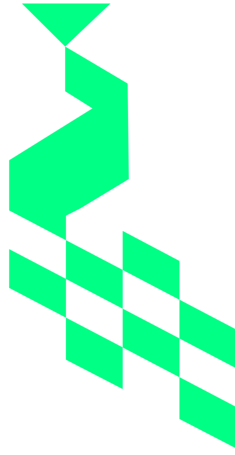


Inter IIT Tech-Meet 11.0

Drona Aviation's Pluto Drone Swarm Challenge

Project Documentation

Team ID - 43

8 February 2023

1 Introduction

This document contains the submission of Team ID 43 for the Drona Aviation's **Pluto Swarm Drone Challenge** at Inter-IIT 11.0. It includes setup instructions and a brief theoretical analysis of the drone's operating principle, as part of the Problem Statement.

1.1 Tasks Achieved -

- **Task 1 :** A self-sufficient wrapper using Python and the MSP Protocol was developed for the provided drone with required capabilities.
- **Task 2 :** A Proportional-Integral-Derivative (PID) controller with minimal error was implemented, utilizing the low-level wrapper from task 1. This allows the drone to automatically move from its initial position to a specified way-point, which is transformed linearly from the camera frame.
  - The drone was successfully made to hover at any given height.
  - The drone was appropriately moved in the pattern
- **Task 3 :** Swarm control of two drones was successfully implemented, with one drone leading and the other following autonomously.
  - The swarm was then configured to follow the same the pattern.

2 Installation and Setup Instructions

2.1 Pre-Installation Assumptions

The following instructions assume that the system has a working installation of Ubuntu 20.04 with Python3 as the default version. Additionally, the necessary video driver (`v4l2`) for the webcam used (oCam-1CGN-U) have already been installed.

2.2 Installing Dependencies

Install the following additional system dependencies:

```
$ pip install numpy pyserial
$ sudo apt install libopencv-dev python3-opencv
```

2.3 Setup Instructions

- Assuming the codebase is present as `~/pluto_ws`. To initiate the pose estimation via ArUco marker, run the camera driver and detection node

```
$ cd ~/pluto_ws/pose_ocam/build
$ cmake .. && make
$ ./ocam 2 # number signifies number of drones expected
```

- For the controller, run controller script in the same workspace in a new terminal

```
$ cd ~/pluto_ws/pluto_control
$ pip3 install .
$ python3 single.py # or swarm.py
```

- Extensive logs of all the attributes related to the flight are generated in `~/pluto_ws/pluto_logs/pose`.
  To view the logs graphically

```
$ cd ~/pluto_ws/visualizer
$ python3 pose.py ../pose/<log-filename>
```

3 Overall Approach and Algorithm Description

3.1 Communication via Wrapper

To use the wrapper, one has to connect to the drone's internal WiFi hotspot and execute appropriate high-level functions. In order to communicate with the drone and send commands to it, we use the python **pyserial** library, which has been used to open a connection to the drone server.

The wrapper takes in the commands of Arm, Takeoff, Land, set the Roll, Pitch, Yaw and Thrust as well as gets the altitude from the onboard controller. The protocol sends out packets which are classified as "In" Packets and "Out" Packets depending on whether the information is supposed is being received from the drone or is being sent to the drone.

Out of the five types of packets mentioned, we have used:

- **MSP_SET_RAW_RC**
- **MSP_ALTITUDE**
- **MSP_SET_COMMAND**

For all the below commands, the packet is made by passing the desired parameter values which include the payload and the type of payload to be converted into a hexadecimal string using an inbuilt python function and subsequently passed on to another function to be made into a packet and then sent to the server, from where it is read and the changes are implemented in the drone.

- **Arm**: Before every flight, the drone needs to be armed. In order to arm the drone, AUX4 of SET_RAW_RC is set to 1699 and transmitted continuously for 2 seconds. The required range is between 1300 and 1700.
- **Setting Roll, Pitch, Yaw and Thrust**: The desired RPYT values are passed and the required packet is made and sent to Pluto.
- **Takeoff**: For taking off, we first check if the drone is armed or not. Then, using MSP_SET_COMMAND and payload set to 1, a packet is sent that prepares the drone for takeoff. Using the Alt_hold mode in MSP_SET_RAW_RC, we set the drone to hold its altitude. Finally, packets are sent with the thrust set to 1650 for 5 seconds after which we set it to the equilibrium thrust (calculated experimentally) and turn off the Alt_Hold mode.
- **Land**: The process of sending the landing packet is similar to the one for Take Off. The payload for MSP_SET_COMMAND being set to 2 instead of 1 and the thrust set to 1540. We are using alt_hold mode to ensure it doesn't drop instantaneously and finally disarm the drone.

