Control Physical Systems Report

Olajuwon Dele, Alexandr Goultiaev, Fanying Liu
Group Number 3

Contents

1 Introduction.

The crazyflie 2.X is a drone by Bitcraze which can be programmed to fly in a safe and controlled manner. The drone is firstly assembled and some simple self-tests are performed to make sure its integrity. To operate the drone, we need either to program the client to control the drone using code or control it with an app on phone or iPad through Bluetooth, as shown in Figure 1. Another way is to control it with the Joystick which was built especially for the project.

Basically, there are four tasks during the lab session:
1.1 Handheld Controller Challenge The drone must fly while being controlled by a microcontroller-based handheld device. The microcontroller is chosen by the students themselves and must be recognised and configured by the cfclient.
1.2 Autonomous Flight Challenge Students must find a way to programme the drone to fly a set pattern autonomously (i.e. without the handheld controller). The drone must be able to fly depending on the sensors available to them (i.e. the lighthouses and lighthouse deck, the flow deck and the z-ranger deck).
1.3 Advanced Controller Challenge For this challenge, the students are free to be creative and implement something novel to control the drone in lieu of the handheld controller. This is based on the understanding of how controllers and other devices can interface with the cfclient.
1.4 Swarm Challenge Groups who have successfully completed the Autonomous Flight Challenge can continue on to try the swarm challenge. Groups control two or more drones to fly in a set pattern.

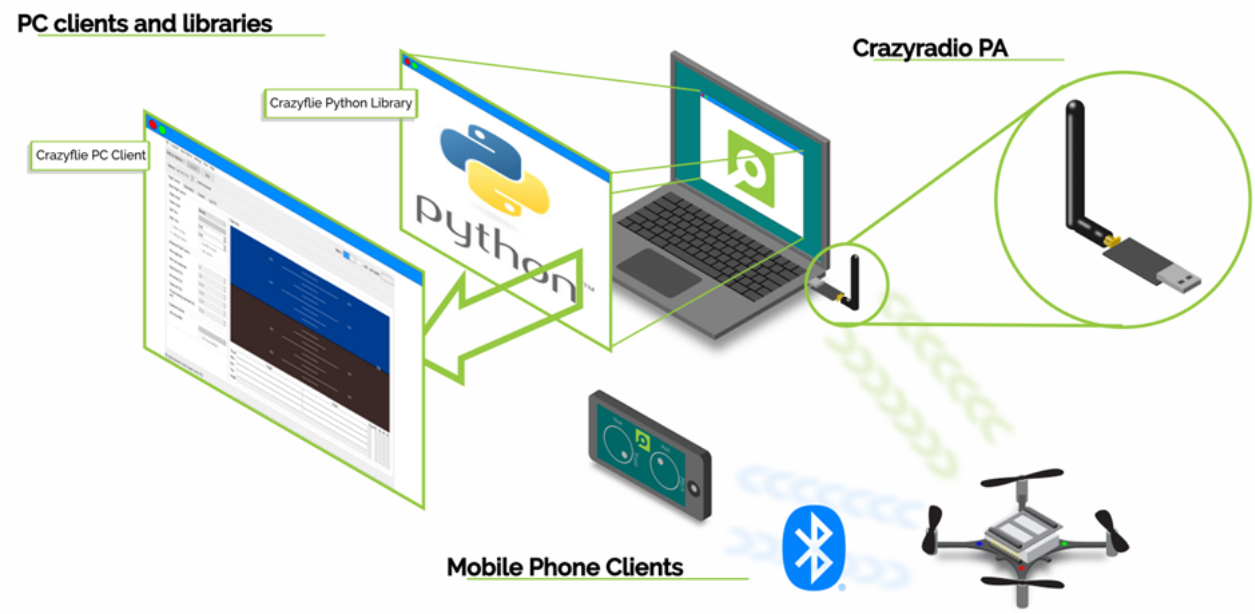
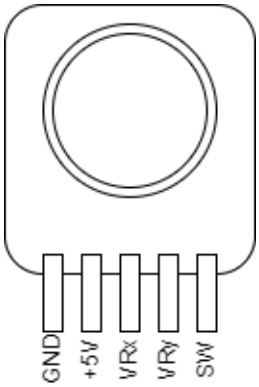


