InterIIT Tech Meet 11.0
Pluto Drone Swarm Challenge

Team ID : 13
Secondary Team ID : 26

Final Report

February 8, 2023

Contents

List of Figures

List of Tables

0 Introduction

This report provides an overview of the InterIIT Tech Meet Problem statement by Drona Aviation, that focused on executing MSP (Multiwii Serial Protocol) packets and PID (Proportional-Integral-Derivative) controller. The competition aimed to challenge participants to demonstrate their skills in developing and implementing high-level control algorithms for drones. The report covers the methodology, and results, and provides insights into the use of MSP packets and PID controllers.

1 MSP Packets and Socket Communication

1.1 Basic Overview

1. Keyboard commands are received in the terminal using the library pynput
2. The input is then processed and a task is defined for every different input
3. A payload is generated corresponding to the rcValues
4. An MSP packet is generated using the payload and a request.
5. The packet is then sent to the drone using a socket connection.

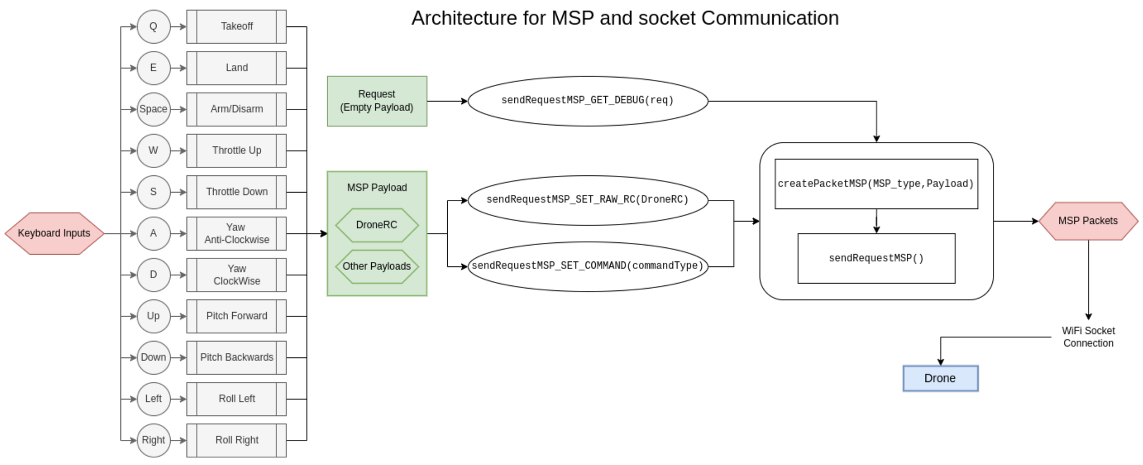


Figure 1: Architecture of the MSP Packets and Socket Communication

1.2 Converting keyboard inputs to payload values

Each keyboard input is mapped to the corresponding action as shown in the flowchart. The payload can broadly be grouped into two categories:

1. **droneRC**: This is the major type of payload used to control the drone movements. The values are internally converted in the drone into the motor voltages. droneRC is a set of 8 values: `Roll, Pitch, Throttle, Yaw, AUX1, AUX2, AUX3, AUX4` each having a range between $900 - 2100$. Each action like Arm/Disarm, Throttle Up/Down etc has corresponding changes on the droneRC values.
2. **Other Payloads**: Certain actions like takeoff and landing require a different type of payload (instead of droneRC values) which fall under this category.

1.3 Requests

Sensor values and other useful data from the drone can be received upon sending 'requests'. These are empty payloads which specify the type of data to be received. The commonly used request is of type `MSP_ATTITUDE` which returns Roll, Pitch and Yaw values from the drone.