3.2 Control System

In the following text, we will set up the theoretical formulation for controlling a drone using MPC(Model Predictive Control) for a limited control problem. The problem specification and the notation to be used is as follows.

- x_D : The desired position
- v_D : The required velocity
- x : The current position
- v : The current velocity
- T : The time horizon for the MPC controller

3.2.1 MPC basics

The idea of the MPC controller is simple.

*Compute a series of inputs that will minimize the error at time T, then supply the first input in this series at each sampling instant.*This idea will become clearer as we continue this exposition.

3.2.2 The simpler case

Let's consider the simpler case when we do not need to control the velocity. Define $e_x = x_D - x$

According to the principle of the MPC controller, we need to figure out the inputs that will get as close as possible to x_D at time T. There can be any number of such trajectories or none at all, depending on the case. In this case there can be

an infinite number of such solutions. Here will choose the simplest of these, the constant acceleration path.
The required acceleration is simple to compute

$$x_D = x + vT + \frac{1}{2}a_x T^2$$

Since $e_x = x_D - x$, we get

$$a_x = \frac{2(e_x - vT)}{T^2}$$

Going back to the MPC view, a series of inputs that causes a constant acceleration a_x will take us to x_D after the time horizon T. The analogous case for controlling only velocity, is simple as well. We are stating the solution here

$$a_v = \frac{e_v}{T}$$

3.2.3 Combining the cases

Continuing, suppose we choose a constant velocity per the current strategy constant acceleration, may not work anymore. This is quite clear since generally a_x and a_v will not be equal. At this point we need to devise a way to reconcile this apparent contradiction. One way to deal with the issue is to choose a more interesting strategy, one that can vary the acceleration so that it may be able to satisfy both the velocity and the position controls.
Let us now formulate the problem a bit differently. Our objective is now to solve the following optimisation problem

$$\underset{a}{\operatorname{argmin}}\ |x_D - x_T| + |v_D - v_T|$$

At this juncture, it will serve to be very clear regarding the notation. x_D and v_D remain the same as before. x_T is the position we expect the drone to be at, if it gets an acceleration a. x_T is, obviously, a function of a. It is trivial to simplify the expression on the right. We will simply write out the final result.

$$a_{desired} = \underset{a}{\operatorname{argmin}}\ \frac{T^2}{2}|a - a_x| + T|a - a_v|$$

The exact coefficients of the two terms on the right depend on T. If we were only controlling either position or velocity, then there is nothing to worry about. But if we are controlling both position and velocity, the two coefficients are equal. If that is not the case then, we will just get a_x or a_v as the optimum. The reason is as follows:
The optimum must lie on the line joining a_x and a_v in the three-dimensional vector space. This is because if there is an optimum which does not lie on the line, then we can drop a perpendicular on this line from the supposed optimum. The intersection of this perpendicular and the line will be at least as good as the supposed optimum. So we need only to consider the points on the line. From here it is trivial to show the truth of the assertion.
when we equate the coefficients, we get $T = 2$. The optimum for the expression, is then going to be any point on the line joining a_x and a_v.

$$a_{desired} = \lambda a_x + (1 - \lambda)a_v$$

λ is between [0,1]. While all points on the line connecting a_x and a_v are solutions, they are not identical. The parameter λ decides the weight given to the two endpoints. A large λ will mean a strong position control and a correspondingly weak velocity control. A small value of λ will have the opposite effect. Thus

$$a = \lambda\frac{2(e_x - vT)}{T^2} + (1 - \lambda)\frac{e_v}{T}$$

3.2.4 Computing the thrust, roll and pitch

We are not controlling the yaw in this problem, thus simplifying the issue greatly. To compute the required thrust, roll and pitch, we only need to apply the laws of motion.

$$f\hat{z_d} - mg\hat{z} = ma$$

It is once again essential to be clear about the notation. f is the required thrust. z_d is the $z - axis$ for the drone, mg represents the drone's weight, z is the $z - axis$ of the ground frame. a is the acceleration computed in the previous section. So

$$f\hat{z_d} = mg\hat{z} + ma$$

This allows as to compute f and z_d, f is the magnitude of the vector in the right and z_d is the direction.
Computing the roll and pitch from here is fairly straightforward. Let R be the rotation matrix that takes the ground frame to the desired drone frame.

