PLUTO DRONE SWARM CHALLENGE

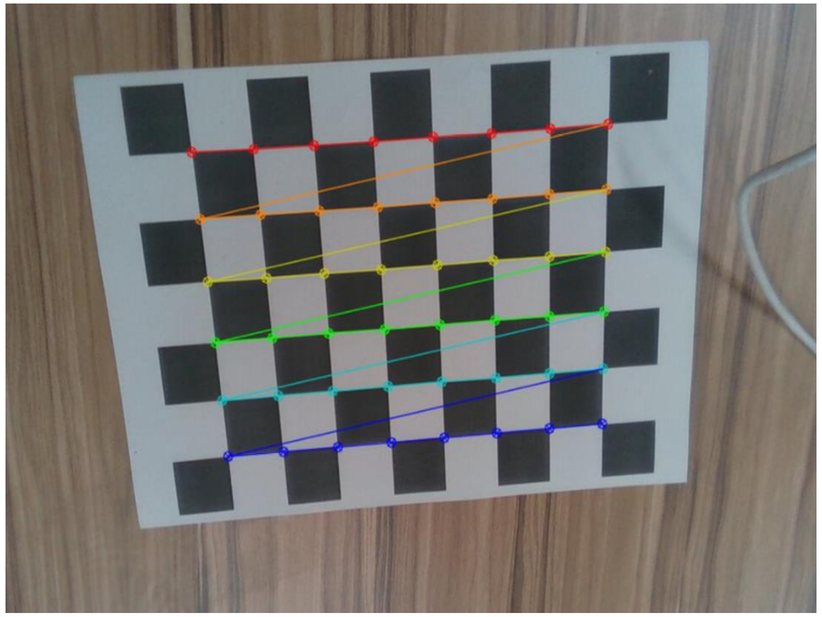
TEAM 21

TASK 1

- The first subtask was to create message packets. This was done using python **struct** library. A connection was established to the drone using the python **socket** library.
- We have created a **Pluto** class which will help users to control the drone.
- It has the following major functions – **ARM, DISARM, TAKEOFF, LAND,BOXARM, SET_ATTITUDE.**
- These abstractions provide complete control over the drone through a simple interface
- The class has support for control using a **XBox360 controller** (works only in Linux). Buttons and controls were mapped to the above functions to send required message packets to the drone through the established connection
- An alternate control system was created to provide similar functionality through **keyboard** inputs in place of a controller using similar methods.

TASK 2

Pose estimation and aruco tag detection:

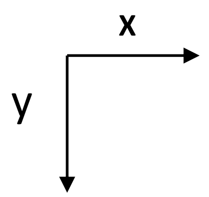


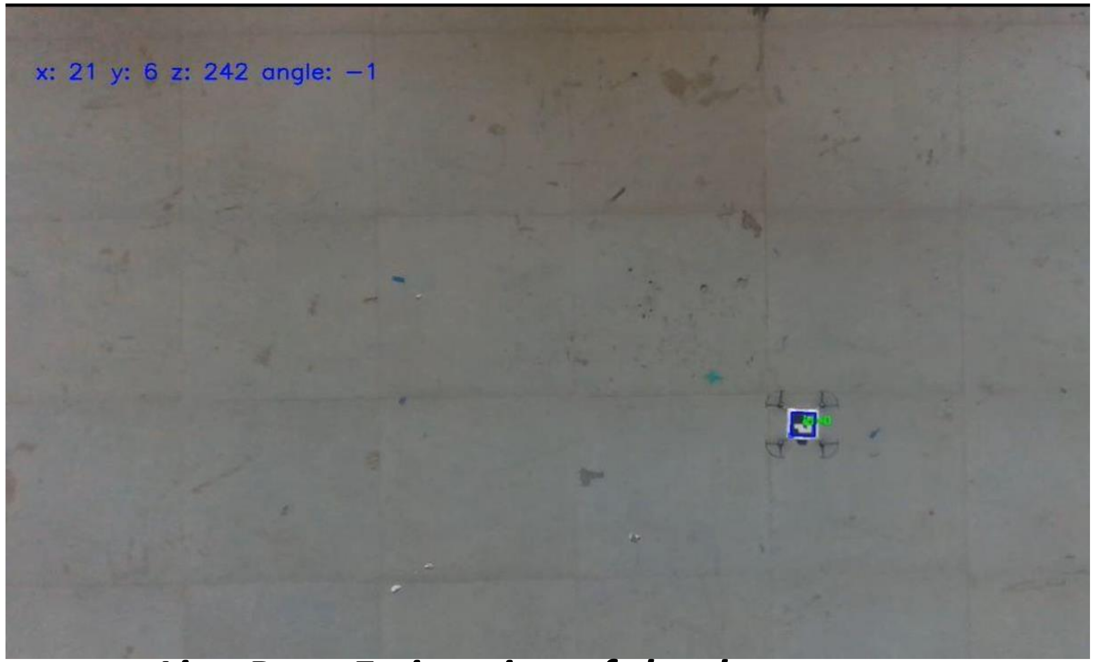
- The camera was calibrated to obtain the **camera matrix** and **distortion coefficients** which is essential for **pose estimation**

