Inter IIT Tech-Meet 11.0

Drona Aviation's Pluto Drone Swarm Challenge

Project Documentation

Team ID - 43

8 February 2023

Contents

1 Installation and Setup Instructions

1.1 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 drivers for the webcam have already been installed.

1.2 Installing Dependencies

Installing Pip & Pip3:

```
$ sudo apt-get update
$ sudo apt-get -y install python3-pip
```

Then, install the following additional system dependencies:

```
$ pip install numpy
$ pip install pandas
$ pip install matplotlib
$ pip install opencv-contrib-python
```

1.3 Version Information for Dependencies

Python Version : 3.8.10
Versions for additional system installed dependencies:

pip	: 20.0.2
pip3	: 20.0.2
opencv	: 4.2.0

1.4 Setup Instructions

- To initiate the pose estimation via ArUco marker, run the detection node.

  ```
  $ cd ~/pluto22_ws/src/pose_ocam/scripts
  $ python3 aruco_ocam.py
  ```

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

```
$ cd ~/pluto22_ws/src/pose_ocam/scripts
$ python3 controller.py
```

- Extensive logs of all the attributes related to the flight are generated in *“src/logs/pose”*
- To view the graphical representation of the logs

```
$ cd ~/pluto22_ws/src/visualizer
$ python3 pose.py ../pose/<log-filename>
```

2 Overall Approach and Algorithm Description

2.1 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 can interpret MultiWii Serial Protocol messages.
As intended, 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_ATTITUDE**
- **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 the changes 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. Finally, a packet is sent with the thrust set to 1700 for a specific period of time after which we set it to the equilibrium thrust (calculated experimentally) and send the required message.
- **Land**: The process of sending the landing packet is similar to the one for Take Off, except for the payload for MSP_SET_COMMAND being set to 2 instead of 1 and the thrust set to 1550.

2.2 Control System

We are using a Model Predictive Control for controlling the drone.
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

2.2.1 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.

2.2.2 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

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

2.2.3 Combining the cases

When we have both objectives, that is, controlling both the position and velocity, things get a bit complex.

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.

We will theoretically formulate MPC using a simple case of constant acceleration. Let us now formulate the problem a bit differently. Our objective is now the following

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

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 ill simply write out the final result.

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

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]. The role of the parameter λ is quite clear.

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

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

2.2.4 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.

2.2.5 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

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

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

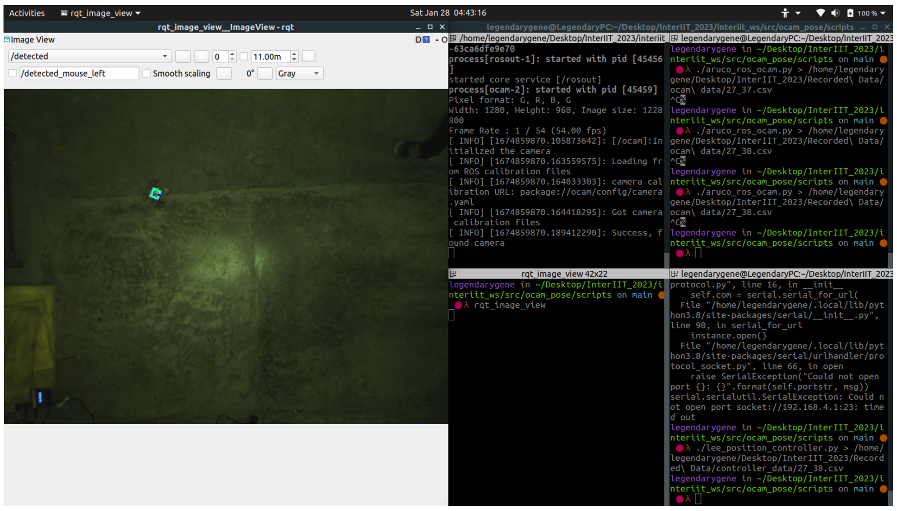
This is the final expression for acceleration which will be in the direction of $\hat{z_d}$. Thus, we successfully built a controller based on the MPC principle using foundational laws. Observe that there are proportional and derivative terms, which are very elegantly produced.

