Pluto Drone Swarm Challenge

Drona Aviation

Final Report

Submitted to Inter IIT Tech Meet 11.0

By

Team 21

Task 1 - Python Wrapper

There are multiple layers of abstraction to help maximize code reuse within the class and provide a clean API.MSP packets were built using the struct package.

The following MSP commands are used and have wrapper functions available:

- MSP_SET_RAW_RC - enable/disable, set roll/pitch/yaw/throttle
- MSP_SET_COMMAND - takeoff/land
- MSP_ACC_CALIBRATION - calibrate the accelerometer
- MSP_MAG_CALIBRATION - calibrate the magnetometer
- MSP_RAW_IMU - read raw sensor values

- MSP packets were made, at the lowest level, using the **`make_in`** command, which accepted a command number and a payload and constructed the required packet bytes (for an IN message)
- To read messages from the drone, we use the **`parse_out`** function, which takes the packet bytes as input, and returns the command and payload (for an OUT message).
- Two functions, **`msp_set_raw_rc`** and **`msp_set_command`**, abstract over the **`make_in`** method and provide ways to construct the corresponding packets by providing the required values:
  - **`msp_set_raw_rc`** takes the attitude values (roll, pitch, throttle, yaw) and the auxiliary values (aux1, aux2, aux3, aux4)
  - **`msp_set_command`** takes the command type as an int (1 for takeoff, 2 for land)
- The **`Command`** class provides another layer of abstraction. It allows one to define a drone by its IP address/interface and provides nicely named methods to control the drone, such as **`takeoff, land, boxarm, arm, disarm, set_attitude, get_raw_vals`**, etc.
  - The methods - **takeoff, land, arm** and **disarm** are self-explanatory in nature
  - The method - **boxarm** can be used to send mean values (close to hovering) to the drone
  - The method - **set_attitude** can be used to send Pitch, Roll, Throttle and Yaw values to the drone
- **`Command`** stores the current state of the drone in its fields, so the code **`command.send()`** can simply build the required message using the internal state
- The **`socket`** library is used to connect and communicate with the drone.
  - Initially, **`telnet`** was used to connect and communicate with the drone. However, the **`telnet`** module in the Python standard library was deprecated in Python 3.11, so it has been removed.
  - Additionally, the **`socket`** library allows binding to specific interfaces, which is useful in Task 3 when connecting to multiple drones is necessary.
- The Command class also has built-in (optional) support for controlling the drone using an Xbox 360 controller (on Linux only), using the xbox360controller Python package.
  - The bindings are
  - Buttons:
    - A - disarm
    - B - arm

        - X - takeoff
        - Y - land
        - Mode - calibrate accelerometer and magnetometer (using MSP_ACC_CALIB and MSP_MAG_CALIB)
    - Left Joystick: yaw and throttle
        - x-axis: turning the drone (yaw)
        - y-axis: changing height (throttle)
    - Right Joystick: roll and pitch
        - x-axis: translate the drone left, and right (roll)
        - y-axis: moving forward and backward (pitch)

Task 2 - Drone Control

1. ArUco tag generation and detection

A custom ArUco dictionary was constructed using the **`cv2.aruco.custom_dictionary`** present in the aRuco module of the cv2 package.

The custom dictionary contained two tags of dimension 4x4. One marker for each drone was used to find the drone's position. We are using a custom ArUco dictionary since it allows the user to have full control over the markers used in their application, including the number of markers, their size, and the identification codes. It also provides the ability to use unique markers for specific tasks, which can improve the accuracy and reliability of the application.