- A **User defined** Aruco dictionary , comprising two tags of dimension 4x4 were generated. This helps in reduction of search size for detecting **Aruco Markers**. This effectively increases the rate of detection.

POSE ESTIMATION

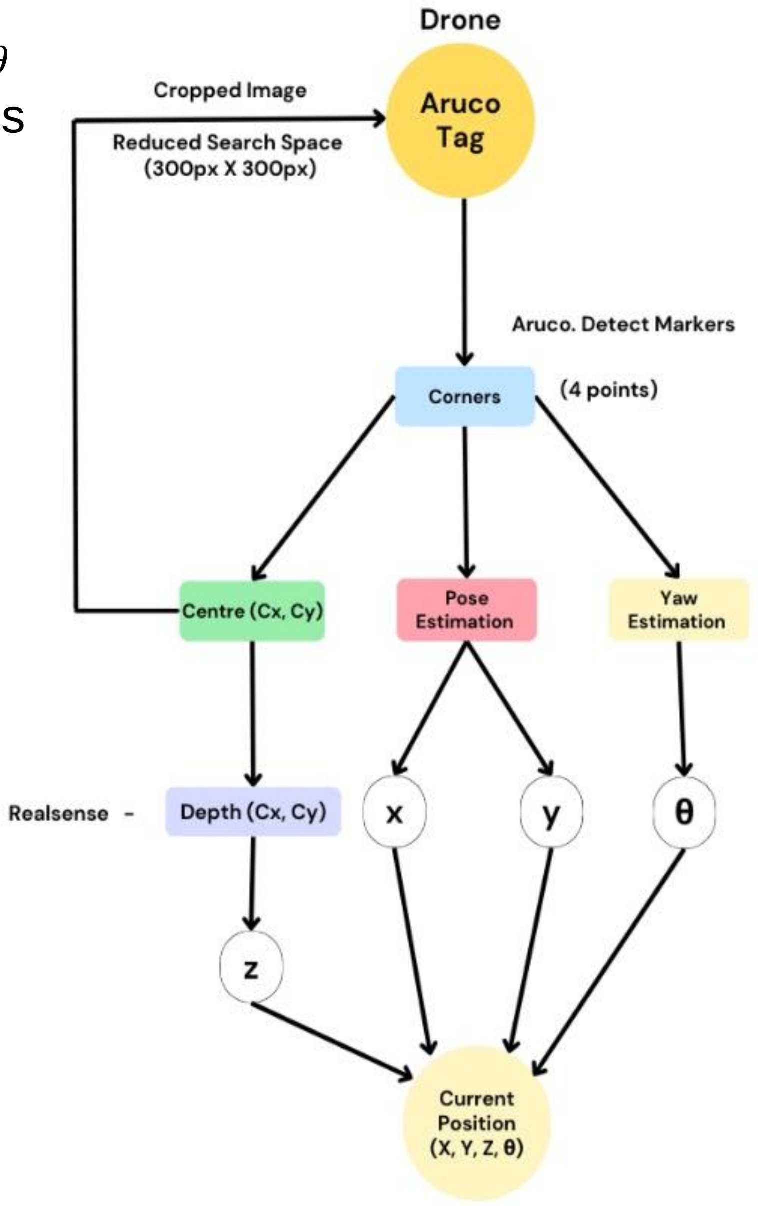
The current position of the drone is identified by **four parameters** : x,y,z,θ
x,y,z are measured **wrt** to the camera while θ is the angle the drone makes with the -ve x axis





Live Pose Estimation of the drone

Reduction in Latency was achieved by cropping the image for Aruco detection to **300px** X **300px** around the previous position of the drone. This led to a reduction of search space by **95.7%**

