DRONA AVIATION FINAL REPORT

[GitHub Documentation]

Introduction

In **Task-1**, we had to create a python wrapper for the drone that enables users to control the drone without using the mobile application.
Task-2 demanded to use ArUco markers and an overhead camera to localize the drone. We had to implement a PID controller to hover it at a constant position and move it in a rectangular path.
In **Task-3**, we had to create a swarm of two drones where one drone follows the other autonomously in a rectangular trajectory of 1x2m.

TASK 1

The python wrapper has been meticulously built upon the Pluto ROS package provided as a reference. The python wrapper begins an instant connection with the drone by initiating a class that depends on the various categories used, concepts of multithreading, and TCP communication to maintain continuous communication with the drone. Transmission occurs using the MultiWii Serial Protocol (MSP) between them, encoded and decoded in the reader and writer files.

File Description

File	Description
drone.py	This file implements the socket connection and starts the server for the communication between the drone and the controlling system, simultaneously starting the reader and writer threads.
reader.py	This file reads the MSP packets and then compares them to the actual format of the packet defined in the prior document, further extracting the parameters of the drone from that packet.
writer.py	This file contains the writer function, defined for creating the MSP packets for communicating with the drone.
key_command.py	This file logs the key pressed on the keyboard and passes it for further processing.
key_handling.py	This file takes the value of the key pressed from key_command.py file and then calls the respective function for teleoperation of the drone..