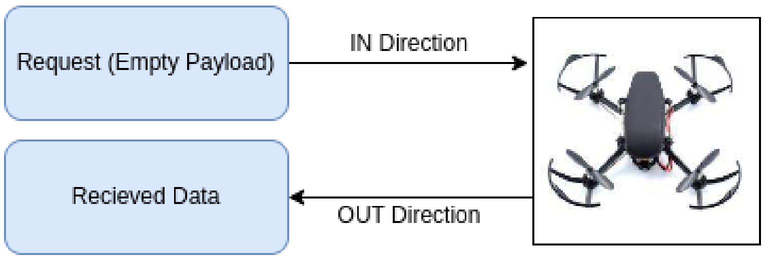


Figure 2: Requests for recieving data from the drone

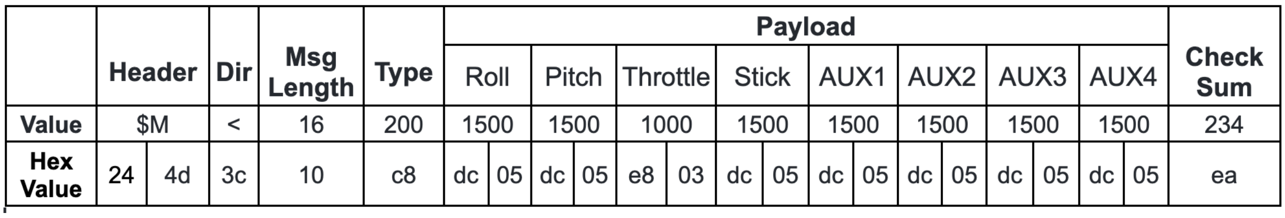
1.4 Payload values to MSP Packets

Pluto uses a Multiwii Serial Protocol (MSP) to receive commands and send back data (like sensor values). Every command is sent as a discrete MSP packet which follows a predefined structure as given below:

Values	Header	Direction	Msg Length	Type of Payload	Payload	Checksum
No of Bytes	2	1	1	1	N	1

Table 1: Structure of MSP Packet

For example, the structure of an MSP packet of type `MSP_SET_RAW_RC` which is used to send RC values to the drone is given below. This MSP packet is used to arm the drone.



	Header		**Dir**	**Msg Length**	**Type**	**Payload**																**Check Sum**
						Roll		Pitch		Throttle		Stick		AUX1		AUX2		AUX3		AUX4		
Value	$M		<	16	200	1500		1500		1000		1500		1500		1500		1500		1500		234
Hex Value	24	4d	3c	10	c8	dc	05	dc	05	e8	03	dc	05	dc	05	dc	05	dc	05	dc	05	ea

Figure 3: Structure of the MSP Packet for 'Arm' Command

1. As mentioned earlier, this is the most useful MSP packet used to send the RC values to the drone.

2. It must be noted that each of the 8 values in droneRC is stored as a 2-byte hexadecimal number in "little-endian" order.

1.5 Socket Connection

For a single drone, the laptop/mobile app acts as the client and the drone acts as the server. The following code snippet creates a client and connects to the drone.

```
TCP_IP = '192.168.4.1'
TCP_PORT = 23
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((TCP_IP, TCP_PORT))
```

Socket connections are used to send messages across a network. We use Python's socket module to achieve this. We create a socket object using `socket.socket()` and the socket type `SOCK_STREAM` uses the TCP protocol. The server is hosted at IP `192.168.4.1` and port 23 (which uses telnet protocol). Once we connect to the server, we use the `client.send()` function to send commands and requests to the drone.

1.6 Code Explanation

The wrapper is structured into two python files.

1. `plutoArm.py`
   This is the main python file to be run. It creates functions for arm, disarm, roll, pitch, yaw etc from the keyboard inputs and further converts them into the corresponding droneRC values. The `userRC()` class implements these functionalities. The script uses multithreading to carry out two processes simultaneously i.e taking Inputs from the keyboard and sending the MSP packets to the drone.

2. `plutoMultiwii.py`
   This python file has the function definitions to create different kinds of MSP packets from the payload values and send the MSP packets to the drone using a socket connection as explained above.