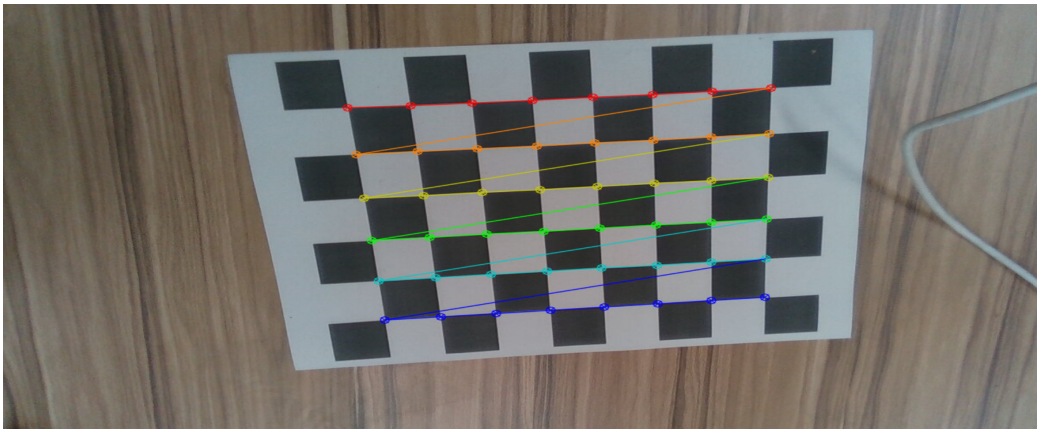
`cv2.aruco.detectMarkers` function is used to detect the ArUco markers in an image and return their image coordinates. A **`custom aruco_display`** function has been used to draw the ArUco markers on the image frame and retrieve the coordinates of the center of marker

2. Pose estimation using the ArUco tag

Here, we first calibrated the camera and then estimated the pose to achieve higher accuracy. For this task, we are using an *Intel Realsense D435* depth camera.

Camera Calibration

We are using the **`cv2.calibrateCamera`** function in OpenCV to calibrate the intrinsic and extrinsic parameters of a camera using a non-linear optimization method.

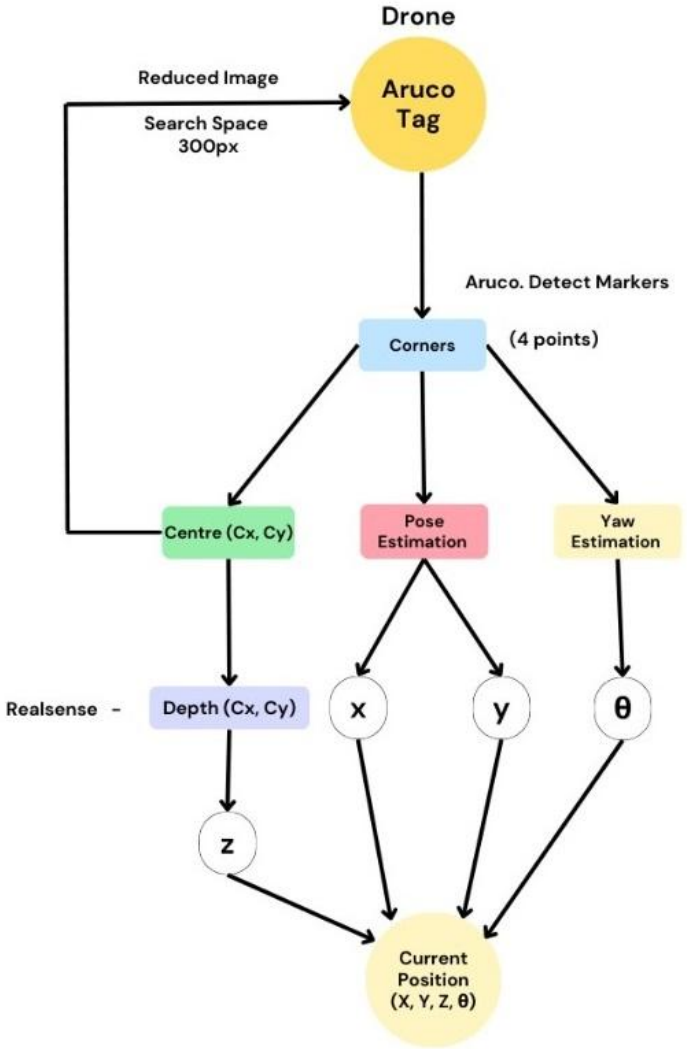


Calibrating the camera using checkerboard

The following steps are being followed.