Figure 1: *Ways to control the crazyflie*

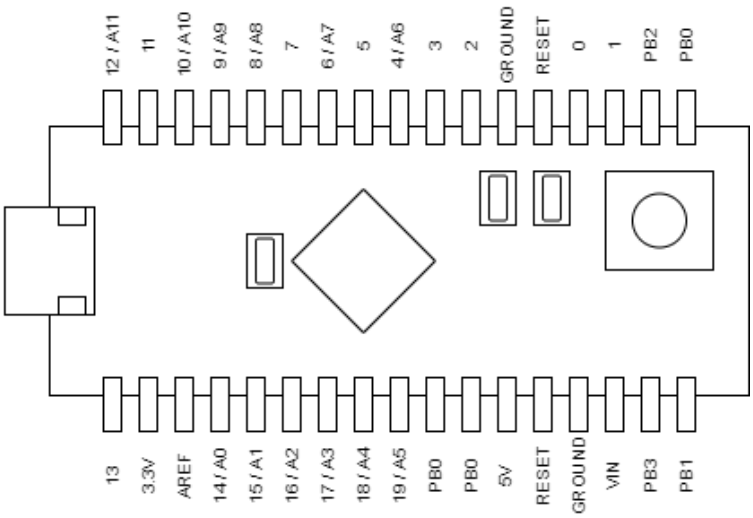
2 Handheld controller challenge.

Our first challenge completed was the handheld controller challenge. The controller needed to be based on a micro-controller, had to be handheld and be recognised by the computer as a controller so that the cfclient is able to recognise and configure it. After having seen some DIY console controllers made with Arduino's on the internet before we focused on Arduino's as the least-complex and better suited micro-controller for our controller. After looking into the cfclient and its controller configuration options we discovered that most console controllers are usable and we knew that we could focus on making a controller that emulates a console controller and that the cfclient would then naturally work as well. The console controller that was chosen as the target to emulate with our Arduino controller was an Xbox 360 console wired controller.

The first major choice was which Arduino micro-controller to use, there are a variety of Arduino boards but only a number of them have native USB support which we needed as we were doing a wired controller and didn't want to set up the communication between the controller and computer. This, and the availability of the Arduino micro board made us choose it as the micro-controller base of our controller. We then used two joysticks which gave us all the freedom of movement we needed for the drone, additionally each joystick can act as a button which we would later use to switch flying modes.

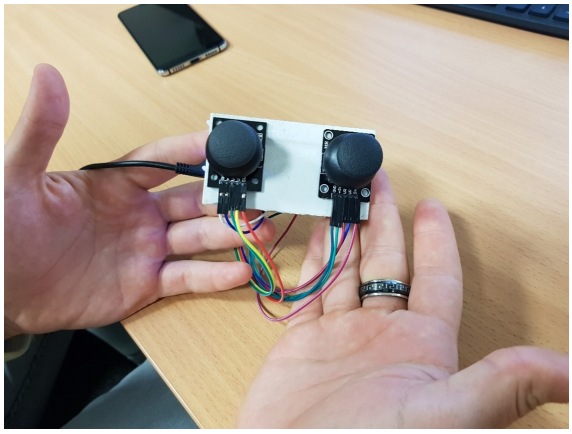


(a) Joystick pin-out.

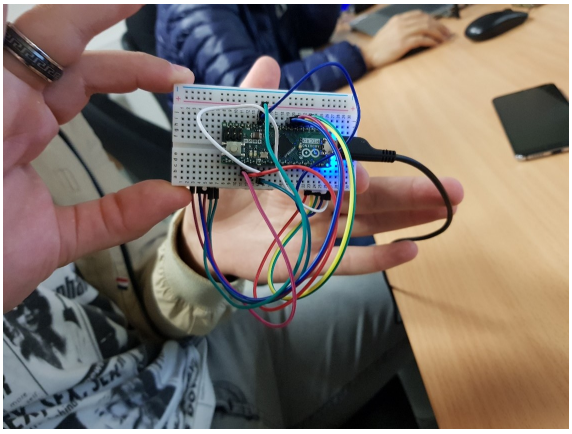


(b) Arduino Micro pin-out with digital and analog pins.

Figure 2: Pin-out diagrams.



(a) Front view.



(b) Back view.

Figure 3: Photos of our controller.

