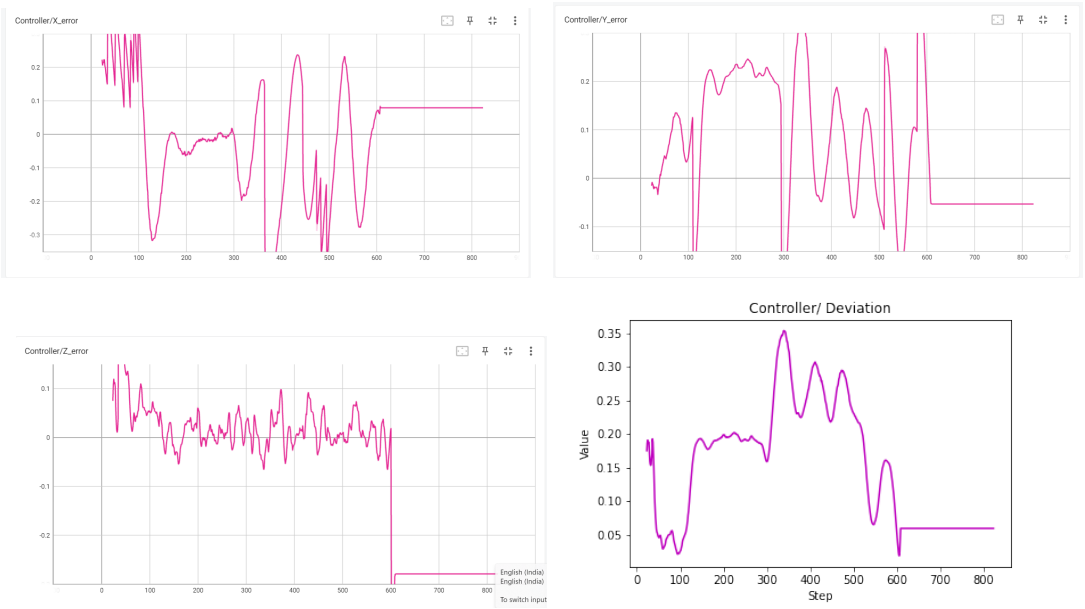

Figure 5: Controller X,Y,Z (m) error and deviation

2.3.4 Trajectory generation and navigation

Subsequently, the project's next phase incorporated trajectory tracking after attaining a stable position hold. To generate a trajectory, the coordinates of the corner points of the rectangle to be tracked are used. Points are sampled between any two corner points based on a specified density. These sampled points along with the corner points served as the desired setpoints for the control system which publishes the control commands at a frequency of 200Hz, enabling the drone to navigate the predetermined path with precision.

2.3.5 Issues and Their Solution:

The battery's health significantly impacted the drone's overall performance, including its range and power output. The propulsion system of Pluto relies entirely on brushed DC motors, which encounter a heating problem in the motors restricting the total flight time, and making the tuning process lengthy. The change in the desired values of the control gains over time can also pose an issue in the tuning process. This can be caused by factors such as wear and tear on the motor, changes in the drone's operating conditions, or changes in the remote computer being used. To eliminate these problems, the following methods were followed:

- Regularly maintaining and replacing the battery to ensure it is in the optimal operating range of approximately 3.7 to 3.9 Volts
- Regularly maintain and replace the motor and other components to ensure they are in optimal condition and stop using them once they are heated.
- Regularly monitoring the gains and adjusting them as needed to ensure optimal performance.

3 Swarm

3.1 Objective

The objective of the swarm challenge is to develop an algorithm that enables real-time communication and decision-making among multiple drones to execute collaborative missions autonomously. Our Swarm algorithm can be used in many ways like in a coordinated drone show featuring multiple drones or getting multiple drone camera views simultaneously with a predefined path.