Table 1: Python Wrapper Files Description

Code Workflow

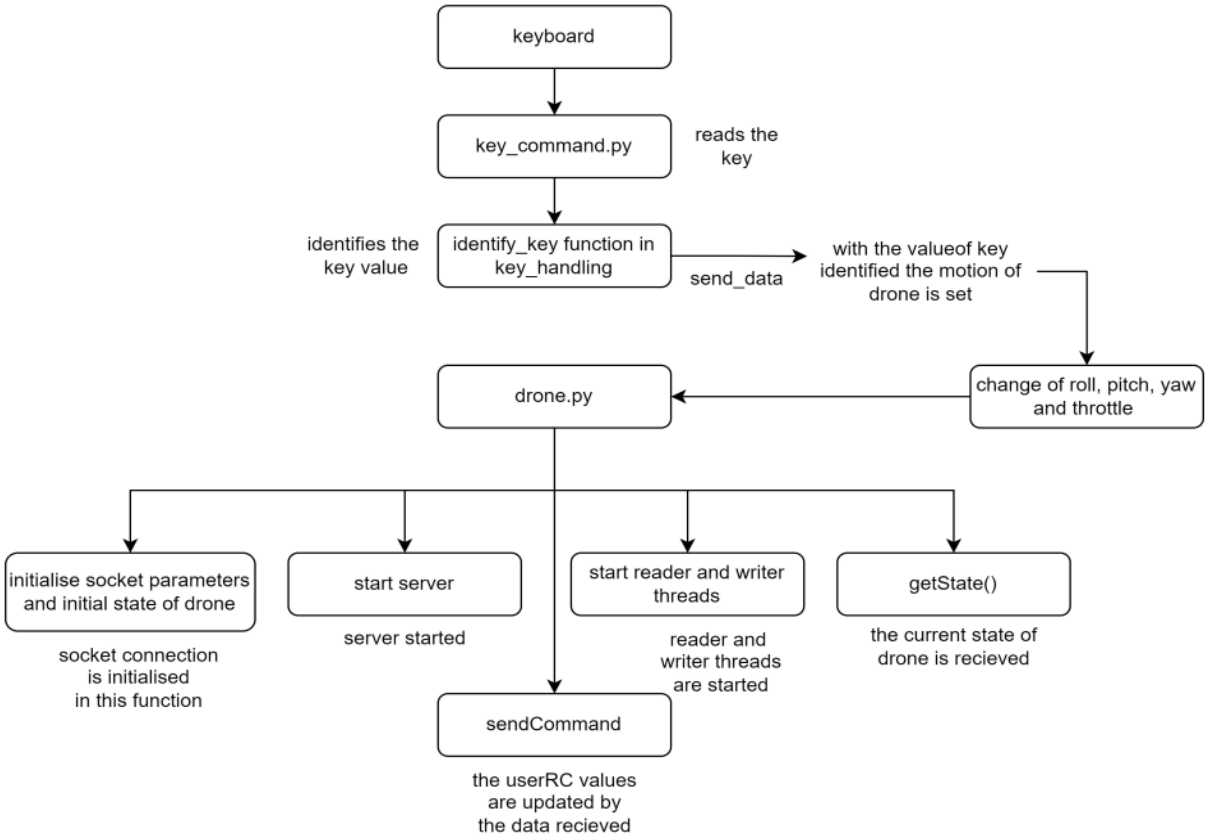

Figure 1: Working of Python Wrapper reflecting the flow of data right from the keyboard to the execution file(drone.py)

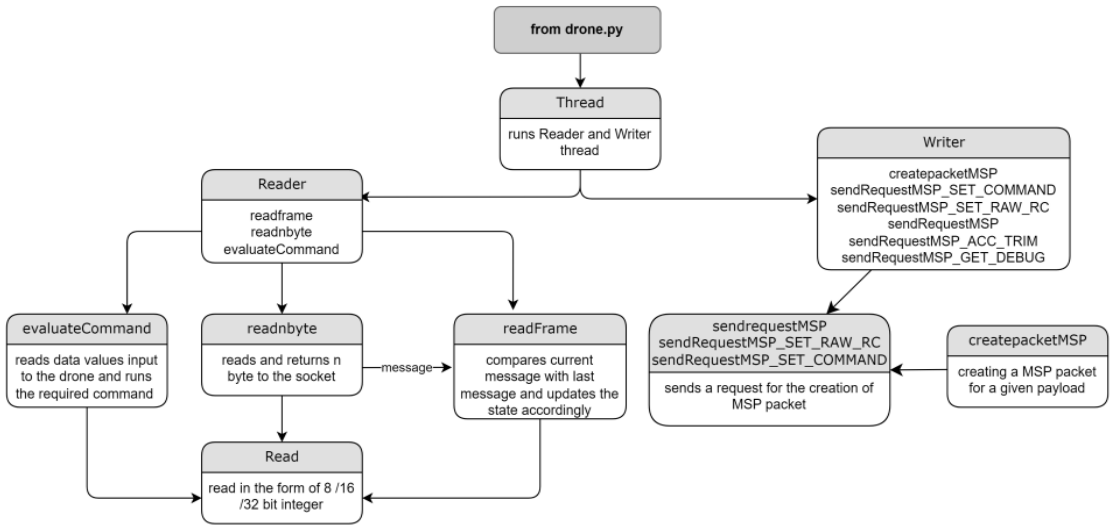

Figure 2: Working of Reader and Writer Threads Highlighting their Respective Fucntions

TASK 2

As per the problem statement, the drone is supposed to hover at a position and move in a 1x2m rectangle maintaining a certain height.
The requirements for the same are:

- **Localization**: Pose Estimation of ArUco Marker
- **Controls**: PID for position and height control
- **Trajectory**: Defines the path for the drone

Camera Used: Logitech BRIO 4K (1080p,60fps)
ArUco Marker: 45x45mm and 4x4 bits (marker ID :0)

Localization

1.1 ArUco Detection and Pose Estimation

ArUco markers are binary square fiducial markers that can be used for camera pose estimation. Their main benefit is that their detection is robust, fast, and simple.

The Pose Estimation process is as follows:

1. Calibrated the camera to calculate its intrinsic matrix and distortion coefficients.
2. Detected the markers using the ArUco library provided by OpenCV.
3. Transformed the point to camera's coordinate frame by calculating extrinsic matrix by solving the classical PnP using OpenCV's SolvePnP().
4. Corrected the noisy height estimation using Machine Learning to get readings accurate enough for optimal height control.

The following figure represents the complete coordinate transformation from 2D image plane to 3D real world.