The drone we would be flying had three axis of freedom: pitch, roll and yaw. Additionally we would require a way of controlling thrust to the propellers which in total were four axis of control that we had to control with our controller. The two joysticks were sufficient for this task as each joystick has two axis of control (i.e. move the joystick up-down and left-right). The control layout we decided on is the one shown in Figure 4 and is based on the Xbox 360 console controller, this layout proved optimal as it allows good and accurate control of the drone while flying and allows all types of movements.

The Arduino was plugged into a breadboard and the joysticks were glued to the backside of the breadboard, making the breadboard itself the body of the controller 3. Then the joysticks were wired to their respective pins on the Arduino board using the analog pins for the RV_x and RV_y pins from the joysticks and digital pins for the SW pins from Figure 5.

As for the Arduino code we made use of a library made for this exact purpose of building console controller emulators with Arduino's called Xinput [1], this library allowed to easily code the joystick controls and make it recognisable as a controller by the computer.

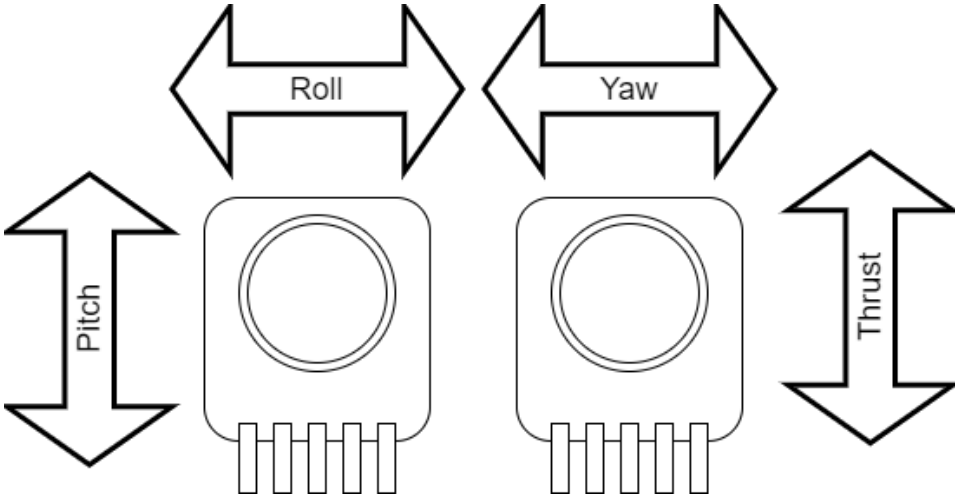


Figure 4: *Controller layout.*

```
#include <XInput.h>
// Setup
const boolean UseLeftJoystick = true; // enable left joystick
const boolean InvertLeftYAxis = false; // use inverted left joy Y
const boolean UseRightJoystick = true; // enable right joystick
const boolean InvertRightYAxis = false; // use inverted right joy Y
const int ADC_Max = 1023; // 10 bit

// Joystick Pins
const int Pin_LeftJoyX = A0;
const int Pin_LeftJoyY = A1;
const int Pin_RightJoyX = A2;
const int Pin_RightJoyY = A3;
const int Pin_ButtonA = 0;
const int Pin_ButtonB = 1;


void setup() {
  pinMode(Pin_ButtonA, INPUT_PULLUP);
  pinMode(Pin_ButtonB, INPUT_PULLUP);
  XInput.setJoystickRange(0, ADC_Max); // Set joystick range to the ADC
  XInput.setAutoSend(false); // Wait for all controls before sending
  XInput.begin();
}

void loop() {
  // Read pin values and store in variables
  boolean buttonA = !digitalRead(Pin_ButtonA);
  boolean buttonB = !digitalRead(Pin_ButtonB);
  // Set XInput buttons
  XInput.setButton(BUTTON_A, buttonA);
  XInput.setButton(BUTTON_B, buttonB);
  // Set left joystick
  if (UseLeftJoystick == true) {
    int leftJoyX = analogRead(Pin_LeftJoyX);
    int leftJoyY = analogRead(Pin_LeftJoyY);
    boolean invert = !InvertLeftYAxis; //most generic joysticks are typically
        inverted
    XInput.setJoystickX(JOY_LEFT, leftJoyX);
```

```
    XInput.setJoystickY(JOY_LEFT, leftJoyY, invert);
  }
  // Set right joystick
  if (UseRightJoystick == true) {
    int rightJoyX = analogRead(Pin_RightJoyX);
    int rightJoyY = analogRead(Pin_RightJoyY);
    boolean invert = !InvertRightYAxis;
    XInput.setJoystickX(JOY_RIGHT, rightJoyX);
    XInput.setJoystickY(JOY_RIGHT, rightJoyY, invert);
  }
  XInput.send(); // Send control data to the computer
}
```

Listing 1: Arduino code for the controller.

