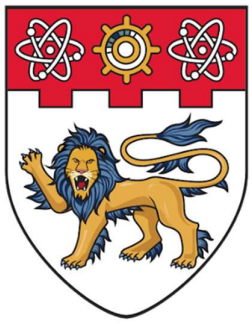
Project No. A1191-201



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# UAV Swarm Control Using Low Cost Commercially Off-the-Shelf (COTS) Drone


**Submitted by: Soh Chia Ning**

**Matriculation Number: U1721521K**

**Supervisor: Prof Xie Lihua**
**Co-supervisor: Shenghai Yuan**

**Examiner: A/P Mohammed Yakoob Siyal**

**School of Electrical & Electronic Engineering**


**A Final Year Project Report Submitted in Partial Fulfilment of the Requirements of the Degree of Bachelor of Engineering**

**2021**

# Table of Contents

# Abstract

Unmanned aerial vehicle (UAV) swarm missions are commonly conducted using custom-made drones due to their capabilities. This project proposes equipping multiple low cost commercially off-the-shelf (COTS) drones with the ability to be simultaneously controlled and applying it to UAV swarm missions in striving to achieve a lower cost alternative to custom-made drones. The COTS drone model used in this project is the DJI Tello EDU.

This report covers how control of the COTS drones is obtained using a mutual access point with a central device, and how port forwarding was used to enable simultaneous control of the drones as well as to receive unique camera feedback from every individual drone at the same time. These capabilities are applied in various drone swarm operations such as visual Simultaneous Localisation and Mapping (SLAM), facade scan for building inspection, area scanning, and drone air performances. Flight tests were conducted, showing that COTS drones were able to perform SLAM within an indoor area, a small-scale facade scan and patrol, and formation flights applicable for performances.

# Chapter 1 – Introduction

## 1.1 Motivations

In current times, unmanned aerial vehicle (UAV) swarm missions typically require the use of custom-made drones. These custom-made drones, such as SKYMAGIC's modified drones for light shows [1], are programmed and designed specifically for swarm missions. However, the problem arising from this is high costs due to the difficulty in producing these drones and their capabilities in bulk. Thus, this project endeavours to develop a lower cost but feasible solution to UAV drone swarm.

## 1.2 Objectives and Scope

The primary equipment used for this project are low cost commercially off-the-shelf (COTS) drones. These are easily obtained in large numbers and are more economical. However, they are not specifically designed for performing drone swarm missions. Therefore, the project aims to develop swarm capabilities for low cost COTS drones for UAV swarm missions to be made more feasible and performed more easily.

In order to achieve this, the project objectives include gaining the ability to control multiple low-cost COTS drones simultaneously and applying it to perform UAV swarm missions.


***Figure 1: Tello EDU Drone***
***Source: Adapted from [2]***

The low-cost COTS drone used in the project is the DJI Tello EDU, as seen in Figure 1. It is a small, lightweight, and relatively inexpensive programmable drone, equipped with an onboard camera with 720p transmission. It features a Software Development Kit (SDK) with its own commands [3]. These commands are used for the drone to perform basic operations, such as taking off, landing, and flying.

Additionally, the Tello EDU has two different modes, both of which are explored in this project, outlined below:

## *Access Point (AP) Mode*

The drone acts as an access point and can form a one-to-one connection with a device (i.e. the project PC) to receive commands from the device and also relay its camera feedback to the device.

## *Station Mode*

The drone connects to an access point. This is not a one-to-one connection and multiple drones can connect to the same access point at the same time. This allows a central device

(i.e the project PC), connected to the same access point, to send commands to all drones simultaneously, enabling multiple drone control. However, camera feedback is not supported in this mode.

This report outlines the challenges and different ways drone swarm control was achieved with the DJI TelloEDU drone.

# Chapter 2 – Literature Review

## 2.1 Single Access Point and Offset

In this method, all drones are connected and controlled from a single access point. There will be one leader drone that accepts commands from the access point. In order to create a drone swarm, the positions of follower drones are offset from the leader drone by a predetermined parameter. Follower drones respond accordingly to the leader's movements to maintain formation [4].