2 PID Controller for the Pluto Drone

2.1 A run down of our PID Schematic

The general philosophy of our control system is we are controlling the RC values for the various movements. `(rcPitch, rcRoll, rcYaw, rcThrottle)`. These controlled RC values are sent to the internal controls of the drone. This translates to the right motor inputs of each of the 4 motors, causing the right motion of the drone.

Let's get into how we control the RC values. We get a pose estimate of the drone from the Aruco marker mounted at the top of the drone. This includes the `Height, X, Y coordinates`. On top of this we are receiving yaw of the drone from its inbuilt IMU. These 4 make our feedback data. We wish to control the RC values corresponding to each part of our feedback data.

Before we get into more details, let's talk about the coordinate system. We are dealing with two coordinate systems - The World Frame (as observed by the camera. XY plane is the horizontal plane) and the Drone Frame (this is the world frame but rotated about the height axis by the yaw reading of the drone).

- When the drone Pitches forward, it moves in the +X direction in the Drone Frame.
- When the drone Rolls to the right, it moves in the +Y direction in the Drone Frame.

4 PID blocks (for each of our RC values) are running simultaneously in our control system. Our desired values are fed into the system as height and coordinate in the XY plane.

- The height value goes into the PID controller for height
- The x,y coordinate is fed with respect to the world frame. This is transformed into the drone frame and fed into the PID controller for pitch and roll accordingly. (the error in x goes to pitch and error in y goes to roll).

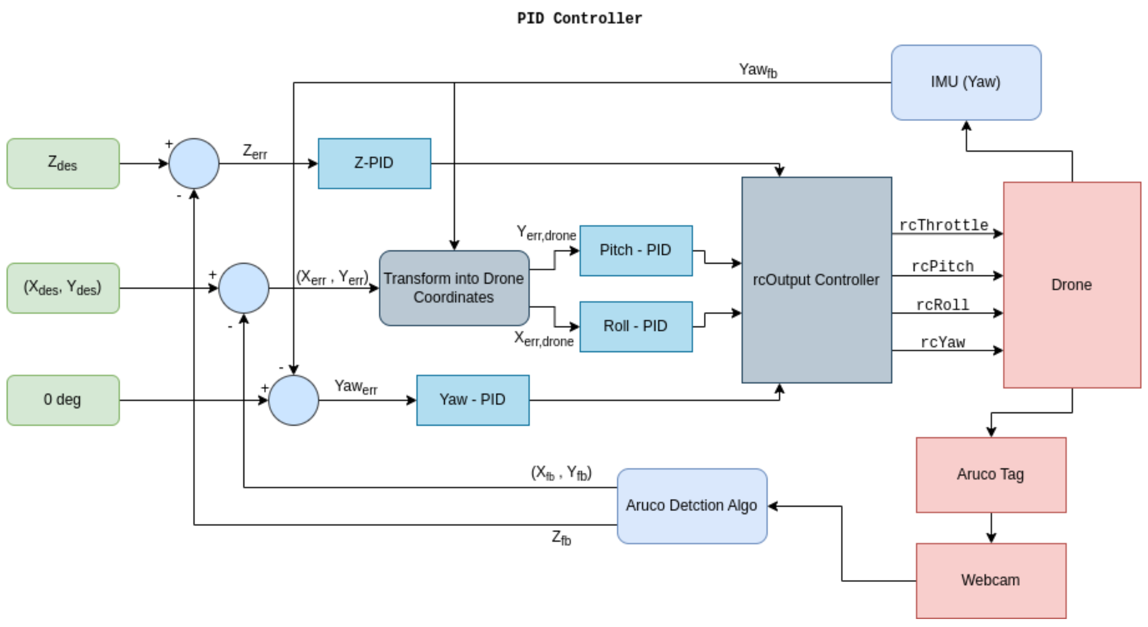


Figure 4: PID

Here's a visual schematic of the entire algorithm.