3 Autonomous flight challenge.

For the autonomous flight challenge it was necessary that the Crazyflie drone was capable of flying and achieving various different positions and achieve the roll, pitch and yaw motions. For this to be possible, the drone must be equipped with the flow deck and the lighthouse deck and use a python code such that it runs autonomously. The flow deck would provide a system information of the location of the drone by measuring its movements relative to the ground. The lighthouse deck improves the accuracy of this location by sensing two base stations which are place opposite to one another in the diagonal corner of a xy plane in and xyz coordinate system. The position of the drone is measured autonomously using the function wait_for_position_estimator.

The drone utilizes the Motion Commander module of the cflib python library for better motion control and control of some of its parameters. The function move_linear_simple includes motions of going left, right, up, down, etc to show the drone implement its roll, pitch and yaw motions. The function run_sequence uses predefined positions and makes the drone fly to these positions.

```
sequence = [
    (0, 0, 0.7),
    (-0.7, 0, 0.7),
    (0, 0.1, 0.7),
    (0, 0, 0.2),
]

def move_linear_simple(scf):
    with MotionCommander(scf, default_height=DEFAULT_HEIGHT) as mc:
        time.sleep(1)
        mc.forward(0.1)
        time.sleep(1)
        mc.turn_left(180)
        time.sleep(1)
        mc.forward(0.1)
        time.sleep(1)
        mc.turn_right(360)
        time.sleep(1)
        mc.up(0.2)
        time.sleep(1)
        mc.down(0.3)
        time.sleep(1)

def wait_for_position_estimator(scf):
```

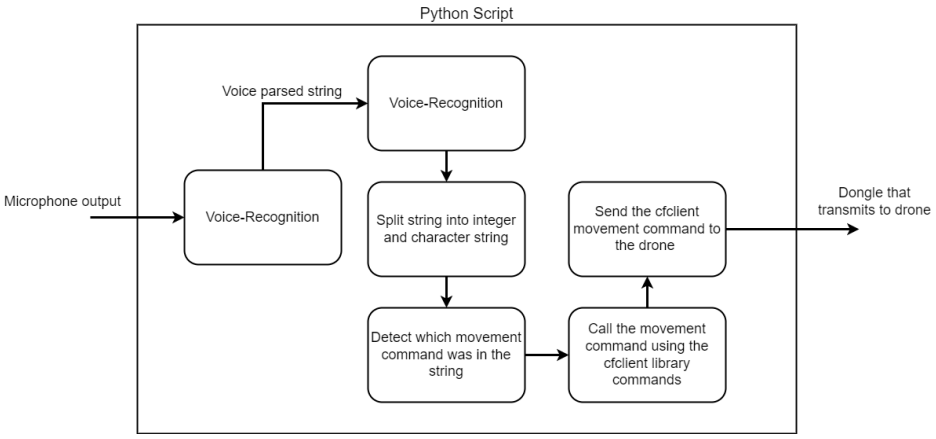
```
    print('Waiting for estimator to find position...')

    log_config = LogConfig(name='Kalman Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

    threshold = 0.001

    with SyncLogger(scf, log_config) as logger:
        for log_entry in logger:
            data = log_entry[1]

            var_x_history.append(data['kalman.varPX'])
            var_x_history.pop(0)
            var_y_history.append(data['kalman.varPY'])
            var_y_history.pop(0)
            var_z_history.append(data['kalman.varPZ'])
            var_z_history.pop(0)

            min_x = min(var_x_history)
            max_x = max(var_x_history)
            min_y = min(var_y_history)
            max_y = max(var_y_history)
            min_z = min(var_z_history)
            max_z = max(var_z_history)

            # print("{} {} {}".
            #       format(max_x - min_x, max_y - min_y, max_z - min_z))

            if (max_x - min_x) < threshold and (
                    max_y - min_y) < threshold and (
                    max_z - min_z) < threshold:
                break


def run_sequence(scf, sequence, base_x, base_y, base_z, yaw):
    cf = scf.cf

    for position in sequence:
        print('Setting position {}'.format(position))

        x = position[0] + base_x
        y = position[1] + base_y
        z = position[2] + base_z

        for i in range(50):
            cf.commander.send_position_setpoint(x, y, z, yaw)
            time.sleep(0.1)

    cf.commander.send_stop_setpoint()


if __name__ == '__main__':
```

```
    cflib.crtp.init_drivers()

    # Set these to the position and yaw based on how your Crazyflie is placed
    # on the floor
    initial_x = 0.0
    initial_y = 0.0
    initial_z = 0.0
    initial_yaw = 90 # In degrees
    # 0: positive X direction
    # 90: positive Y direction
    # 180: negative X direction
    # 270: negative Y direction

    with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
        set_initial_position(scf, initial_x, initial_y, initial_z, initial_yaw)
        reset_estimator(scf)
        run_sequence(scf, sequence,
                     initial_x, initial_y, initial_z, initial_yaw)
}
```