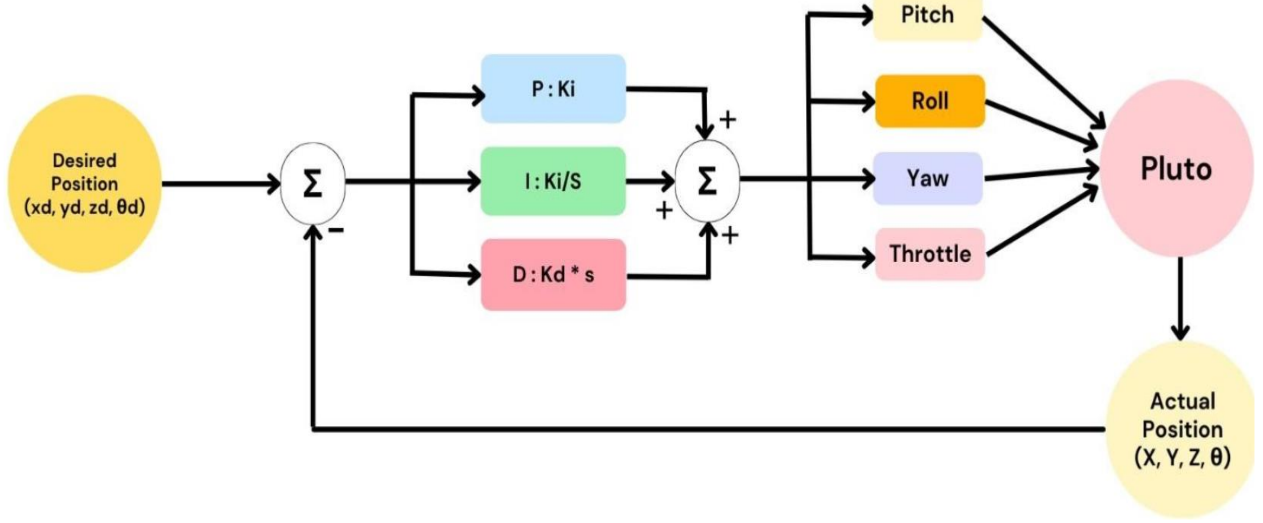


PID CONTROL

- We implemented a **PID** (Proportional - Integral - Derivative) controller for the drone to make it follow the desired trajectory with **minimal error**.

- We tuned the **PID** values through various methods, ranging from simple trial and error, to adjusting the values based on how close or far the drone was to the desired position

- We have three constants for each value whose error we want to reduce: $\mathbf{K_p}$, $\mathbf{K_i}$, and $\mathbf{K_d}$. Hence, for the four Degrees of Freedom we want to control, we have **12 constants.**