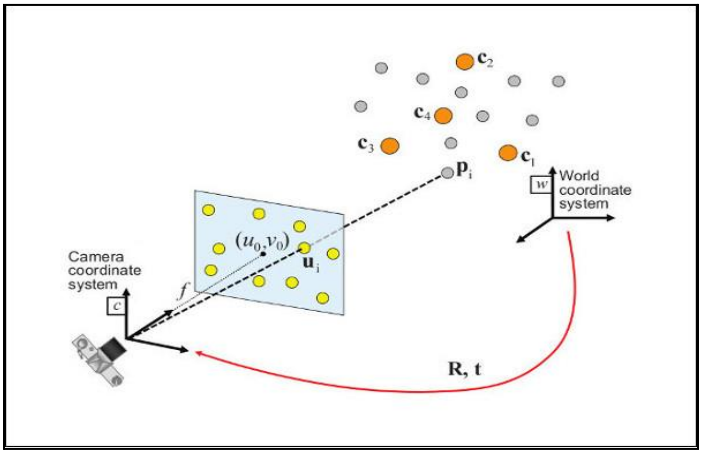

Figure 3: Transformation of coordinates

Implementing classical pose estimation method using arUco marker detection and coordinates transformation, we found the x and y positions quite accurate. For height, the estimations were satisfactory around the center of camera's FOV but were quite erroneous around the boundaries. We have shown some samples of observations and the errors associated in Table 3.

In order to improve accuracy, machine learning came into the picture.
w

1.2 Pose Correction using Machine Learning

The most fascinating and **novel** part of our approach arrives when we include the use of **Machine Learning** in order to increase the precision of the drone's height estimation.

1.2.1 Height correction:

For increasing the precision of height of the marker with respect to ground, we prepared our **own dataset** with the help of a Time of Flight (ToF) sensor, which is a type of scanner-less LIDAR that uses high-power optical pulses in durations of nanoseconds to capture depth information up to ranges of 8m.

Likewise, using linear regression on the pose returned from transformed coordinates, and true height from ToF, we predicted the drone's height precisely.

1.2.2 Noise and Fluctuation:

For better tolerance against noise and fluctuation, we introduce **Kalman Filter**. The filter combines current acceleration from the imu sensor and pose estimate from the regression function to provide a pose estimate for the next time step. This estimate and the current pose estimate are used to provide the final pose to the controller. Other than smoothing data, it also provides protection against fault cases like the camera failure to localize the drone.

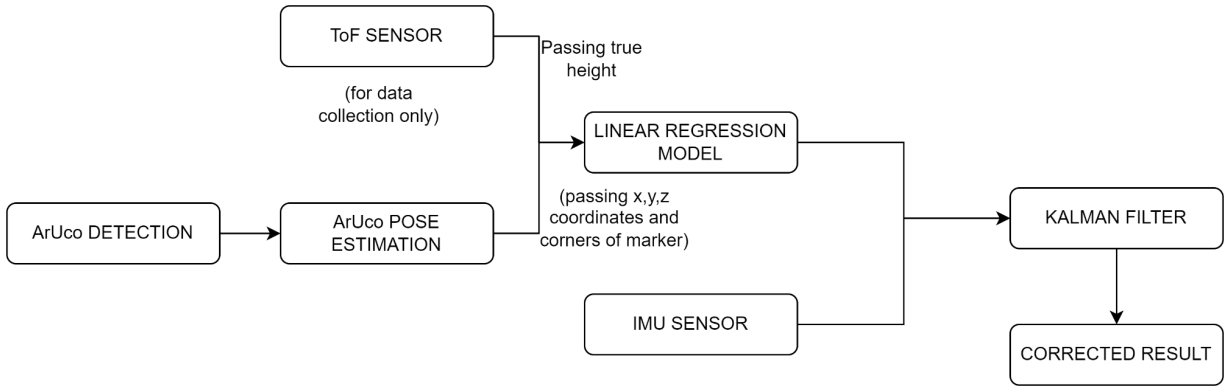

Figure 5: Pose Estimation pipeline

1.2.4 Data:

z_true	z_predicted	z_corrected
1.15	1.56	1.14
0.83	0.98	0.84
1.17	1.32	1.20
Average Error	6.31 %	0.036%

Table 3: Ground truth height, ArUco estimated height, and ML estimated height values