1. Initializing intrinsic parameters, such as the camera matrix and distortion matrix, and the extrinsic parameters, such as the rotation and translation vectors.
2. The essential input data needed for the calibration of the camera is the set of 3D real-world points (x,y,z coordinates, called object points) and the corresponding 2D coordinates (called image points) of the points in the image. 14 images of a checkerboard were taken from a static camera from different angles and orientations.
3. Observing image points and corresponding 3D world points in each image and storing them in lists.
4. Projecting world points onto the image plane using the intrinsic and extrinsic parameters.
5. Calculating the reprojection error between observed image points and the projected world points.
6. Minimizing the reprojection error by using Zhang's technique, a non-linear optimization method. In this technique, the intrinsic parameters, such as the camera matrix, and the extrinsic parameters, such as the rotation and translation vectors, are updated iteratively until the reprojection error is minimized.
7. Validating the calibrated parameters by computing the reprojection error.
8. Storing the calibrated parameters for future use in upcoming tasks.

As a result, this function returned an estimated camera matrix `cameraMatrix;` estimated distortion coefficients `distCoeffs` and other auxiliary parameters. We will only be using the Camera Matrix and distortion coefficients. These have been stored in calibration_matrix.npy and distortion_coefficients.npy respectively.



Pose Estimation

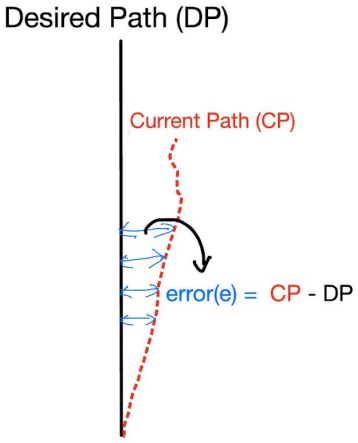
Pose estimation is required to determine the position and orientation of the drone relative to our realsense camera. We are using the **`cv2.aruco`** module for performing pose estimation using ArUco markers. We have defined our custom **`drone_pose`** function that takes the image frame, camera matrix, distortion matrix, and id number of the ArUco markers, which is detected to estimate the pose of that marker.

First, we detect the ArUco marker as specified in the ArUco tag generation part. Then passing, the detected ArUco id number along with the camera matrix, distortion matrix (calculated during the camera calibration step), and the current image frame to the **`drone_pose`** function. Inside the function, to estimate the pose of the marker relative to the

camera, we are using the **`cv2.aruco.estimatePoseSingleMarkers`** function. This function takes the corner points of the marker and a set of intrinsic camera parameters as inputs. It returns the rotation and translation vectors that describe the pose of the marker in 3D space relative to the camera. Our **`drone_pose`** function returns image **`frame,tvec`**, and **`rvec.`** We can call the **`drone_pose`** function multiple times to estimate the pose of multiple markers in the frame.