$$\hat{z_d} = R\hat{z}$$

where

$$R = R_z(\alpha)\,R_y(\beta)\,R_x(\gamma) = \overbrace{\begin{bmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{bmatrix}}^{\text{yaw}}\overbrace{\begin{bmatrix}\cos\beta & 0 & \sin\beta\\ 0 & 1 & 0\\ -\sin\beta & 0 & \cos\beta\end{bmatrix}}^{\text{pitch}}\overbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\gamma & -\sin\gamma\\ 0 & \sin\gamma & \cos\gamma\end{bmatrix}}^{\text{roll}}$$

$$= \begin{bmatrix}\cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma\\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma\\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma\end{bmatrix}$$

with α, β and γ being the yaw, pitch and roll angles

z is simply $[0, 0, 1]^T$. $\hat{z_d}$ is known. R has two unknowns, roll and pitch, but yaw is known. Remember that we are not controlling the yaw, so we do not need to change it. On expanding the right side of the expression, we will obtain two equations and two unknown which can be easily solved for roll and pitch.

3.2.5 Amending the acceleration

Earlier, we had computed the acceleration to be

$$a = \lambda\frac{2(e_x - vT)}{T^2} + (1-\lambda)\frac{e_v}{T}$$

Now we will throw in the last bit of complexity. Suppose the desired position x_D is not constant, that is, $\dot{x_D} \neq 0$ and $\ddot{x_D} \neq 0$. This is not hard to deal with. To deal with $\ddot{x_D}$, we will just add $\ddot{x_D}$ to a. This is essentially to cancel out the relative acceleration.
To deal with the velocity problem, we can just replace the v with $v - \dot{x_D}$. It is quite easy to see where this comes from. We will further simplify it.

$$e_x = x_D - x$$

$$\dot{e_x} = \dot{x_D} - v$$

Substituting it in, we finally get

$$a = \lambda\frac{2(e_x + \dot{e_x}T)}{T^2} + (1-\lambda)\frac{e_v}{T} + \ddot{x_D}$$

$$\boxed{a = 2\lambda\frac{e_x}{T^2} + (1-\lambda)\frac{e_v}{T} + \ddot{x_D} + 2\lambda\frac{\dot{e_x}}{T}}$$

This is the final expression for acceleration which will be in the direction of $\hat{z_d}$. Thus, we successfully built a controller based on the MPC principle using foundational laws. Observe that there are proportional and derivative terms, which are very elegantly produced.

3.3 ArUco Detection

The main benefit of ArUco markers is that a single marker provides enough correspondence (its four corners) to obtain the drone pose and orientation. Also, the inner binary codification makes them especially robust, allowing the possibility of applying error detection and correction techniques. The marker is made such that it is easily identifiable in an image in various conditions by methods like thresholding, edge detection, segmentation etc. The ID of the marker is identified by a binary value encoded by the black and white squares that compose the marker. This reliable, fast and continuous detection feature in image feeds allows ArUco markers to be used in tasks requiring robust coordinate information, like pose estimation. OpenCV's ArUco marker detection is an algorithm for detecting ArUco markers.

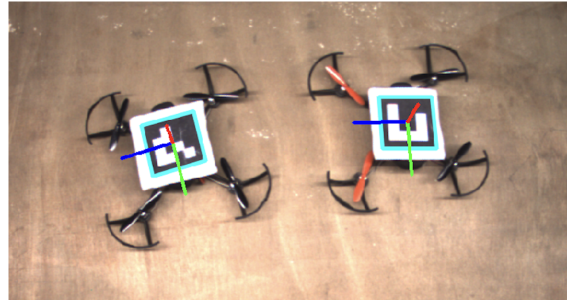


Figure 1: Detection of ArUco markers on the drones

There are two major steps that are being followed for detection -

- **Pre-processing**: The video driver written fetches 8-bit grayscale images from oCam which goes through median blurring followed by sharpening for highlighting edges and suppressing false possible contours.
- **Marker identification**: ArUco markers are square-shaped markers with black and white regions arranged in a specific pattern. The algorithm checks each contour to see if it corresponds to an ArUco marker by analyzing its shape and the pattern of the black and white regions.