2.3 ArUco Detection

For pose estimation, we are using ArUco markers. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques. The ArUco module is based on the ArUco library of OpenCV, a popular library for detection of square fiducial markers.

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 feature

of reliable, fast and continuous detection 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. It must be noted that a marker can be found rotated in the environment, however, the detection process needs to be able to determine its original rotation so that each corner is identified unequivocally. This is also done based on the binary codification.


Activities
rqt_image_view
Sat Jan 28 04:43:16
100 %
rqt_image_view__ImageView - rqt
Image View
/detected
0
11.00m
/detected_mouse_left
Smooth scaling
0°
Gray
legendarygene@LegendaryPC:~/Desktop/InterIIT_2023/interiit_ws/src/ocam_pose/scripts
/home/legendarygene/Desktop/InterIIT_2023/interiit_
legendarygene@LegendaryPC:~/Desktop/InterIIT_2023
rqt_image_view 42x22
legendarygene@LegendaryPC:~/Desktop/InterIIT_2023

```
-63ca6dfe9e70
process[rosout-1]: started with pid [45456
]
started core service [/rosout]
process[ocam-2]: started with pid [45459]
Pixel format: G, R, B, G
Width: 1280, Height: 960, Image size: 1228
800
Frame Rate : 1 / 54 (54.00 fps)
[ INFO] [1674859870.105873642]: [/ocam]:In
itialized the camera
[ INFO] [1674859870.163559575]: Loading fr
om ROS calibration files
[ INFO] [1674859870.164033303]: camera cal
ibration URL: package://ocam/config/camera
.yaml
[ INFO] [1674859870.164418295]: Got camera
 calibration files
[ INFO] [1674859870.189412290]: Success, f
ound camera
```

```
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ ./aruco_ros_ocam.py > /home/legendary
gene/Desktop/InterIIT_2023/Recorded\ Data/
ocam\ data/27_37.csv
^C%
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ ./aruco_ros_ocam.py > /home/legendary
gene/Desktop/InterIIT_2023/Recorded\ Data/
ocam\ data/27_38.csv
^C%
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ ./aruco_ros_ocam.py > /home/legendary
gene/Desktop/InterIIT_2023/Recorded\ Data/
ocam\ data/27_38.csv
^C%
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ
```

```
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ rqt_image_view
```

```
protocol.py", line 16, in __init__
    self.com = serial.serial_for_url(
  File "/home/legendarygene/.local/lib/pyt
hon3.8/site-packages/serial/__init__.py",
line 90, in serial_for_url
    instance.open()
  File "/home/legendarygene/.local/lib/pyt
hon3.8/site-packages/serial/urlhandler/pro
tocol_socket.py", line 66, in open
    raise SerialException("Could not open
port {}: {}".format(self.portstr, msg))
serial.serialutil.SerialException: Could n
ot open port socket://192.168.4.1:23: time
d out
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ ./lee_position_controller.py > /home/
legendarygene/Desktop/InterIIT_2023/Record
ed\ Data/controller_data/27_38.csv
legendarygene in ~/Desktop/InterIIT_2023/i
nteriit_ws/src/ocam_pose/scripts on main ●
 ●λ
```

Figure 1: Despite the small size, our algorithm detects the ArUco marker placed on the drone

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

- **Pre-processing**: The first step is to convert the input image from RGB to grayscale, as the detection algorithm works better with grayscale images.
- **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.
- **Pose estimation**: Once the ArUco marker is identified, the algorithm estimates the pose (position and orientation) of the marker relative to the camera. This is done by finding the corners of the marker in the image and using them to compute the transformation between the marker coordinate system and the camera coordinate system. The algorithm behind this has been explained in detail later.
- **Decoding**: Finally, the algorithm decodes the information encoded in the marker, such as its ID or any other data.

Given an image containing ArUco markers, the detection process has to return a list of detected markers. Each detected marker includes:

- The position of its four corners in the image (in their original order).
- The ID of the marker.

It may come into one's thought that the marker id is the number obtained from converting the binary codification to a decimal number. However, this is practically impossible since the number of bits is too high for markers of larger sizes and it is impractical to manage such huge numbers. Instead, the marker id is simply the index of the marker in the dictionary it belongs to.