## 2.2 Drones Connected to a Mutual Access Point and Given the Same Commands

In this method, all drones receive the exact same command from a common access point. This would lead to each drone performing commands simultaneously, allowing them to move in sync while maintaining formation and removing the need for leader and follower drones [5].

## 2.3 Micro Aerial Vehicles (MAVs) for Drone Swarm

This method highlights the use of micro aerial vehicles for swarms to increase agility, compactness in formations, and safety. They can also be equipped with a protective carbon fibre cage effective at over 4m/s and thereby allow them to continue their mission should they collide with other objects [6].

## 2.4 Autopilot System for MAVs

This system uses dual radios, a global positioning system, an inertial measurement unit, and sensors for temperature and humidity to enable the autopilot system to estimate its state, localisation, and wireless networking [7].

## 2.5 Determining the Position of a Subject Using a Reference Node

This method uses a virtual reference node – a known point within an indoor setting, coupled with inertial navigation, to determine a subject's position within an indoor setting with an error of x = 0.049m and y = 0.056m. This accuracy is comparable to the commercially available LinkTrack's UWB local positioning of 0.100m [8].

## 2.6 Multi-Camera Motion Tracking System

In this system, multiple motion capture cameras are used to track the position and angles of many dynamic objects accurately with low latency, which is useful for localisation tests [9][10].

# Chapter 3 – System Structure

To facilitate to control of multiple drones in both modes of connection, different system structures were created, each attuned to suit the respective connection mode.

## 3.1 Station Mode Control Structure

Drones in station mode can connect to a common access point to receive unique commands from the project PC that is also connected to the same access point. In this structure, all devices are connected to the 192.168.1 network domain as provided by the access point.

The structure of connection and IP addresses of the drones and project PC are shown in Figure 2.

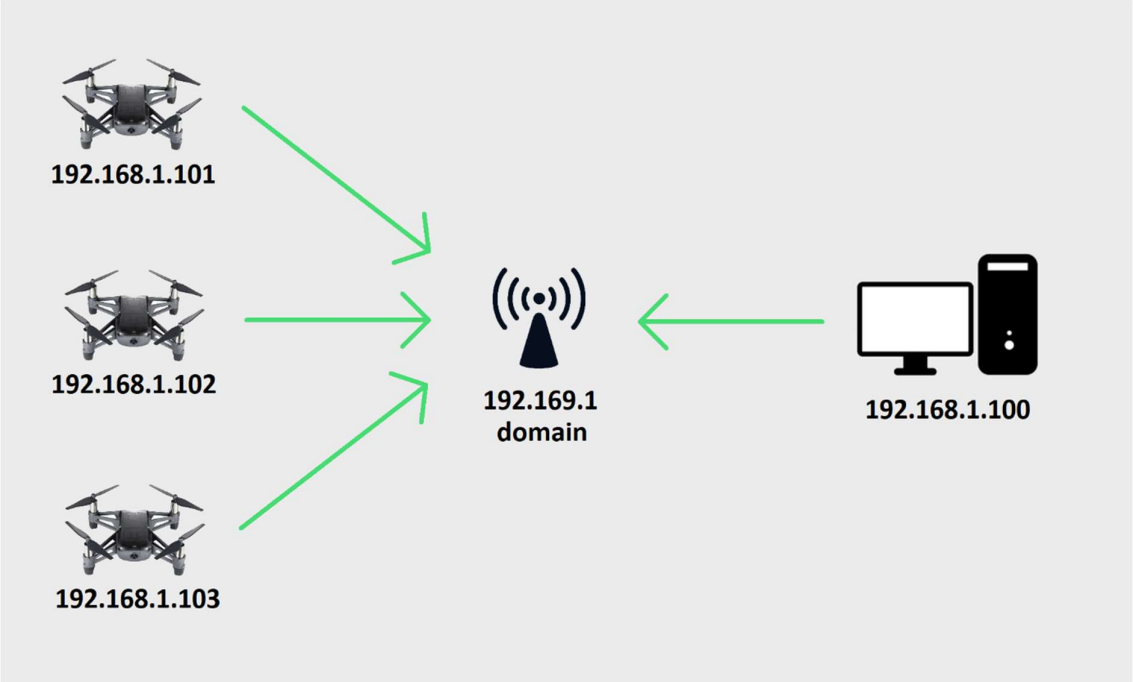



***Figure 2: Station Mode Control Structure***

While this structure allows for simultaneous control of multiple drones, it is incomplete for swarm control as it does not support camera feedback.