Pose estimation of the drone via realsense D435 camera at 10ft height

3. PID control for the drone

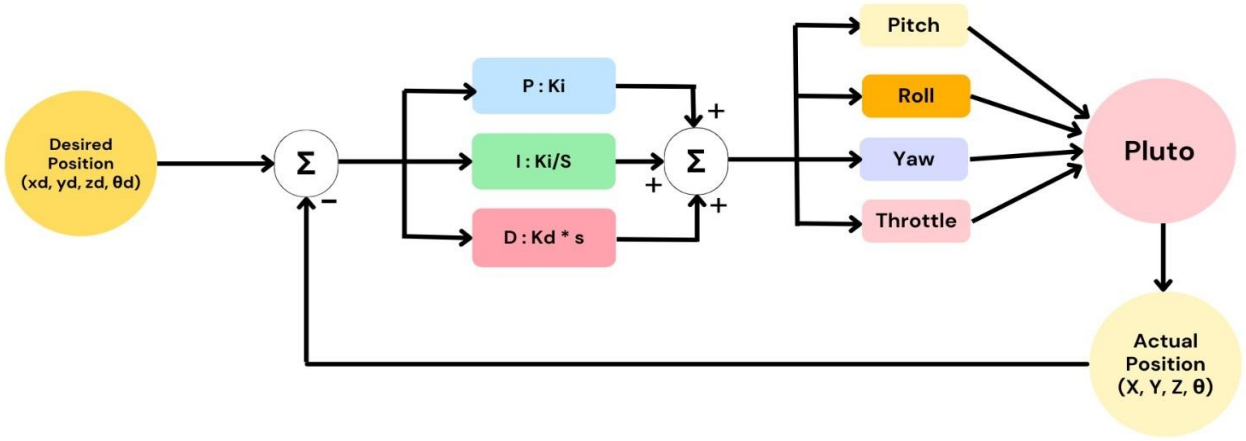


We implemented a PID (Proportional - Integral - Derivative) controller for the drone to make it follow the desired trajectory with minimal error.

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t},$$

For this, the current time is recorded, and the elapsed time is calculated by subtracting the previous time. This elapsed time represents the time between two consecutive frames where the drone is detected.

We set a desired position as our final goal, and for that error between the current position and desired position is calculated. The derivative of the error is determined by dividing the change in error by the change in time.



The proportional term (P) is then calculated by multiplying the error by the proportional gain (kp). The integral term (I) is calculated by multiplying the integral gain (ki) and the accumulated error over time (error_sum). Finally, the derivative term (D) is calculated by multiplying the derivative gain (kd) and the derivative of the error (derr). The sum of the three terms, P + I + D, represents the final control output of the algorithm. This result (P + I + D) is then added to the mean values of the commands.

We implemented this algorithm for all four commands' throttle, pitch, roll, and yaw. Moreover, we got the correction values through PID for each of the commands in real time, and new values were sent to the drone to reach the desired trajectory.

4. Tuning the PID values

- For each value whose error we want to reduce, we have three constants: Kp, Ki, and Kd. Hence, for the four Degrees of Freedom we want to control, we have 12 constants.
- Each constant acts as a gain and amplifies the respective type of error. Hence, it is vital to tune the constants such that the value of the Process Variable(in this case, roll, pitch, etc.) settles to the desired value in a minimum time.
- Since it is impossible to use brute force to tune 12 PID values, we need to use proven techniques to narrow the range for tuning.
- One of these is the Ziegler Nichols Method. It involves setting the Proportional and Derivative components to zero and finding the maximum Kp value beyond which the drone executes unstable oscillations. We note the corresponding constant value KMax and the frequency of oscillation f0. Using these values, we can get the approximate constant values from the standard settings for series-type controllers.
- Ziegler method is a highly aggressive tuning technique tested on linear monotonic controllers, so we could not decide on a fixed value for Kmax from what we observed.
- We tried a technique of adjusting the constant values based on the proximity of the current position to the desired position. For instance, when the drone gets closer to the desired position, the Kp value is toned down, and the Kd value is increased. This technique saw significant success in control, although the results became chaotic and unpredictable.

Here we took different values of gains kp, ki, and kd for different actions in a list containing four elements, one for each pitch, roll, throttle, and yaw, respectively.

These values were obtained by tuning the gains, looking at the curves of each value, and changing the gain to get good results.

5. Reducing Latency

High latency results in poor accuracy of pose estimation with respect to real position, in turn resulting in incorrect inputs to the drone. Therefore, reducing latency is of utmost importance.
Our method for increasing the speed and accuracy of detection of the ArUco tag involved restricting the search space for the tag to a 300-pixel x 300-pixel square around the last known position of the tag. This results in a reduction of search space by 95.7% (as only 300x300 pixels out of 1920x1080 pixels need to be examined).
This reduced the latency greatly (by up to 30%). Overall, this resulted in much smoother drone movement.