3.4 Pose Estimation

Using this theory, we model the mathematics behind the pose estimation of an ArUco marker involving solving a perspective-n-point (PnP) problem:

- **Image plane to camera coordinate system**: Let (x, y) be the image coordinates of a point in the image plane and (X, Y, Z) be the coordinates of the same point in the camera coordinate system. The relationship between the image coordinates and the camera coordinates can be described by the following equation:

$$X = \frac{(x - c_x)}{f_x} * Z \quad , \quad Y = \frac{(y - c_y)}{f_y} * Z$$

  where (c_x, c_y) is the principal point of the camera and (f_x, f_y) is the camera's focal length measured in pixels.

- **Object points to image plane**: Let (X', Y', Z') be the coordinates of a point in the marker coordinate system, (R, T) be the rotation and translation of the marker relative to the camera, and (x', y') be the image coordinates of the same point in the image plane. The relationship between the object points and the image points can be described by the following equation:

$$x' = f_x * \frac{(R[0,0] * X' + R[0,1] * Y' + R[0,2] * Z' + T[0])}{(R[2,0] * X' + R[2,1] * Y' + R[2,2] * Z' + T[2])} + c_x$$

$$y' = f_y * \frac{(R[1,0] * X' + R[1,1] * Y' + R[1,2] * Z' + T[1])}{(R[2,0] * X' + R[2,1] * Y' + R[2,2] * Z' + T[2])} + c_y$$

  where R is a 3x3 rotation matrix and T is a 3x1 translation vector.

- **Solving PnP**: The goal of the PnP solver is to find the rotation and translation (R, T) that minimize the projection error between the object points and the image points. The projection error can be described by the following equation:

$$E = \sum (\sqrt{((x' - x)^2 + (y' - y)^2)})$$

  The PnP solver uses an optimization algorithm to minimize the projection error and find the solution for (R, T), pose of the marker.

3.5 Noise Filtering

Before moving to the technical details, we must answer the *why*. We need filtering because sensors give noisy data, and this has the effect that the position estimates we obtain from the camera are noisy as well. Graphically the line plot of the position with respect to time is jagged with abrupt changes, which is problematic. Our controller needs the derivative of this plot and when we derive this noisy plot, we get a messy function that takes absurd values.

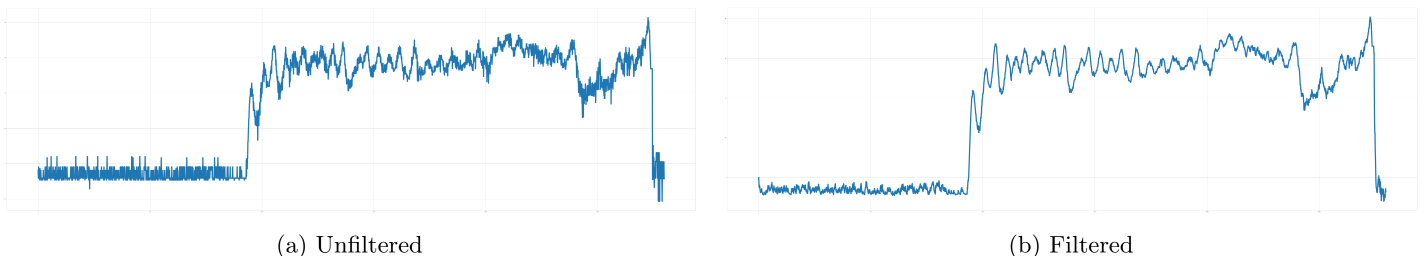
The Linear Kalman Filter (LKF) filters and estimates system states affected by random noise or uncertainty. It helps Newtonian mechanics-based systems with linear equations. The LKF approach analyses sensor data to estimate an item's actual position despite noise or measurement errors. The method forecasts the object's position at each time step using a Newtonian physics-based mathematical model and uses sensor data to correct for errors. The LKF algorithm's system model implies the object's acceleration varies linearly between iterations. The item's velocity and location vary immediately with the acceleration, which is constant for a short duration. This assumption simplifies LKF approach mathematical computations, making it more computationally efficient.