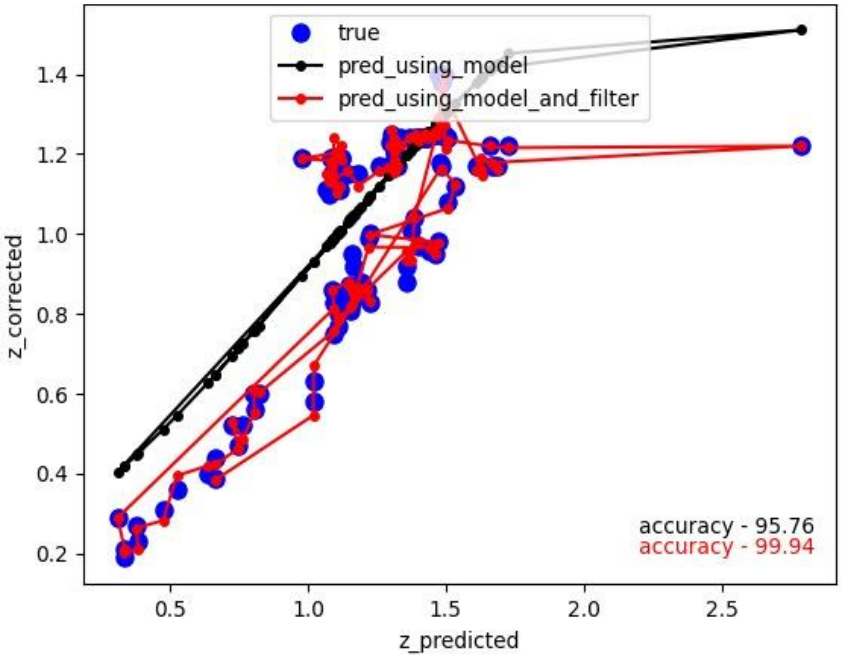

Figure 6: Height estimation

Controls

PID has been implemented to control the movement in the x,y,z directions. PID estimates the value of roll, pitch, and thrust to achieve the required x,y,z coordinates.

We attempted to tune the PID controller for minimizing the error in trajectory traversal. The method includes experimental determination of dynamic characteristics of the control loop and estimating the parameters to produce the desired performance.

Any change in the pitch and the roll of the drone changes the direction of thrust force hence causing an erroneous movement. To counter this, the component of thrust along the z-direction is taken. Similarly, the drone's coordinate system is transformed into camera's coordinate system using yaw values for controlling the x, and y movements. We used **trackbars** to dynamically tune PID values and **matplotlib** library to visualize the system's continuous state and target state continuously.

The errors in the roll, pitch, and yaw are corrected by the use of the following set of equations:

$$Thrust_{PID} = K_{p_1}\Delta z_{err} + K_{d_1}\frac{\mathrm{d}(\Delta z_{err})}{\mathrm{d}t} + K_{c_1}\int \Delta z_{err}dt$$

$$Thrust_{applied} = \frac{Thrust_{PID}}{cos(roll)cos(pitch)}$$

$$roll_{PID} = K_{p_1}\Delta y_{err} + K_{d_1}\frac{\mathrm{d}(\Delta y_{err})}{\mathrm{d}t} + K_{c_1}\int \Delta y_{err}dt$$

$$roll_{applied} = \frac{Thrust_{PID}}{cos(yaw)}$$

$$pitch_{applied} = \frac{Thrust_{PID}}{cos(yaw)}$$

$$Pitch_{PID} = K_{p_1}\Delta x_{err} + K_{d_1}\frac{\mathrm{d}(\Delta x_{err})}{\mathrm{d}t} + K_{c_1}\int \Delta x_{err}dt$$

Trajectory

In order to move the drone in a certain 1x2 m rectangle, we have made a function that returns a specific list of coordinates for the drone to move and all the movement is controlled by PID. Firstly, we have passed the 3D real-world coordinates of the destination to be reached by the drone.
A certain number of steps are fixed between the path and therefore the whole path is broken down into several chunks of paths.
A function continuously checks whether the next checkpoint has been reached by verifying its accuracy with the drone's current coordinates.
Once, the next checkpoint is reached, the step and next coordinate get updated and the drone moves through all these checkpoints till it reaches its final destination.