6. Hover the drone

One of the main parts of Task 2 was to hover the drone at a desired height. To hover the drone using a PID algorithm, the following steps were taken:

1. Initializing the video stream: The video stream was initialized using the user-defined **`DepthCamera`** function to access realsense D435 for the video stream. This was done to gather real-time information about the drone's current position.
2. Taking off: The drone was instructed to take off, ensuring that it was in a stable position before proceeding with the next steps.
3. Setting the desired position: The desired position for the drone was set as a reference point for the PID algorithm. For the desired position, we took the x and y coordinates same as in the current position of the drone immediately after the take-off. We set the z-coordinate to be equal to the desired depth, measured wrt the camera. This was done so that the algorithm could calculate the error between the drone's current position and the desired position.
4. Implementing the PID algorithm: The PID algorithm was implemented using the method above. The algorithm used the Proportional (P), Integral (I), and Derivative (D) terms to continuously update the values of throttle, pitch, roll, and yaw so that the drone moves toward the desired position.
5. Monitoring the drone's position: The drone's current position was continuously monitored by detecting the ArUco marker present above the drone using the video stream, and the error was calculated in real time. The PID algorithm used this error to make necessary adjustments to the drone's movements, ensuring that it remained stable and hovered at the desired position.
6. Checking the presence of the drone: After every iteration of the above steps, we checked for the position of the marker on the drone. If we didn’t find the marker for 30 consecutive frames, we sent the command for landing the drone to prevent any damage to the drone.

By following these steps, we were able to get the drone to fly and follow the commands given via PID algorithms, but we couldn’t complete it with the accuracy we needed, and the drone was not able to hover for more than a few seconds.

7. Move the drone in a rectangular motion.

We used waypoint based navigation to move the drone in a rectangular way.
The set points were (-1,0.7),(-1,0.3) and (1,0.3) and (1,-0.7) . The drone starts at (1,-0.7) and ends at the same point.

For individual waypoint navigation, we used the PID Control based algorithm used for hovering the drone. The algorithm used for hovering has been encapsulated to a function named **drone_navigation** which takes the desired position as input. As soon as the drone reaches within a certain region of error,the drone hovers using box_arm and returns True . This is repeated for all the waypoints involved

If we don't detect the drone for 20 consecutive frames, we will send the command to land it, to prevent any damage. Moreover, if we want to land our drone for no reason, we can always do so by raising the exception of KeyboardInterrupt.

This was the algorithm we tried for rectangular motion, but here we couldn't achieve accuracy and complete this task. We believe this can be accomplished by tuning the PID gains a few more times to achieve accuracy.

Task 3 - Drone Swarm

One primary problem to solve in this task is to simultaneously connect to, and individually control two drones at the same time. We have accomplished this on Linux using a secondary external WiFi adapter.

- An external USB WiFi adapter was used so that two interfaces were available - one internal, built into the laptop (e.g., `wlp170s0`), and the other connected externally via USB (e.g., `wlp0s20f0u4`). Each interface was connected to the WiFi network created by one drone.
- On Linux, sockets have a SO_BINDTODEVICE option, which forces a socket to listen to only one device (aka interface). This means that even if both sockets listened to the same IP address, the sockets were bound to different devices so that they wouldn't interfere with each other.
- So the `Command` class was modified to accept an optional interface name in the constructor
    - If an interface is specified, and the program is being run on a Linux platform, then the SO_BINDTODEVICE option is set so that the socket only listens to the specified interface
- So two instances of **`Command`** can be made, each connecting to a different interface, and hence, a different drone.

This method was validated by using two separate Xbox controllers connected to the same laptop that were capable of controlling two drones at the same time individually.

However due to time constraints, we were unable to perform the rectangular motion as described in the problem statement. However the crux of the problem has indeed been solved by our team.