LKF uses the anticipated and measured positions to estimate the item's true position at each time step. The system model predicts the location, while sensor data measures it. The LKF approach employs "optimal estimation" to combine these two data sources and find the object's most likely position. The pseudo-code for the algorithm is as follows:

Algorithm 1 Linear Kalman Filter

1: **Input :** $(X_{k-1}, P_{k-1}, U_k, Z_k, F_k, B_k, Q_k, H_k, R_k)$:
2: X_{k-1} - initial belief vector;
3: P_{k-1} - initial covariance matrix;
4: U_k - control vector;
5: Q_k - process noise covariance;
6: H_k - observation model;
7: R_k - observation noise covariance
8: **Output :** (X_k, P_k):
9: X_k - final belief vector;
10: P_k - final covariance matrix
11: $P_k \leftarrow$ Identity Matrix;
12: $X_k \leftarrow$ Identity Vector

13: **repeat**
14: $\hat{X_k} \leftarrow F_k * X_{k-1} + B_k * U_k$ ▷ Prediction Step
15: $\hat{P_k} \leftarrow F_k * P_{k-1} * {F^T}_k + Q_k$
16: $K \leftarrow \hat{P_k} * {H^T}_k * (H_k * \hat{P_k} * {H^T}_k + R_k)^{-1}$ ▷ Update Step
17: $X_k \leftarrow \hat{X_k} * K * (Z_k - H_k * \hat{X_k})$
18: $P_k \leftarrow \hat{P_k} - K * H_k * \hat{P_k}$
19: **until** end of input
20: **Publish** X_k **and** P_k



(a) Unfiltered (b) Filtered

Figure 2: Z-Coordinate Estimates

In this, we can observe that the first graph is much more jagged and noisy than the second. This is what Kalman filters do for us: Cut the noise.

3.6 Bringing it all together

The ceiling camera detects the corners of the ArUco marker and evaluates the drone's current attitude. It reports the same to the ground station which applies Kalman filter to each of the coordinates individually to decrease noise. The data is then processed by the MPC controller, which sends roll, pitch, yaw, and thrust to the drone via the MSP protocol.

3.6.1 Setting up the Swarm Control

The initial step in **connecting the 2 Pluto drones** with the same ground station is to change from Mode 2 to Mode 3 which is a combination of AP and STA modes. The STA mode allows all the drones to connect to a WiFi to which the ground station is connected. Also, the drones are assigned a static IP for communication. We always opted for Mode 3 (STA + AP) so that the connection to the drone isn't permanently lost because the connection of the ESP module of the router is very unstable. It fails to automatically connect to the router on startup and needs reconfiguration on every reboot. The process to configure the drones for the swarm is:

```
$ telnet 192.168.4.1 23 # establishing connection to the drone
+++AT MODE 3 # setting to (AP + STA) mode
+++AT STA <SSID> <PASSWORD> # ssid and password of the WiFi
+++AT SETIP <IP>  # assign the desired static IP
```

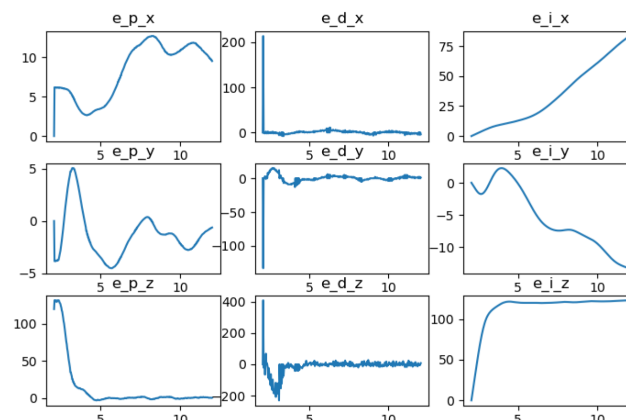
Talking about the overall software architecture, we have two important processes.

- **ArUco detector & Position Estimator (backend)**: Written in C++, this process is tasked with taking the camera feed at high frames per second as input and segmenting out the ArUco marker. As segmenting the images for ArUco detection is intensive, the process itself has been threaded for minimizing loss of fps drop due to computation. One thread takes the image input from the camera and writes it to a shared global variable. The second thread handles processing task by detecting ArUco markers (on image currently stored in the shared variable), estimates the raw pose and sends it to the controller node via socket communication in which this node acts as a client.