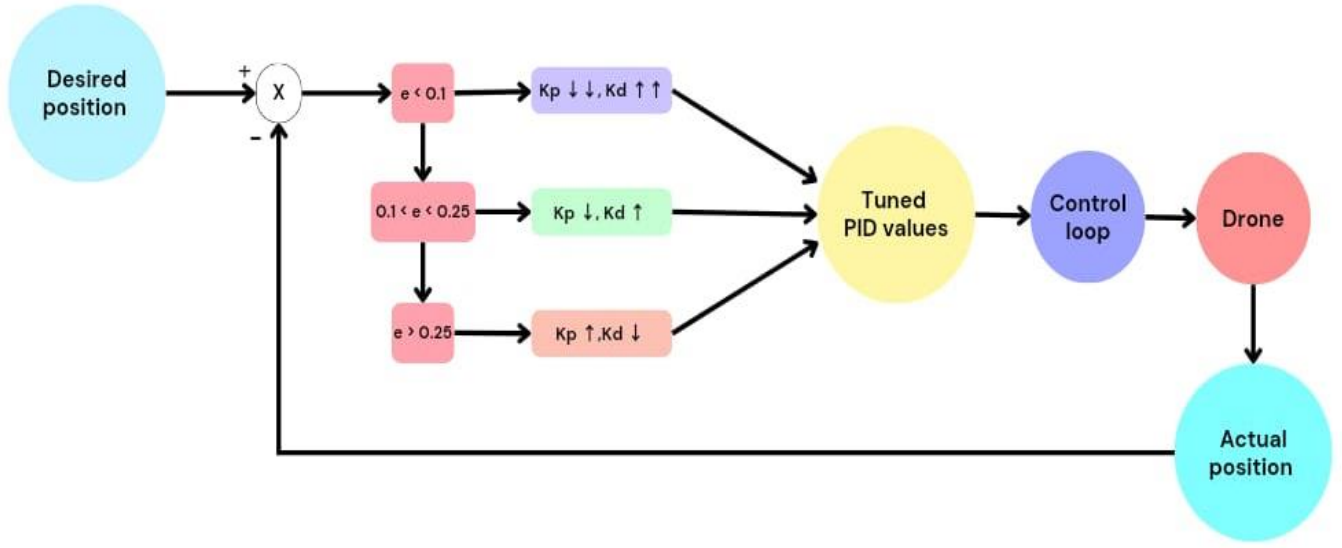

Desired Position (xd, yd, zd, θd)
Σ
−
P : Ki
I : Ki/S
D : Kd * s
+
+
+
Σ
Pitch
Roll
Yaw
Throttle
Pluto
Actual Position (X, Y, Z, θ)

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt},$$

PID TUNING …

Ziegler Nichols Method : This heuristic method involves setting $\mathbf{K_i}$ and $\mathbf{K_d}$ to zero, and increasing the $\mathbf{K_p}$ value till it reaches a $\mathbf{K_{max}}$ value, where the value we intend to control oscillates with a frequency of f_0. Using these two values, we can find the approximate constant values for a PID controller.

Proximity Adjustment : This is an optimization technique wherein we adjust the PID constants based on how close the drone is to the desired position. For instance, as the drone approaches closer to the desired position, we increase the $\mathbf{K_d}$ value and decrease the $\mathbf{K_p}$ value, in order to minimize overshoot**.**