## 3.2 AP Mode Individual Control Structure

In the AP mode, drones could form a one-to-one connection with a device. In this project's case, the device to be connected to is the project PC.

Once a one-to-one connection is formed, the drone can receive commands directly from the project PC and relay its camera feedback. The structure of connection and IP addresses of the drone and project PC are shown in Figure 3.

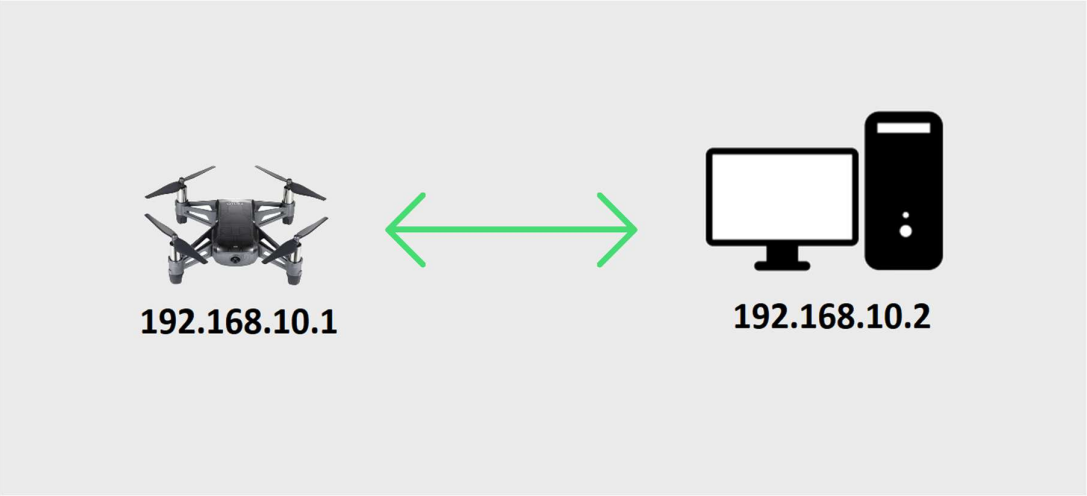



***Figure 3: AP Mode Individual Control Structure***

## 3.3 AP Mode Multiple Control Structure

Having camera feedback is essential in many drone swarm operations. Thus, it was important to come up with a way to achieve camera feedback from multiple drones.

Table 1 summarises the comparisons between the DJI TelloEDU's two connection modes – AP mode and Station mode:

| **Access Point (AP) Mode** | **Station Mode** |
|---|---|
| Able to receive commands from the project PC | Able to receive commands from the project PC |
| Able to give camera feedback to the project PC | Unable to give camera feedback to the project PC |
| Unable to have multiple drones controlled simultaneously | Able to have mutiple drones controlled simultaneously |

***Table 1: Comparisons between AP Mode and Station Mode***

As seen above, while AP mode supports camera feedback, it does not support the control of multiple drones simultaneously. This is because the DJI TelloEDU drones are manufactured with the same IP address when acting as an access point. This disables the drones (in AP mode) from being controlled individually for swarm control when connected to the same project PC, as the PC is unable to distinguish between the multiple drones because they have the same IP address. This is illustrated in Figure 4.