TASK 3

As per the problem statement, Task 3 asked to control a follower drone that traces a similar path as that of the primary drone, which was completed in Task 2.

Our solution was to utilize **Multi-UAV Swarm Algorithms** to achieve an accurate following of the drone.

A swarm is generally defined as a group of behaving entities that together coordinate to produce a significant or desired result. A swarm of UAVs is a coordinated unit of UAVs that perform a desired task or set of tasks.

Flocking Algorithm

The 3 simple rules, in a programmable sense concerning our task, are:

1. **Alignment**: The follower drone attempts to move in the average direction(or average velocity direction) of the primary drone.
2. **Cohesion**: The follower drone attempts to move toward the average position of the primary drone.
3. **Repulsion**: The follower drone attempts to move away if it gets too close to the primary drone.

Alignment:

The follower drone should look at how it desires to move (a vector pointing to the target), compare that goal with how quickly it is currently moving (its velocity), and apply a force accordingly.

Pseudo Code:

```
align(self, target) {
      Vector desiredVelocity = target.Velocity();
      Vector steerForce = Vector.sub(desiredVelocity, currentVelocity)
       steerForce.limit(maxforce);
       self.applyForce(steer);
 }
```

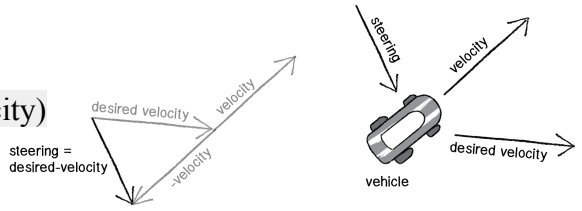

Figure 7: Steering Force

Cohesion:

The follower drone must also try to move toward the average location of the primary drone.

Pseudo Code:

```
cohesion(self, target) {
      Vector relPosition = target.Position() - self.Position();
```

```
        Vector steerForce = Vector.sub(relPosition, target.Velocity() ) ;
        steerForce.limit(maxforce);
        self.applyForce(steer);
  }
```

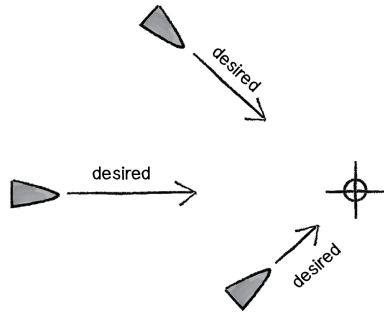

Figure 8: Steering Towards

Separation

The final part of the algorithm is simply to restrict the two drones from getting closer than a threshold distance.

Pseudo Code:

```
seperation(self, target) {
        Vector relPosition = target.Position() - self.Position();
        distance = relPosition.Magnitude();
        if distance < threshold:
                Vector steerForce = -1 * relPosition;
                steerForce.limit(maxforce);
                self.applyForce(steer);
  }
```

Figure 9: Steering Away

Implementation

This entire conception and algorithm were compiled in a simulation using PyBullet and Gym Environment.

Due to time and hardware constraints, we were unable to transform the simulation into a hardware implementation. However, we achieved a highly accurate path-following algorithm, which precisely guides the follower drone in the correct trajectory.

CONCLUSION

We were successfully able to complete Task 1 and Task 2 using the Pluto drone and Task 3 in simulation. The drone was able to continuously hover at a particular position with a maximum error of 5cm and traverse the 1x2m rectangle in only 33s autonomously with some minor deviations. With the help of PID controllers and a Machine Learning integrated Localization System, we ensured that the refinement of motion has taken place and we were able to move the drone in the smoothest way possible. Due to time and hardware constraints, we completed Task 3 in the PyBullet simulation. We implemented the Flocking Algorithm for a swarm of two drones and successfully traversed both drones in only 64sec. This way we were able to successfully complete all the required tasks in this event.

REFERENCES:

1. Pluto ROS Package
2. MSP Packets Documentation
3. Camera Calibration
4. Opencv ArUco documentation
5. Camera calibration working(Pinhole)
6. Linear Regression
7. Kalman Filter
8. PID-Control System
9. PID Controller-Working and Tuning Methods
10. Flocking Algorithm
11. Gym-PyBullet-Drones Repository