2.2 Robustness

Simple PID code is often not sufficient in practical purposes due to the noise and non-linearity introduced to the system. So in order to tackle these problems, we have used "Derivative filtering" and "Integral anti-windup" terms.
Derivative filtering :

$$K_d \cdot \frac{d(e(t))}{dt} = K_d \cdot \left[\frac{d(input)}{dt} - \frac{d(feedback)}{dt}\right]$$

The reference input will be constant for most of the time and when input changes, the derivative term will shoot up. So it is reasonable to input the derivative term as

$$-K_d \cdot \frac{d(feedback)}{dt}$$

The value received from the sensor will be noisy with a mixture of high frequencies whose derivative will contribute to a big value. Hence we use a low pass filter that smooths the high frequency contents. The derivative term with filter will be

$$\frac{\alpha K_d s}{s+\alpha} \cdot e(s)$$

Integral anti-windup : Another problem that arises due to the non-linearity in the system is the "integral windup". This is caused by the saturation of plant output at maximum and minimum values. The integral term should stop integrating when the output reaches maximum or minimum value and the error is positive or negative respectively.

2.3 Tuning Process

Since we don't have an exact mathematical representation of the drone system, the best available method is Ziegler-Nicholas method for tuning PID. By this method we first try to tune the proportional part. The K_p is taken such that it is the half value of the gain when the system is about to oscillate. Then K_i and K_d are chosen accordingly to reduce the steady state error and the overshoot respectively.

2.4 Estimation techniques and sensor filters

During our extensive tuning and testing phase of our controller, one major concern and issue we faced was whenever the detection of the Aruco went out, the drone used to behave erratically.

There are 2 major situations where the drone goes out of detection.

- One is the usual noise/flickering detection caused by the resolution of the camera and frames where the Aruco was either not detected or a separate set of pixels was detected as an Aruco marker.
- The other situation is when the drone goes out of frame.

In an ideal scenario where the model and dynamics of the drone was known to us, we would've made use of an Extended Kalman Filter for nonlinear dynamics estimation. Since we do not have an easy way of obtaining the system dynamics, we will have to use other methods to deal with sensor noise and undetected frames.

So, on top of the controller code, a condition was imposed that makes the drone default to RC values that brings it down to the ground every time the Aruco goes out of detection for an extended period of time. (Not when the detection flickers). To tackle the flickering issue and prevent the drone from dropping down at each flicker, the RC input defaults to the values it was being fed just before the flickering until the detection comes back on.

The addition of these fail-safes enhanced the entire tuning process and ensured we focused more on the actual tuning process rather than sensor estimation challenges.

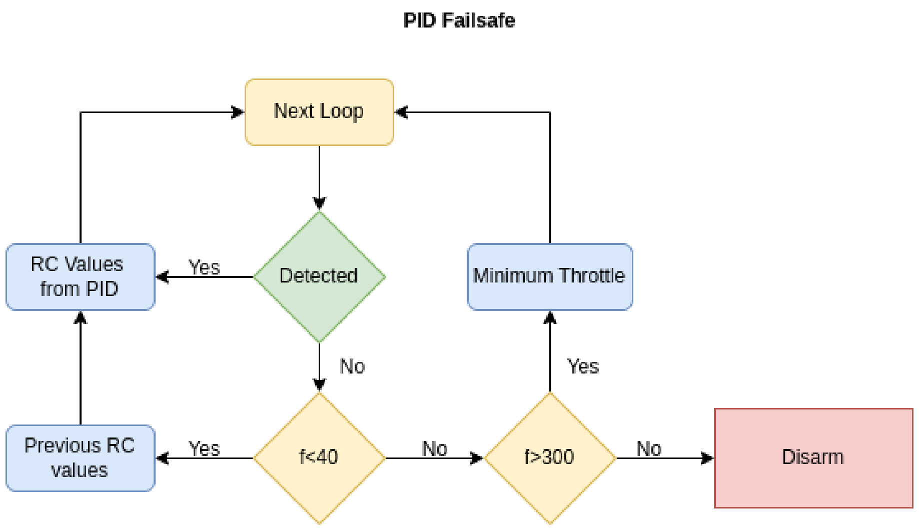


Figure 5: Fail-safe Mechanism

2.5 Rectangle Motion

Once a stable hover and control over drone movements was achieved, we worked on the rectangle motion. The logic used is as follows :