2.4 Pose Estimation

2.4.1 Pinhole Camera Model

The pinhole camera model was assumed in order to implement the AruCo Detection and Pose Estimation. It is simplest camera model which decribes the mathematical relationship of the projection of points in 3D-space onto a image plane.

Let the centre of projection be the origin of a Euclidean coordinate system, and the plane, which is called the **image plane** or **focal plane**, $Z = f$. Under the pinhole camera model, a point in space with coordinates $(X, Y, Z)^T$ is mapped to the point on the image plane $(\frac{fX}{Z}, \frac{fY}{Z}, f)^T$ using straight lines between the camera and the object. Ignoring the final image coordinate, the central projection mapping from 3D world space to 2D image coordinates is,

$$(X, Y, Z)^T \longrightarrow (\frac{fX}{Z}, \frac{fY}{Z})^T \tag{1}$$

The central projection can be expressed as a linear mapping between their homogeneous coordinates in terms of matrix multiplication by,

$$\begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{bmatrix} \tag{2}$$

In theory, the origin of coordinates in the image plane is assumed to be at the principal point, the point of intersection of the image plane and the principal axis. This may not be true in practice, hence, *Eq.* (2) is expressed as,

$$\begin{bmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{bmatrix} = \begin{bmatrix} \alpha_x & s & p_x & 0 \\ 0 & \alpha_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{bmatrix} \tag{3}$$

where the first matrix on the right-hand side is called the **camera calibration matrix**(K). In K, **s** is called the skew parameter(0 for most cameras) and $\alpha_x = fm_x$ and $\alpha_y = fm_y$, the focal length of the camera in terms of the pixel dimensions. (p_x, p_y) is the coordinate of the principal point.

2.4.2 Identifying the Pose

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:

$$X = \frac{(x - cx)}{fx} * Z$$

$$Y = \frac{(y - cy)}{fy} * Z$$

  where (cx, cy) is the principal point of the camera and (fx, fy) is the camera's focal length.

- **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:

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

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

  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.

2.5 Noise Filtering

Before moving to the technical details, we must answer the *why*. Why do we even need noise filtering. The answer is in the mechanics of our controller and the realities of sensors. 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. This is a problem for us. Our controller needs the derivative of this plot. But if we take the derivative of this noisy plot, we get a messy function that takes quite random, useless, and absurd values.

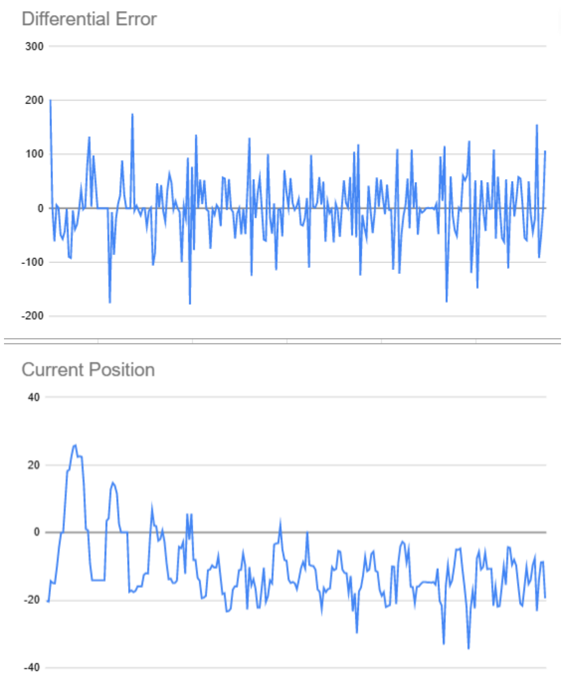


Figure 2: Unfiltered X Estimate, the differential error is, visibly a mess

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.