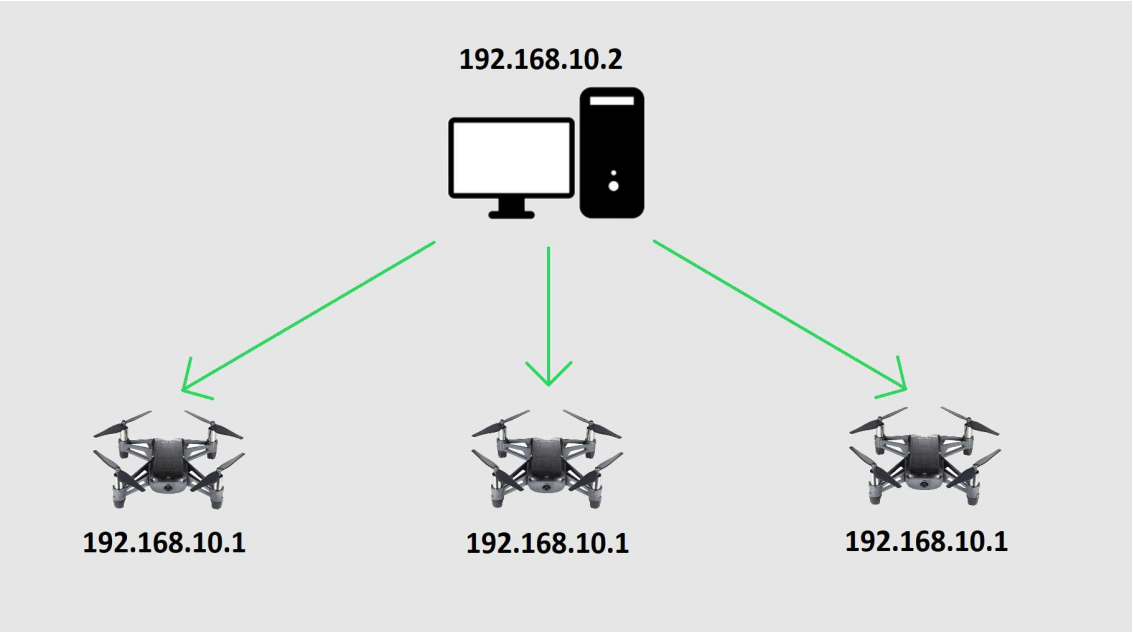



***Figure 4: Drones Having the Same IP Addresses***

Therefore, performing swarm control with this method is not possible.

On the other hand, the control of multiple drones is supported when the drones are in station mode. When connected to a common access point, the drones are given unique IP addresses by the access point. This means that the IP addresses of the drones will be distinct and distinguishable by the project PC that is also connected to the same access point as elaborated previously in section 3.1 and Figure 2.

As the project PC can uniquely identify the drones in station mode, it is able to send commands to the drones, which can then execute these commands. However, the DJI TelloEDU is not manufactured to support camera feedback in station mode.

Therefore, to achieve multiple drone control whilst having camera feedback, a network transmission model was created. This involved keeping the drones in AP mode, the only mode that supports camera feedback, while developing a way to make the drones distinguishable by the project PC.

### 3.3.1 Network Transmission Model

In this network transmission model, Raspberry Pis were used for portforwarding, where there is one raspberry Pi for each drone. The drones remain in the AP mode and form a one to one connection with a Raspberry Pi each in the 192.168.10 network domain. Meanwhile, the Raspberry Pis and project PC are connected to a common access point, where they share the 192.168.1 network domain with the access point. Table 2 shows the IP addresses of each device.

| Device | IP Address |
|---|---|
| Drone 1 (AP Mode) | 192.168.10.1 (manufactured IP Address) |
| Drone 2 (AP Mode) | 192.168.10.1 (manufactures IP Address) |
| Drone 3 (AP Mode) | 192.168.10.1 (manufactured IP Address) |
| Raspberry Pi 1 (connected to Drone 1) | 192.168.10.2 (domain with drone)<br>192.168.1.104 (domain with project PC) |
| Raspberry Pi 2 (connected to Drone 2) | 192.168.10.2 (domain with drone)<br>192.168.1.105 (domain with project PC) |
| Raspberry Pi 3 (connected to Drone 3) | 192.168.10.2 (domain with drone)<br>192.168.1.106 (domain with project PC) |
| Project PC | 192.168.1.100 |

***Table 2: IP Addresses of Devices in the Network Transmission Model***

Figure 5 shows the structure of connection using the network transmission model.