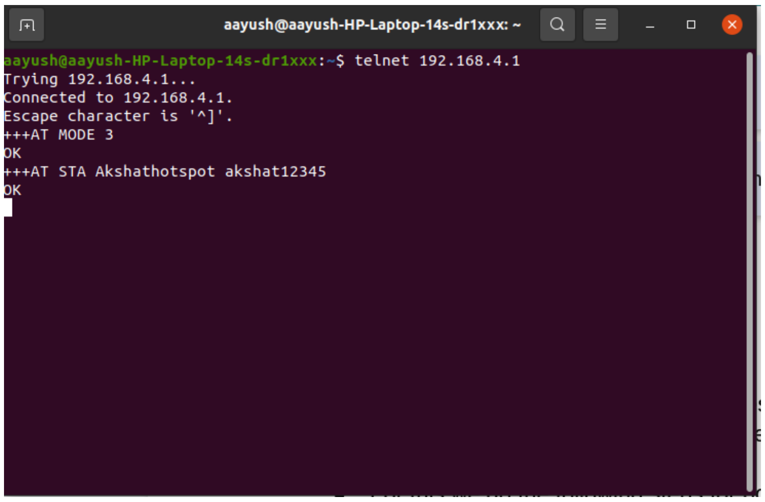
1. We determine the coordinates of the 4 corners of the (1×2) rectangle beforehand.
2. The desired x, y coordinates are dynamically changed as the code runs.
3. Suppose the drone is moving from point 1 to point 2. Here our desired x,y coordinates that are fed into the controller would be that of point 2.
4. As soon as the drone is within a reasonable distance from point 2 (tolerance) the desired coordinates x, y changes to that of point 3.
5. The process continues until the last set point is reached, at which point the drone hovers down to the ground.

3 Pluto Swarm Implementation

3.1 Enabling Drone connections with mobile Hotspot

To be able to connect to multiple drones and control them with our wrapper, we need the drone to behave both as server and client, or more precisely, set both the drones in both Station and Access Point Mode.
For this we do the following steps for both drones :

1. `telnet 192.168.4.1`
2. `+++AT MODE 3` (will set the drone to both Station and Access Point Mode)
3. `+++AT STA [ssid] [password]` (insert your hotspot name and password here)
4. Once this is done, note the IP Address assigned on our hotspot device



```
aayush@aayush-HP-Laptop-14s-dr1xxx:~$ telnet 192.168.4.1
Trying 192.168.4.1...
Connected to 192.168.4.1.
Escape character is '^]'.
+++AT MODE 3
OK
+++AT STA Akshathotspot akshat12345
OK
```

Figure 6: Connecting Drone to Hotspot

3.2 Setting up list of drones as multiple clients

We create a client object for each ip address and store it in a list. Each client can now be accessed from the list using an index, which corresponds to a particular drone.

```
client1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client1.connect(("192.168.117.107", TCP_PORT))
client2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client2.connect(("192.168.117.52",TCP_PORT))
mul_clients = []
mul_clients.append(client1)
mul_clients.append(client2)
```

3.3 Establishing Connection between System and Drones

We now connect our system with the same network, which can now communicate with both the drones and control them simultaneously.
We now proceed to rewrite the previously made functions for arming, takeoff, sendRequest etc for a multiple client system

3.4 Implementing connection and MSP request functions for multiple drones

1. For a single drone we were dealing with 8 values in droneRC and userRC as follows `[Roll, Pitch, Throttle, Yaw, AUX1, AUX2, AUX3, AUX4]` and accordingly developed the packet creation, `MSP_SET_RAW_RC` and `MSP_GET_DEBUG` functions.

2. We now add another value to the userRC and droneRC, which is the index of the drone we want to control, which makes the list size as 9. We rewrite the functions with the index as an argument, and the corresponding client to which the data has to be sent is referenced from the `mul_clients` list.

