Pluto Drone:

Swarm Challenge

Team ID: 45

Task 1

Developing a Python wrapper

01 Our approach began with a straightforward socket algorithm as a means of transmitting and receiving data packets, but it lacked a two-way communication channel for real-time drone control.

02 Using Telnet as the communication protocol was the next obvious course of action for us. As a result, we were able to establish an IP connection and engage in real-time communication with the drone.

```
n(self):
th Telnet('192.168.4.1', 23) as tn:

  throttle = 1000
  yaw = 1500
  pitch = 1500
  roll = 1500
  armed = None

  print("ready to take commands")
  last_button = input("Enter command: ")
  try: ···

  except KeyboardInterrupt: ···
```

03 We began by delivering packets for the interface while detecting the keyboard commands while using nested loops, which resulted in delayed output. The program's temporal complexity increased as a result of the constant looping in and out; to address this, we turned to multiprocessing techniques.

04 We were able to read keyboard inputs and communicate with the drone using packets thanks to multiprocessing, which also allowed us to quickly switch to other functions like takeoff and landing.

05 To make sure that both applications share the same memory space and that the drone gets packets constantly without switching to default parameters for stability, threading was specifically used.

```
  last_button = None
> class keyboard(Thread): ···

> class drone(Thread): ···


  d1 = drone()
  obj2 = keyboard()
  obj2.start()
  d1.start()
```

06 The struct module is used to send data packets as byte objects. Struct.pack first prepares the values as needed before passing them as arguments.

```
def make_in(self, command: int, byte_arr: bytes):  # method to create an "IN" and "OUT" communication packets

    cmd = struct.pack(f"<cBB{len(byte_arr)}s", b'<',
                      len(byte_arr), command, byte_arr)

    crc = 0
    for c in cmd[1:]:
        crc ^= c
    crcb = bytes([crc])
    return b"$M" + cmd + crcb
```

07 Using the aforementioned function, motion inputs are sent, with the payload and command code changing for various instructions.

```
def msp_set_raw_rc(self, roll=1500, pitch=1500, throttle=1000, yaw=1500, aux1=2100, aux2=900, aux3=1500, aux4=1500):
    payload = struct.pack("<8H", roll, pitch, throttle, yaw, aux1, aux2, aux3, aux4)
    return drone.make_in(self, 0xc8, payload)

def arm(self): # method to arm the vehicle
    return drone.msp_set_raw_rc(self, throttle=1000, aux4=1500)


def disarm(self): # method to disarm the vehicle
    return drone.msp_set_raw_rc(self, aux4=900)

def takeoff(self):
    return drone.make_in(self, 0xd9, struct.pack("<H", 1))
```

08 Empty payloads are supplied first in order to obtain sensor inputs, enabling us to obtain data packets that must be decoded in order to obtain useful information.

```
    def msp_altitude(self):
        payload = bytearray(6)
        return drone.make_out(0x6d, payload)
```

09 Lists of this data are received and can be shown on the terminal or, in our instance, appended to the log files for subsequent analysis of the responses.

```
tn.write(self.msp_attitude())
att_data_pack = tn.read_eager()
att_data = att_data_pack.split(b'$M')

try:
    att_res = struct.unpack('<BBBBBB', att_data[1])
except:
    pass

try:
    with open("attitude_data.txt", "a+") as f:
        f.write(str(att_res)+"\n")
except:
    pass
```

To sum up, our Python wrapper communicates with the drone via the telnet protocol and accepts keyboard inputs. The github Readme file or by reading the code's comments will provide access to the comprehensive instructions. Additionally, by transmitting empty payloads, our wrapper enables the user to get sensor data from the drone, and this data is accessible via the associated log files.

Task 2

Objective 01:

Hovering the Pluto drone at a particular height using ArUco Tag

01 The fundamental concept is to estimate the drone's 3D pose using the information from the Aruco marker and then incorporate the input into our control loop.

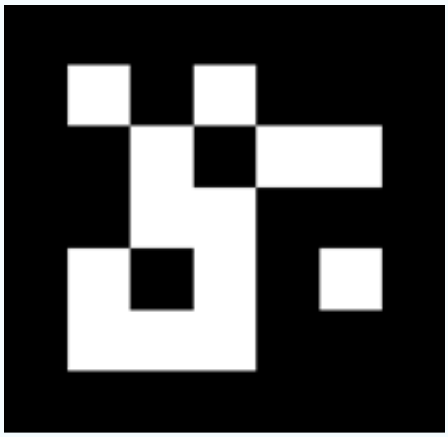
02 For this, a camera was positioned at a height of 3 metres on the ceiling. 6 cm long ArUco markers were created and applied to the drone, which was positioned in the camera's field of vision. We first used the 7X7 ArUco dictionary for appropriate detection before moving on to height and posture estimation. However, it had a limited field of view when utilising an ordinary RGB camera.