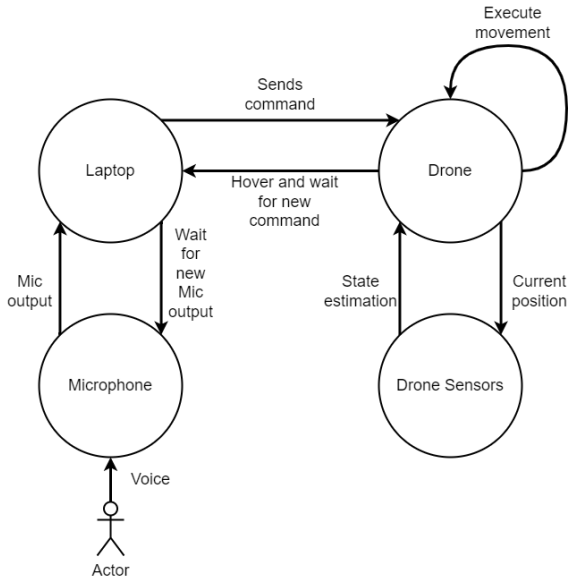
Listing 2: Python Autonomous Motion Code

4 Advanced controller challenge.

For the advanced controller challenge we had more creative freedom to design our controller. Our idea consisted of a voice controlled drone, therefore our controller is a python script that would control the drone by using voice-recognition to understand our voice commands and then interface with the cfclient library in python and finally relay the commands to the drone via the dongle.



(a) Pipeline of advanced controller.



(b) State diagram of advanced controller.

Figure 5: Visual representations of advance controller operation.

For the voice recognition part of the script a Python package called SpeechRecognition [2] and PyAudio [3] was installed. Figure 5a shows a diagram of the operation pipeline of our voice controller. The first task was to parse what commands we say into a string, we decided to only give out commands in a specific format: $\{action\ we\ want\ (characterstring) + by\ how\ much\ (integer)\}$ for example if we wanted to move forward by half a meter we would say: $\{forward + 50\}$ where the integer number is centimeters when the command is forward, back, left, right, up or down; and degrees when the command is turn left or right.

The parsed sting would then be split into its character string (directional command) and its integer part (distance or degrees) and a set of if conditions would check if the parsed character string coincided with any of the allowed commands that we hard-coded with them being: forward, back, up, down, land, left, right and turn. Then that if statement would call a function in the motion commander using the integer value as input to the function call.

5 Swarm challenge.

For the swarm challenge we had two drones work in a synchronous manner to make both a circle and a figure 8. These drones were capable of operating in a synchronous manner using the Swarm module in the cflib python library. The drones had their own respective heights such that they don't clash during the synchronous motion. The figure 8 trajectory was achieved by mapping the drones positions after a 4 sec interval of time, the heights remain constant while the x and y positions are equated using the equations below which obtains the trajectory displayed in Figure 6.