3.5 Swarm algorithm used

To implement the swarm motion, we use multiple flags to ensure that the drones can roughly follow each other on the rectangle. The first thing we do is define safe tolerance zones about each of the points. Following the diagram, if the first drone is at position one and second drone on position two in the beginning, the first drone is directed towards position two and the second towards position three. This is done by using the PID targets. Now once both of the drones reach the tolerance zone about their specific point, they are directed to go to the next point.

Given the fact that the PID cannot be perfect and we will run into sensor inconsistency issues, the algorithm is designed such that the next set points for both the drones will not be assigned unless both of them have reached the tolerance zones at least once.

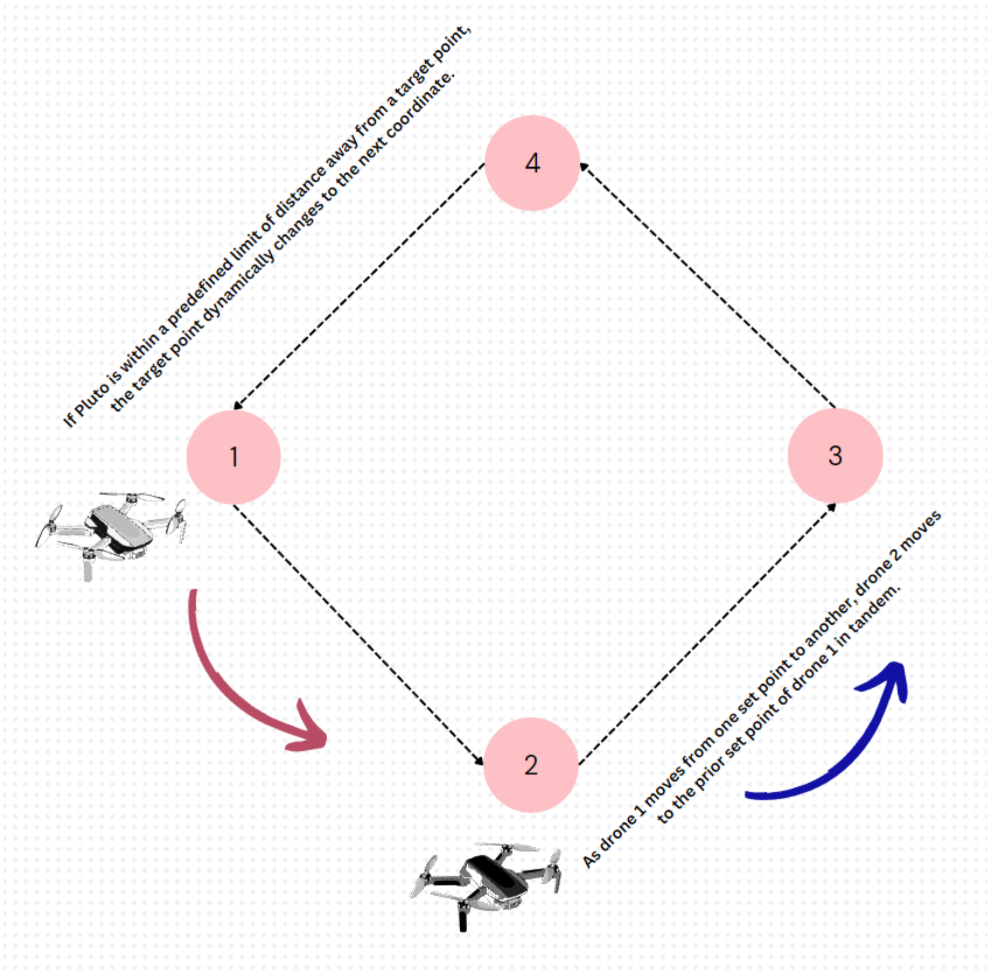

If Pluto is within a predefined limit of distance away from a target point, the target point dynamically changes to the next coordinate.
4
1
3
2
As drone 1 moves from one set point to another, drone 2 moves to the prior set point of drone 1 in tandem.

Figure 7: Rectangle Motion on a swarm