03 We experimented with numerous pre-processing methods, including median blur and thresholding with precisely calibrated parameters. However, the RGB camera's FPS limitation made detection at high speeds challenging.

04 The next logical step was to experiment with various ArUco dictionaries. For a 5X5 marker of the same length, this produced the best results for us. We later modified our camera for accurate detection, and using Intel's RealSense depth camera, we were able to obtain all needed detection data.

05 It should be noted that we are only using the RGB values from the camera and not the depth information because using the depth camera resulted in an inaccuracy of about 4 cm, whilst using the basic ArUco distance detection resulted in a smaller mistake. Using RGB frames directly, without any pre-processing, offered us the best detection even at high speeds due to better frame rate because RealSense's quality was considerably superior.

7x7 ArUco



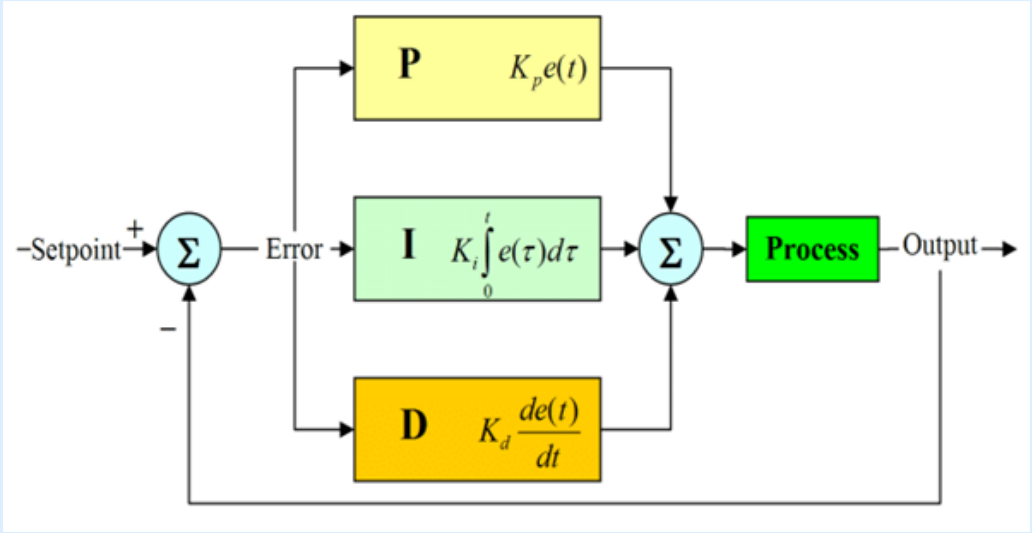
5x5 ArUco

```
18      super(height,self).__init__()
19      height.aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_ARUCO_ORIGINAL)
20      height.parameters = cv2.aruco.DetectorParameters()
21      height.parameters.cornerRefinementMethod=cv2.aruco.CORNER_REFINE_SUBPIX
22
23      height.markerLength = 0.07
24
25      height.focal_length=3.6
26      height.width,height.length=(1080,720)
27
```

ArUco detection code

06 The next stage was to use feedback loops to control the drone, and numerous approaches were taken into consideration for this. The initial step was to begin using the suggested PID control techniques.

07 The throttle, roll, pitch, and yaw are the four components of drone motion that our technology implements in the same way. Functions that add Kp, Kd, and Ki as vectors along with their respective error terms to the equilibrium vales, or 1500, were defined.



Control Loop Diagram

The feedback for the control loop is coming from ArUco Detection data.

08 As its name implies, the PID technique employs three error terms. Calculating the proportional error is as simple as comparing the required X, Y, Z, and yaw values to the existing values. The error vector is saved as "previous error" for the differential term after each loop iteration, and the first differential of the error is obtained by calculating and dividing their difference by the loop iteration time in the subsequent loop iteration.

09 The loop was given the necessary update frequency in order to prevent the difference from becoming stuck at zero.

10 Similar to how errors are added to the "error sum" with each iteration, integral terms are defined by multiplying by the time difference.

11 We reset the error term to prevent the integral from getting too big and creating issues with unwinding.

```
throttle_temp= 1500+ 5*(kp[0]*error[0]  +  kd[0]*((error[0]-previous_error[0])/dt)  +  ki[0]* (previous_error[0]+error[0])*dt)
roll_temp = 1500+ 7*(kp[1]*error[1]  +  kd[1]*((error[1]-previous_error[1])/dt)  +  ki[1]* (previous_error[1]+error[1])*dt)
pitch_temp = 1500+ 7*(kp[2]*error[2]  +  kd[2]*((error[2]-previous_error[2])/dt)  +  ki[2]* (previous_error[2]+error[2])*dt)
yaw_temp = 1500+ 7*(kp[3]*error[3]  +  kd[3]*((error[3]-previous_error[3])/dt)  +  ki[3]* (previous_error[3]+error[3])*dt)
```

The final functions thus look something like this

```
current_height= height_max-height.distance
error_height =  height_set - current_height
error_roll= - height.x
error_pitch= - height.y
error_yaw= - height.yaw
# print(height.x," ",height.y," ",height.yaw)

error=[height_set - current_height , -height.x , -height.y , -height.yaw]
# error = np.array(height_set - height.z, setpoint[i][0] - height.x ,setpoint[i][1] - height.y )
# error_yaw = yaw_setpoint - height.yaw
```

With the error calculations as follows

12 The following stage was to determine the constant values, Kp, Kd, and Ki, and tune them for the best control. Utilizing auto-tuning techniques like relay feedback and model predictive approaches is the most effective way to do this.