3.2 Solution overview

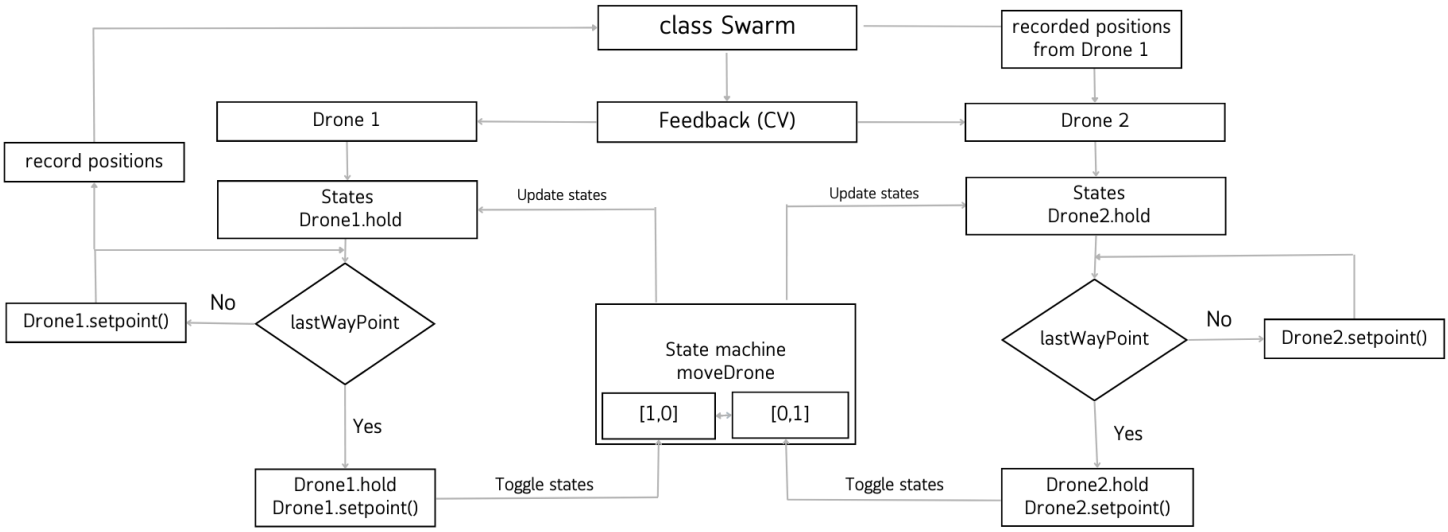

Figure 6: Overall Pipeline

Both drones are connected over the same mobile hotspot using the PuTTy Telnet client. The remote computer used to run the control scripts needs to be connected to this hotspot to proceed further.

A fixed set of waypoints are given to the first drone. The first drone's trajectory is recorded and provided to the second drone as its desired trajectory. Once both drones have completed the rectangle, both land at their respective positions. The algorithm used to execute this maneuver and the control flow have been further elaborated upon below.

Our solution is developed to be generalized to multiple drones. The logic remains the same, however, the state machine will need to be redesigned according to the number of drones in operation.

3.3 Control flow

- Let's assume the second drone is present at (0, 0). With respect to this, the first drone is at(0, y) or (x, 0). The first drone will move to (x, y).
- The swarm class is initialized, and the set of waypoints to be given as input for the first drone is generated. Both drones are armed and the processes for the drones to communicate with the computer are started. Each process deals with the flight of one drone.
- The main process will continuously check the states of both drones and decide when toggling is necessary. This is also responsible for updating the current position of all the drones in the camera's field of view.

- The maneuver begins with the first drone taking off and traveling along one side of the rectangle. The positions of the first drone are recorded and stored in an array to be later passed to the second drone.
- As soon as the first drone reaches the target vertex, it sets its state to hold and toggles its moveDrone flag to zero and hovers, waiting for the second drone to move forward. This change in state triggers a toggle in the state of the second drone, enabling it to move.
- The motion of the second drone has an offset of one vertex of the rectangle with respect to the first drone. When moving from (0, 0) to the start position of the first drone, the recorded positions of the first drone are not used by the second drone but are appended in a list of arrays. Once the second drone completes a side of the rectangle, every list element shifts forward by 1.
- On completing one side, it changes its state to hold and toggles its moveDrone flag to zero, which triggers a toggle in the state of Drone 1. Drone 1 is now free to continue its trajectory.
- This process repeats for all four sides of the rectangle, and once all four sides are complete, both drones land, ending the task.

3.4 Issues and Their Solution:

Every time a drone is added to a single process, the addition of more threads increases the computation time per loop, which in turn reduces communication and increases the error in waypoint tracking.
A centralized swarm system has been developed by assigning each drone its own process in a multiprocessing setup and sharing common variables across processes. This leads to a generalized algorithm capable of controlling a large number of drones through definition of their state machine.

4 Conclusion

With the technical details mentioned above, a python wrapper was developed to communicate with Pluto, estimate its pose using a monocular camera and execute an autonomous mission using PID control. All the methods presented in the report were experimentally verified and produced the desired results to achieve the objective of the problem statement.

References

[1] Bradski, G., 2000. The OpenCV Library. Dr. Dobbx27;s Journal of Software Tools.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] An introduction to tkinter,Lundh and Fredrik,1999