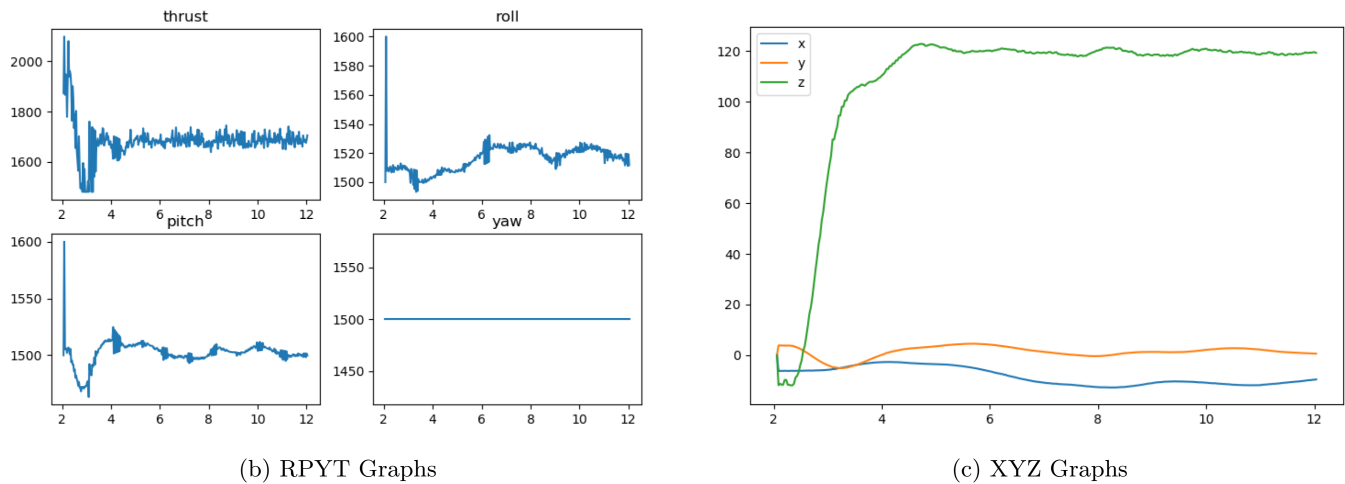
- **Controller Node (frontend)**: This node is written in python and acts as a server for the socket communication. It receives the raw position from the backend, applies kalman filter to it for smoothening, uses the control theory mentioned above to calculate the desired Roll, Pitch, Yaw and Thrust and sends it to the drone using the MSP protocol.

- **Swarm Node**: Controlling both drones at the same time poses synchronization problems. A simple solution is to run two controller threads independently. This is not enough, however since both drones need to traverse the circuit in sync. We set up a locking mechanism which communicated across threads. Each drone would latch onto its current position until the other drone also reached the designated waypoint. Only then would both the drones be allowed to proceed forward. The overall architecture involves a parent thread, which creates two children each of which is assigned to a particular drone.

We also extensively logged the data from every flight, which enabled us to fix bugs, tune parameters and glean information about the drone. Below is an example of the logs from one of the hovering flights:



(a) Graphs of Proportional, Derivative and Integral Errors



(b) RPYT Graphs (c) XYZ Graphs

Figure 3: Graphical Logs

3.7 Experimental Setup

The experimental setup is comprised of a ground station, a camera attached to the ceiling and a Pluto drone with an ArUco marker attached onto it.

(a) Ground Station connected to the Camera (b) Camera duct taped to the ceiling (c) Drone with ArUco attached

Figure 4: Experimental Setup

3.8 Problems Faced

While working on the given drone, the team faced many issues and difficulties, which taught us a lot about handling drones. This included issues involving both hardware and software. The problems varied from broken propellers to noisy data from the camera:

- **High Agility with difficult Control:** The drone has exceptional maneuverability, making controller design and parameter tuning challenging. To tackle this challenge, the team designed an MPC controller that is very agile yet stable and has attributes quite similar to a PID controller.

- **Noisy data and low feedback frequency:** The drone's location was determined using an oCam camera with a detection rate of 30 Hz. However, this data was highly inaccurate with significant spikes, as seen in the logs. To resolve this, a Kalman filter was employed to produce smoother data for improved drone control.

- **Variation of Thrust with Voltage:** When flying the drone, it was noticed that the resulting thrust was influenced by both the published thrust and the battery voltage. To address this issue, we factored in the effect of battery voltage on thrust and updated the published equilibrium thrust accordingly.

- **Fish-Eye Camera and low FOV:** The challenge was to cover an extensive route and thus, a camera was needed that could cover the whole path. The team opted for a Fish-eye camera, which resulted in distorted and low image quality, but allowed for coverage of the entire track. This problem of the fish-eyed camera was partially sorted by using a code for undistorting the image. However, this issue caused an error in the y coordinate, thus leading to slight inaccuracy in the calculation of the height of the drone.