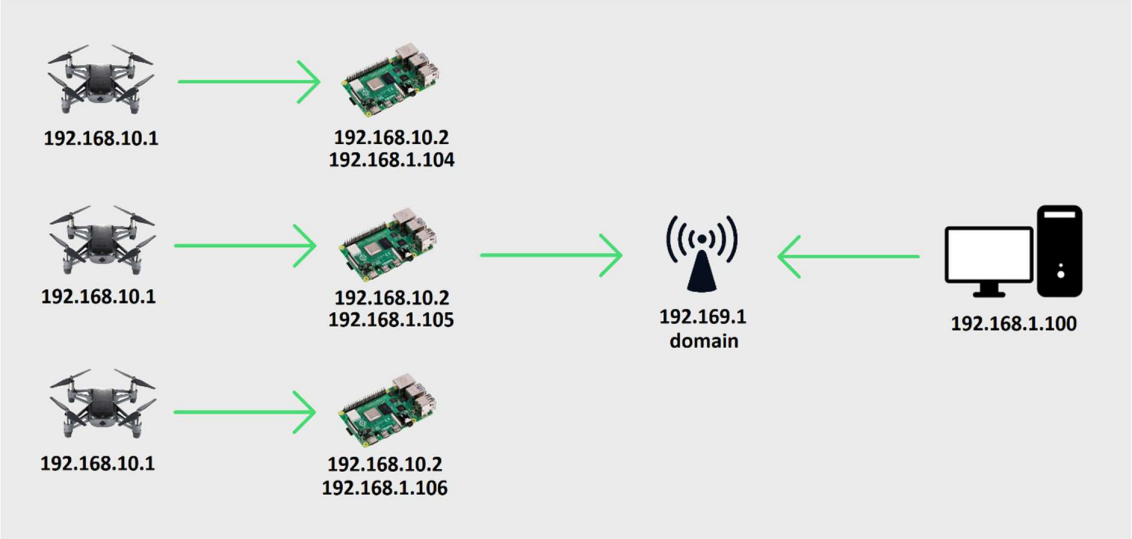



***Figure 5: AP Mode Multiple Control Structure Using Portforwarding***

As shown in table 2 and Figure 5, the Raspberry Pis are connected to two domains each, they act as the medium to facilitate communications between the project PC and drones in AP mode. This is done using iptables commands that are run on the Raspberry Pis before the communication begins. Once the commands are run on the Raspberry Pis, they become rules that the Raspberry Pi will enact upon. The following are four iptables commands use in this project:

1. FORWARD command

   The FORWARD command is used to allow the forwarding of data packets between the two network domains, 192.168.1 and 192.168.10. This is essential as the project PC and drones are in separate domains, and communications between them is impossible without data packets crossing over between the domains.

2. Nat table's PREROUTING command

   PREROUTING is a command that makes changes to a data packet before it enters the device. If the destination address of the data packet is the device's IP address, the data packet will enter and be processed by the device. If not, the data packet

will be passed on to its intended receipient. In this project, the PREROUTING command is used to redirect packets originally intended for the Raspberry Pis to either the project PC or the drones, facilitating the communications.

3. Nat table's POSTROUTING command and MASQUERADE command

   These two commands are used together. The POSTROUTING command is applied on data packets leaving the device, while the MASQUERADE command is used to change the source IP address of a data packet to the current device's IP address. When applied together on both domains of the Raspberry Pi, data packets that are leaving the Raspberry Pi will have the source IP address as the Raspberry Pi's IP address.

Using those commands, the following rules are established on each of the Raspberry Pis:

1. FORWARD

   The forwarding of packets between network domains 192.168.1 and 192.168.10 is accepted.

2. PREROUTING: changes the destination IP address from the project PC's to the drone's

   The source IP address of an incoming data packet is checked. If the source IP address matches the IP address of the project PC, the destination IP address of the data packet is changed to the IP address of the drone.

3. PREROUTING: changes the destination IP address from the drone's to the project PC's

   The source IP address of an incoming data packet is checked. If the source IP address matches the IP address of the drone, the destination IP address of the data packet is changed to the IP address of the project PC.

4. POSTROUTING and MASQUERADE

Packets leaving the Raspberry Pis will have the source IP address as that of the Raspberry Pi's IP address.

Before communications are initiated between the project PC and the drones in AP mode, the commands are first fun on the raspberry Pi's to establish the rules. Next, the project PC sends data packets to a Raspberry Pi, where the source IP address of the packets are 192.168.1.100 (project PC's IP Address), and the destination IP address of the packets are the receiving Raspberry Pi's IP Address.