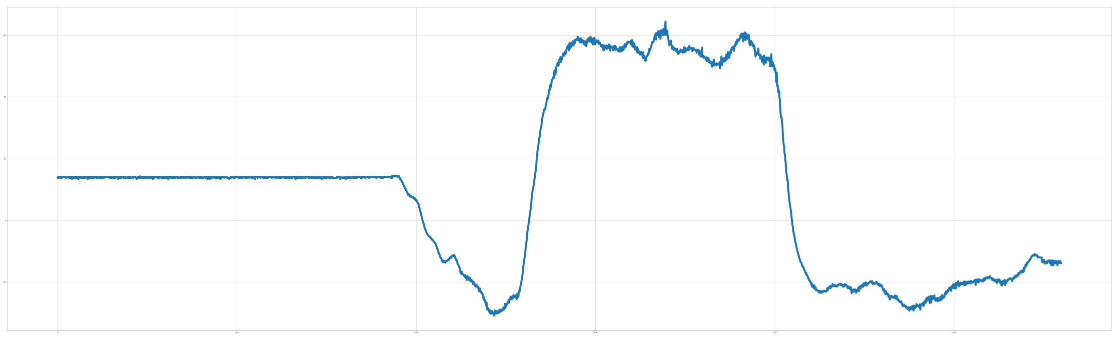
The idea is thus quite simple. We have two sources of data; one is the actual sensor data, which may be noisy. The second is our *anticipation* for the sensor value. A question may naturally arise, why is the anticipated value even useful. This is quite intuitive. We know that despite the noise, the sensor data follows certain trends, and we can model those. Using this model, we can obtain a distribution over the next possible sensor value. We can thus combine these two sources to obtain a much better estimate.

The pseudo-code for the algorithm is as follows:

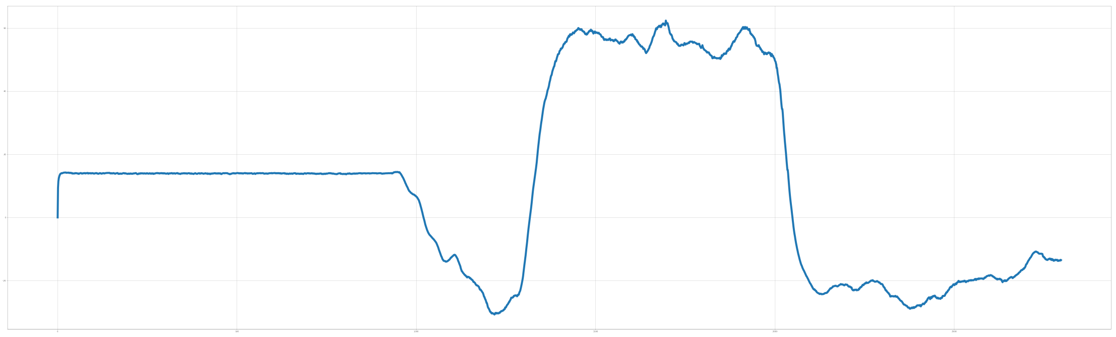
Algorithm 1 Linear Kalman Filter

```
1:  **Input :** $(X_{k-1}, P_{k-1}, U_k, Z_k, F_k, B_k, Q_k, H_k, R_k)$ :
2:
3:  $X_{k-1}$ - initial belief vector;
4:  $P_{k-1}$ - initial covariance matrix;
5:  $U_k$ - control vector;
6:  $Q_k$ - process noise covariance;
7:  $H_k$ - observation model;
8:  $R_k$ - observation noise covariance
9:  **Output :** $(X_k, P_k)$:
10: $X_k$ - final belief vector;
11: $P_k$ - final covariance matrix
12: $P_k \leftarrow$ Identity Matrix;
13: $X_k \leftarrow$ Identity Vector
14:
```