$$x = 0.4 * cos(t);$$
$$y = 0.4 * sin(2 * t)/2;$$

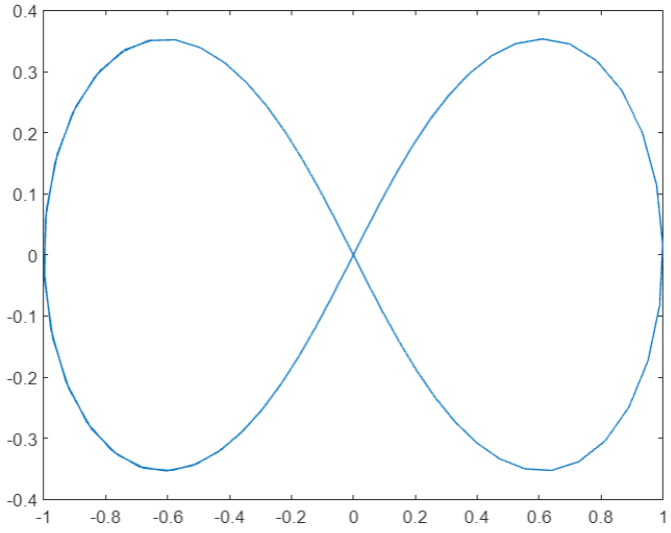


Figure 6: *Figure 8 Trajectory*

After the figure 8 trajectory, the circle trajectory enforeced. This is achieved by dividing 360 by the time of the circle trajectory interval and using this number as the yaw angle velocity. This is so the yaw angle itself will turn hit an yaw angle of 360 degrees after each interval.

```
def run_sequence(scf, params):
    print("flying")
    cf = scf.cf
    # Number of setpoints sent per second
    fs = 4
    fsi = 1.0 / fs
    # Compensation for unknown error :-(
    comp = 1.3
    # Base altitude in meters
    base = 0.5
    d = params['d']
    z = params['z']
    poshold(cf, 2, base)

    ramp = fs * 2
    for r in range(ramp):
        cf.commander.send_hover_setpoint(0, 0, 0, base + r * (z + base) / ramp)
        time.sleep(fsi)
```

```
poshold(cf, 2, z)

for _ in range(1):

    # The time for one revolution
    circle_time = 20

    for t in range(circle_time):
        print(t)
        #Figure 8 trajectory
        scale = (2 / (3 - math.cos(2*t)))
        x = 0.4 * scale * math.cos(t)
        y = 0.4 * scale * math.sin(2*t) / 2
        cf.commander.send_position_setpoint( x, y, z, 1)
        print(x)
        print(y)
        time.sleep(1)

    #Circle trajectory
    circle_time = 40
    for t in range(circle_time):
        cf.commander.send_hover_setpoint(d * comp * math.pi / 8,
                                         0, 360.0 /8 , z)
        time.sleep(fsi)

poshold(cf, 2, z)
for r in range(ramp):
    cf.commander.send_hover_setpoint(0, 0, 0,
                                base + (ramp - r) * (z - base) / ramp)
    time.sleep(fsi)
poshold(cf, 1, base)
cf.commander.send_stop_setpoint()
```

Listing 3: Swarm Sequence Snippet

6 Conclusion.

After a semester of unremitting efforts, the four challenges introduced in the beginning of the report were well addressed.

6.1 Handheld Controller Challenge: We choose an Arduino micro board with two joysticks, each joystick has two degrees of freedom so it there were four control channels in total: thrust, pitch, yaw and roll. The Arduino was plugged into a breadboard and the joysticks were glued to the backside of the bread-board and wired to their respective pins on the Arduino board using the analogy pins.

6.2 Autonomous Flight Challenge: The autonomous flight as well as advanced controller and warm flight were achieved by the python scripts. Firstly, import the related modulars from the Crazyflie python library, then define the function that includes the intended sequential motions of the drone. Finally call this function in the main programme, the drone will perform the pre-defined motions.

6.3 Advanced Controller Challenge: What we do is to introduce a voice-recognition modular to transfer our voice commands into the python code. Two voice recognition packages were installed and our voice commands were parsed into a string, and then split into a directional command and a number so that they can be integrated into the python commands.

6.4 Swarm Challenge: Two drones were controlled in our swarm demonstration. The lighthouse deck was used to avoid the accumulative errors, and the `send_hover_setpoint` command were used to define the trajectory of the drones. Basically, the two drones took off, then drawn a shape of “8” and finally draw a

circle with waving height.

There are some things that can be improved, especially the success rate of the speech recognition needs to be improved. Some more robust speech recognition package which can recognise less accurate pronunciation should be applied. And sometimes unexpected error appears when the script failed to recognise the command, the code still needs to be refined to handle such unexpected situations.

References

[1] Dave Madison. Arduinoxinput. https://github.com/dmadison/ArduinoXInput, 2019.

[2] Anthony Zhang. Speech recognition. https://github.com/Uberi/speech_recognition#readme, 2017.

[3] Hubert. Pyaudio. https://github.com/jleb/pyaudio, 2015.