13 These automatic tuning techniques need precise placement, which can only be achieved utilising the information from the onboard sensors. It was logical to stick with manual tuning based on responses since our goal was to only use camera vision as feedback.

14 We increased Kp until a clear periodic motion could be seen, starting with Kp levels that are a P only control. The crucial constant, abbreviated Kc, was then noted, and the oscillation's period, abbreviated Tc, was determined.

15 Then, once the increment was smooth and steady, an integral term was introduced and raised.

16 Many of these experiments were conducted, and based on the results, the proportional, derivative, and iterative constant values for each of the four motion parameters — throttle, roll, pitch, and yaw — were established.

We further propose some automatic PID tunings that could have been used if more time was available:

Ziegler-Nichols tuning method

This is a classic tuning method that uses a step response test to determine the PID constant values.

Relay feedback method

This method uses a relay to oscillate the system, and the resulting oscillations are used to estimate the PID constant values.

Genetic algorithms

This is an optimization method that uses a population of candidate solutions and iteratively improves them based on a fitness function that measures the control performance.

Machine learning-based approaches

This involves using machine learning algorithms to determine the optimal PID constant values based on data collected from the system.

Task 2

Objective 02:

To move the drone in a rectangular fashion

01 In order to achieve this, a setpoint matrix was developed, with the four corners of a rectangle serving as the setpoints [1], [2], [3], and [4].

02 In order to determine the distance between the drone and the current setpoint, a radial error term is defined. The setpoint shifts from I to i+1 whenever this mistake falls below a specific threshold.

03 This method particularly ensures that PID always stays on even while at some setpoint.

04 At each shift the PID control ensures that the drone comes back to the setpoint and when it reaches there, the setpoint is shifted.

To sum up, our approach modifies the hovering code somewhat by initialising numerous setpoints rather than a single one. PID attempts to hover it at one while shifting the setpoint when it succeeds. Any desired drone trajectory can be achieved using a method similar to this one.

Task 3

Drone Swarm

Since the drone was previously in host mode and the device running the code connected to it in client mode, job 3 requires us to connect to and operate numerous Pluto devices simultaneously. We must use AT instructions to switch the Pluto drone's mode to client mode for Task 3.

Connect to the drone wifi and use the following command in a terminal to open a telnet connection:

- telnet 192.168.4.1 // drone wifi ip.
- Set the drone in both Station(STA) and Access Point Mode(AP) : +++AT MODE 3
- Set the SSID and password of your device hotspot in the pluto drone: +++AT STA SSID password
- Start your device(laptop) hotspot, and the drones will connect to it. Note the IP address assigned to it for running the pluto drone thread for multiple pluto drones.

Now that we have two independent drone class objects with the correct IPs, we can control them by initiating them on two separate threads. The objective is to have the second drone maintain a specific distance from the first one, and if the distance grows within a specific range, then supply the first drone's 3D pose as a setpoint goal for the second drone.

We need to employ two ArUcos, each with a unique ID associated with the two drones. To obtain the 3D poses of both drones, we employ the same ArUco detection technique as in Task 2. We then continue to assess the euclidean distance between the two Plutos and store it in certain variables so that we may compare it.

The second drone object's "follow bool" variable should be initialized. If the euclidean distance calculated above is more than the required value, the current pose of the first Pluto is used as the new setpoint goal of the second Pluto using the same method as in job 2.

Conclusion

Thus, we were successful in developing a Python wrapper for operating the Pluto drone. Telnet was utilised to let us connect with the drone. We were able to send and receive the required control packets for the drone after the link was established. The drone's height and pose were then estimated using ArUco Markers for our calculations. We accomplished this with the aid of a depth camera that was mounted to the ceiling of our work area. The PID algorithm, which was used to hover and steady the drone, was fed the pose data that had been gathered.

Following this, we successfully completed our second challenge, which was to move the drone in a 2x1 rectangle path.
The Python wrapper created in Task 1 can be used to implement the drone swarm task, which was our next goal. By using the OpenCV libraries and the ArUco markers that have been pasted on the first drone, we can control one drone while the second drone follows it.

Thus, all tasks assigned were completed.
The drone was used to test Tasks 1 and 2, and the tests were **successful**.