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

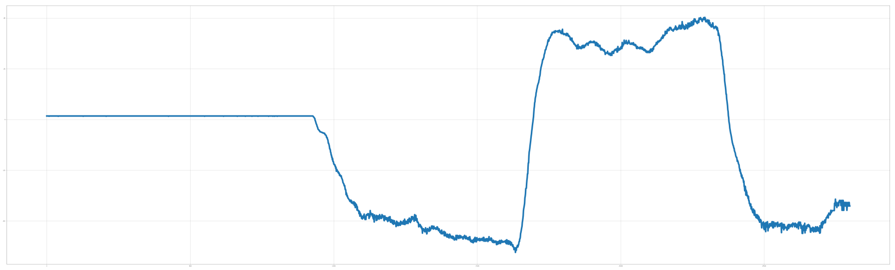


(a) Unfiltered



(b) Filtered

Figure 3: X-Coordinate Estimates


(a) Unfiltered

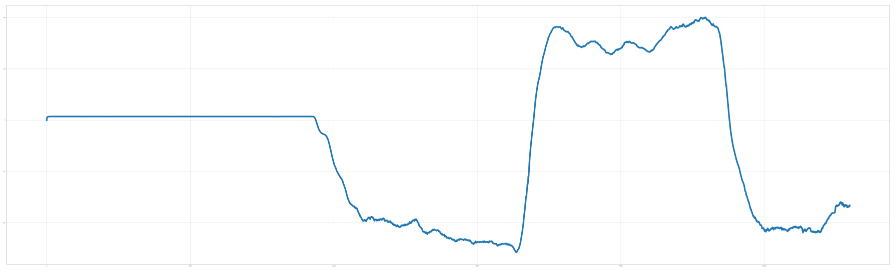
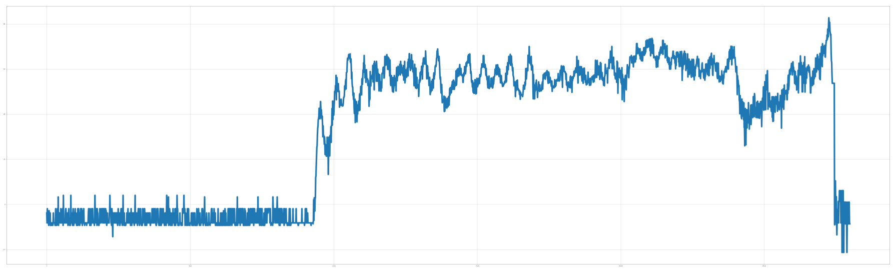

(b) Filtered

Figure 4: Y-Coordinate Estimates


(a) Unfiltered

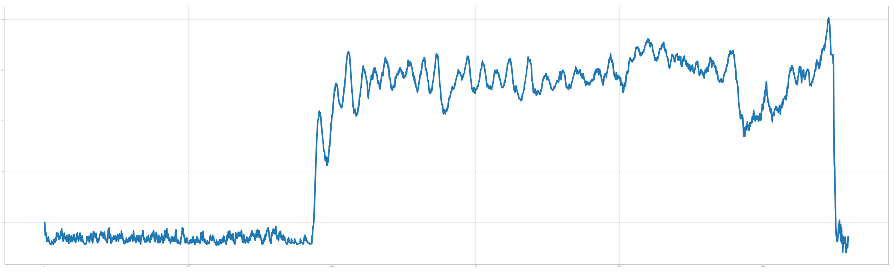

(b) Filtered

Figure 5: Z-Coordinate Estimates

In each of these three pairs, 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.

2.6 Bringing it all together

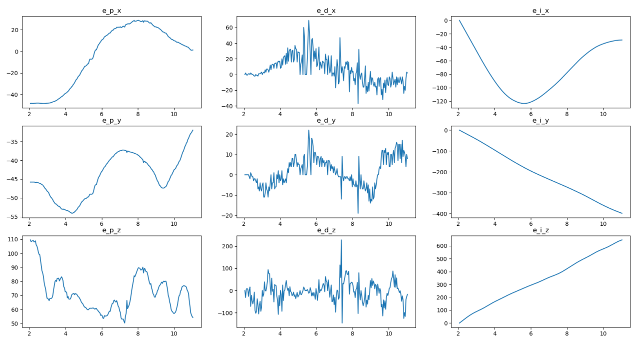
The ceiling camera detects the corners of the aruco marker and evaluates the drone's current attitude. It reports the same to the ground station. The MSP protocol is used to send this data to the onboard controller. The Kalman filter is then applied to the measurements to decrease noise. The data is then processed by the MPC control, which sends pitch, roll, yaw, and thrust to the motors.