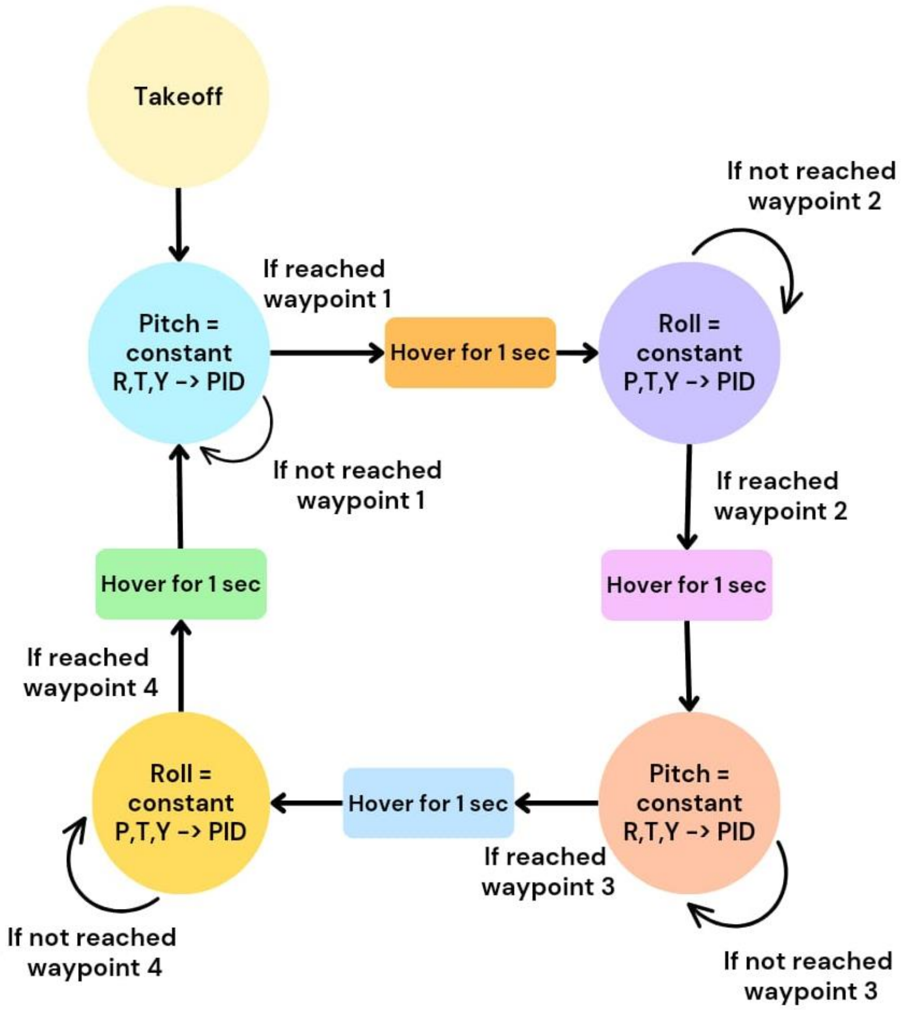
Flow Chart for Proximity Adjustment`

DRONE HOVERING

- Initialise the video stream
- **Take off** and wait for 0.5 seconds
- **X** and **Y** coordinate of drone is stored and set as the desired position
- The **Z** (desired distance from camera) is set to a fixed value (in our case 2m)
- The desired angle is set to 0
- After this the **PID Algorithm** kicks in and tries to stabilise the drone to the desired position
- The drone pose is estimated repeatedly using the method mentioned previously

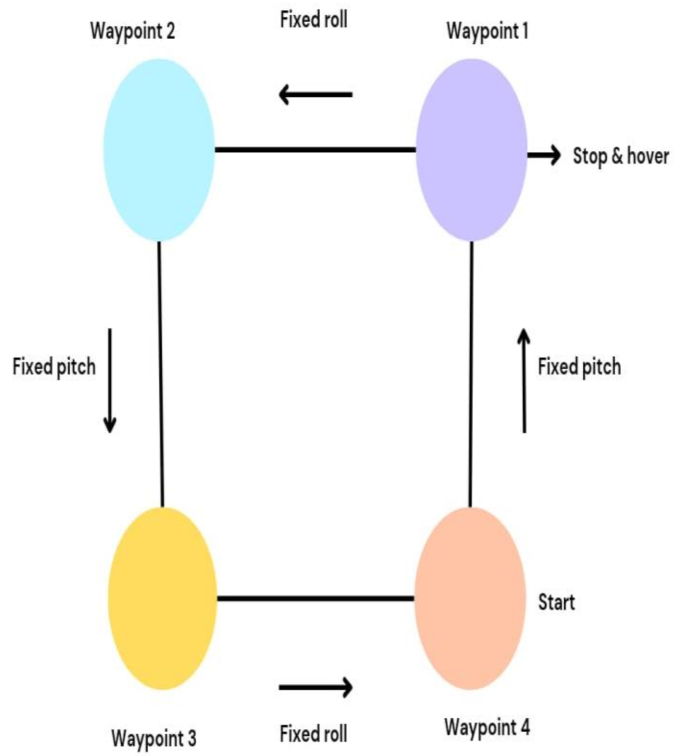
In case, the drone is not detected for more than **20 frames** (implying drone is out of camera's range) , the drone lands.

RECTANGLULAR MOVEMENT



Takeoff
Pitch = constant R,T,Y –> PID
If reached waypoint 1
Hover for 1 sec
Roll = constant P,T,Y –> PID
If not reached waypoint 2
If not reached waypoint 1
If reached waypoint 2
Hover for 1 sec
Hover for 1 sec
If reached waypoint 4
Pitch = constant R,T,Y –> PID
Hover for 1 sec
Roll = constant P,T,Y –> PID
If reached waypoint 3
If not reached waypoint 4
If not reached waypoint 3

State Machine for movement of drone in a rectangle



Waypoint Based Navigation (predefined)

- WAYPOINT 1 : (-1,-0.7)
- WAYPOINT 2 : (-1,0.3)
- WAYPOINT 3 : (1, 0.3)
- WAYPOINT 4 : (1,-0.7)

For fixed z and θ values

TASK 3

- The **primary problem** in this task was to command two different drones at the same time. This problem was overcome using a secondary external **WiFi** adapter.
- Due to this, 2 interfaces were available to us - one internal, built into the laptop, and the other connected externally via USB.
- Therefore, even if both sockets listened to the same IP address, the sockets were bound to different interfaces, so they could not interfere with each other.
- The **Pluto** class was modified to accept an optional interface name in the constructor. An interface can be specified and given that the program is being run on a Linux platform, such that the socket only listens to a desired interface.
- Therefore, two instances of ‘Command’ can be made, each connecting to a different interface, and hence, a different drone.
- This functionality was tested using separate input devices (two Xbox Controllers) and found to be functional.

CLOSING REMARKS

- We were successful in implementing Task 1 which provides an interface to control the drone through a python wrapper
- Although we were unable to make the drone complete a rectangle, we were able to accurately determine the position of the drone and were successful in hovering the drone
- We were able to connect to two drones at the same time without interference. But, due to lack of time and two fully working flight controllers, were unable to implement the Drone Swarm mentioned in Task 3