Once the Raspberry Pi receives the data packet, it changes the destination IP address of the data packet to the drone's IP address, 192.168.10.1, based on the PREROUTING rule. Since the destination IP address is no longer the Raspberry Pi's IP address, the Raspberry Pi will pass on the data packet to the recipient, which is now the drone. Since forwarding of packets between the two network domains is accepted, the data packet can be sent to the drone. Before leaving the Raspberry Pi, the source IP address of the data packet is changed from the project PC's to the Raspberry Pi's. Thus, when the drone finally receives the data packet, it believes the sender is the Raspberry Pi.

Hence, the drone will send it's camera feedback to the Raspberry Pi – destination IP address of data packets sent by the drone will be the Raspberry Pi's IP address.

When the data packet from the drone is received by the Raspberry Pi, it changes the destination IP address of the data packet to the project PC's IP address, then changes the source IP address of the packet to its own IP address when passing on the packet to the project PC. This way, the project PC also believes the sender of the data packet is the Raspberry Pi.

Through this method, we can see that neither the drone nor the project PC are aware of each other's existences, yet they are able to communicate with one another. Moreover, the project

PC is also able to now control multiple drones in AP mode as it can distinguish between the IP addresses of the different Raspberry Pis when it previously could not.

### 3.3.2 Enabling Camera Feedback from Multiple Drones

Even when the project PC has gained the ability to distinguish between multiple Raspberry Pis and control multiple drones simultaneously, there is still one more step needed for achieving unique camera feedback from each drone

In addition to being manufactured with the same IP addresses in AP mode, the DJI TelloEDU are also pre-programmed to broadcast their camera feedback to the port 11111. Thus, when multiple drones relay their camera feedback to the project PC through portforwarding on the Raspberry Pis, the project PC receives all the different camera feedback on port 11111.

This poses a problem when the project PC plays the video feed from port 11111, as it is not possible to view the camera feedback separately, resulting in the video feed overlapping.

To resolve this, another iptables nat table PREROUTING command was used on the project PC. This command establishes a rule that redirects data packets received on port 11111 from a Raspberry Pi to a different port (e.g 11113). Table 3 and 4 illustrate an example of how multiple camera feedback are separated into different ports, enabling unique playback of each drones' video feed.

| Source IP address of Data Packet | Receiving Port of Data Packet |
|---|---|
| 192.168.1.104 | 11111 |
| 192.168.1.105 | 11111 |
| 192.168.1.106 | 11111 |

***Table 3: IP Addresses and Ports Before PREROUTING on the Project PC***

| Source IP address of Data Packet | Receiving Port of Data Packet |
|---|---|
| 192.168.1.104 | 11114 |
| 192.168.1.105 | 11116 |
| 192.168.1.106 | 11118 |

***Table 4: IP Addresses and Ports After PREROUTING on the Project PC***

# Chapter 4 – Implemented Approaches

## 4.1 Drone ORB-SLAM with Robot Operating System

This method successfully achieved getting a drone to perform an improved form of Simultaneous Localisation and Mapping (SLAM) – ORB-SLAM, where SLAM is a common operation performed by drone swarms.

ORB-SLAM is able to explore and create a 3D reconstruction of a wide variety of scenes, compute camera trajectories, perform global relocalisation, and close large loops in real-time, while automatically perform initialization from planar and non-planar scenes robustly. It is also capable of maintaining a compact map and enhancing tracking robustness through the use of a 'survival of the fittest' keyframe selection. This is achieved by having keyframes inserted at a really fast rate during explorations.

Robot Operating System (ROS) is an open-source robotics middleware/meta-operating system used for robotic software development which has tools, libraries and conventions intended to simplify the creation of complex robot behaviour [11]. It is able to be run on the Ubuntu OS [12].

Using an ROS package from an online source in [13] and making changes to cater to the project, the drones could be flown individually and perform the ORBSLAM once connected to the project PC. In this implementation, the drones were in AP mode with the control structure depicted in 3.2.

Figure 6 shows the progress of the ORBSLAM as it is being carried out. The green dots in the video feedback on the right provided by the drone represents features of the scene being picked up in real-time.


***Figure 6: Drone Camera During ORBSLAM***

The features are then mapped onto the graph as shown in Figure 7.