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

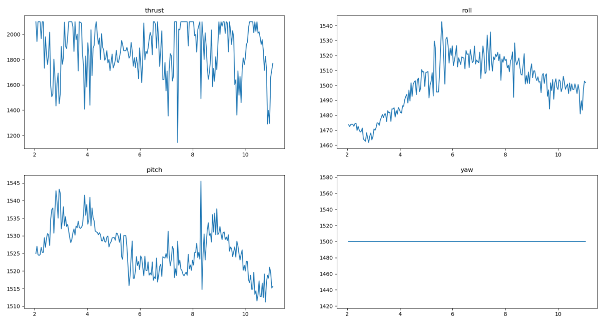
- **ArUco detector & Position Estimator**: This process is tasked with taking as input the camera feed and segmenting out the arUco marker. Then this information is used to compute the position in the real world. This position undergoes smoothing using a kalman filter.
- **Controller Node**: This uses the theory of the controller described above to compute the required roll, pitch, thrust and yaw, which will be needed to take the drone to the required waypoint.

The crucial thing to note about these two processes is that they are independent of each other. The two are not serial, but rather wholly different processes. They communicate using a shared memory space, which can be read by or written to by both these processes. This allows the two of them to run at different loop rates without causing any synchronization issues.

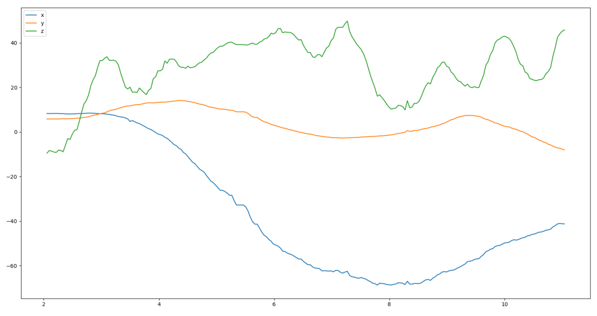
We also extensively logged the data from every flight, which enabled us to fix bugs, tune parameters and glean information about the drone. Below is an example of the logs from one of the flights:



(a) Graphs of Proportional, Derivative and Integral Errors



(b) RPYT Graphs



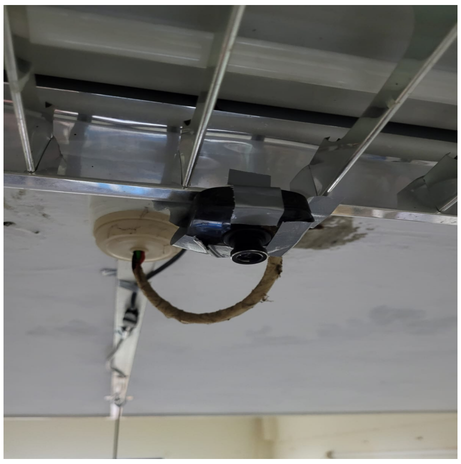
(c) XYZ Graphs

Figure 6: Graphical Logs

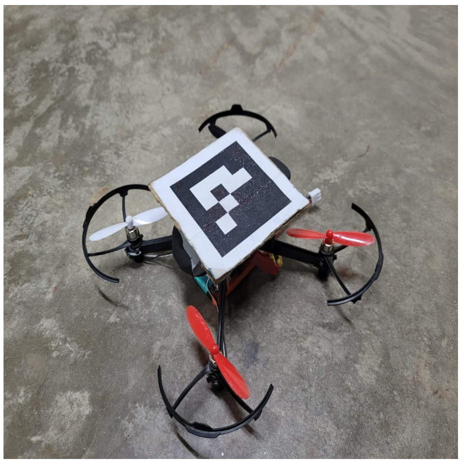
2.7 Experimental Setup

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

(a) Ground Station connected to the Camera



(b) Camera duct taped to the ceiling



(c) Drone with ArUco attached

Figure 7: Experimental Setup

2.8 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:

- **Loss of Propellers:** The drone's high agility, made tuning it very challenging. This process lead to a large number of collisions during testing and tuning. As a result, the loss of propellers was inevitable.
- **High Agility with difficult Control:** The drone has exceptional maneuverability, making controller design and 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:** 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.

2.9 References

- Geometric tracking control of a quadrotor UAV
- Kalman Filter
- openCV-docs
- An Improvement on ArUco Marker for Pose Tracking Using Kalman Filter
- pyserial docs