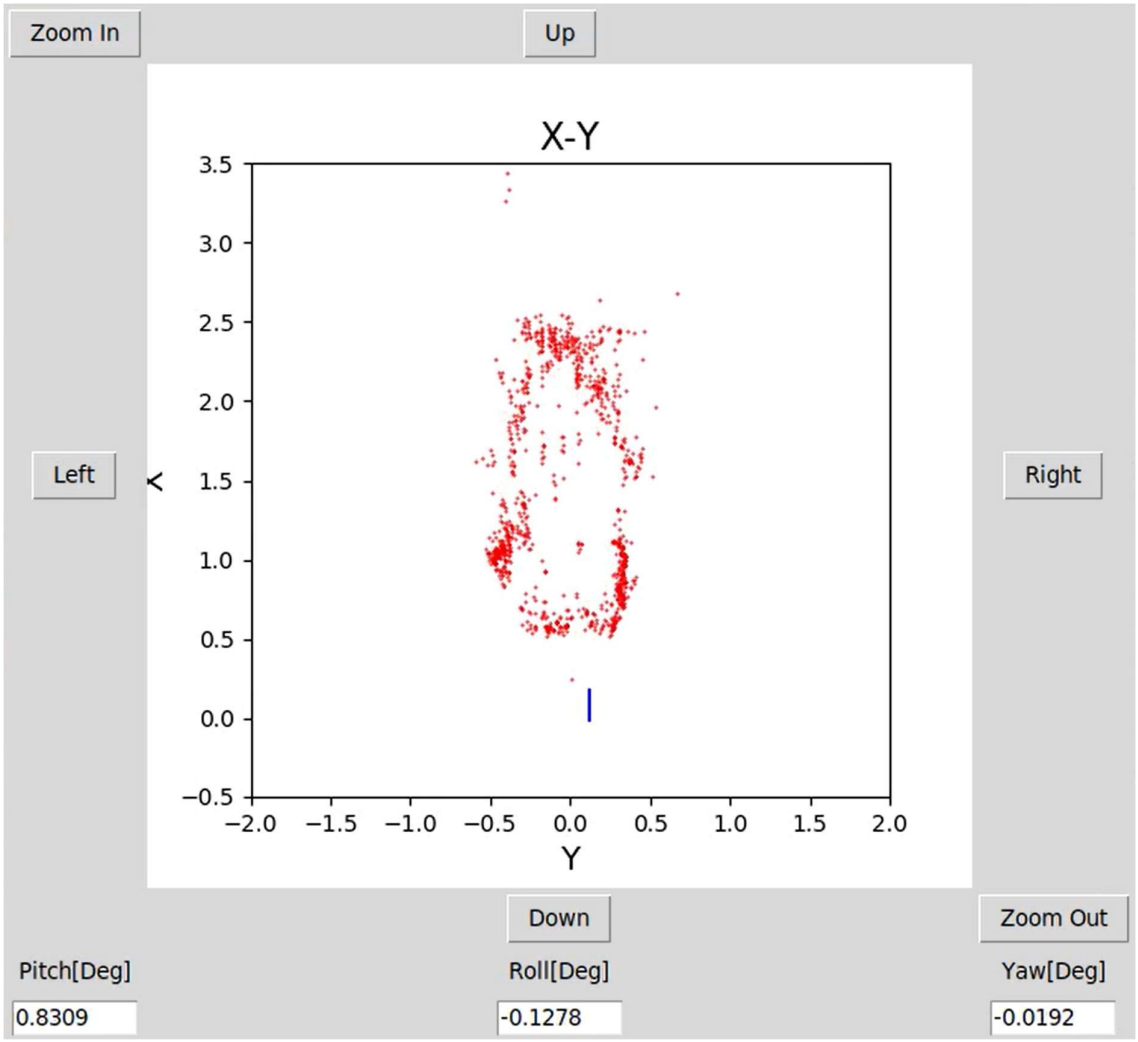



***Figure 7: Mapping of Drone's Camera Footage During ORBSLAM***

Figure 8 shows the user interface (UI) of the controller used to manage the drone's flight during the performance of ORBSLAM. A drone's flight path can be generated and controlled by creating a list of waypoints using x, y, and z values, as seen in the highlighted section in Figure 8.

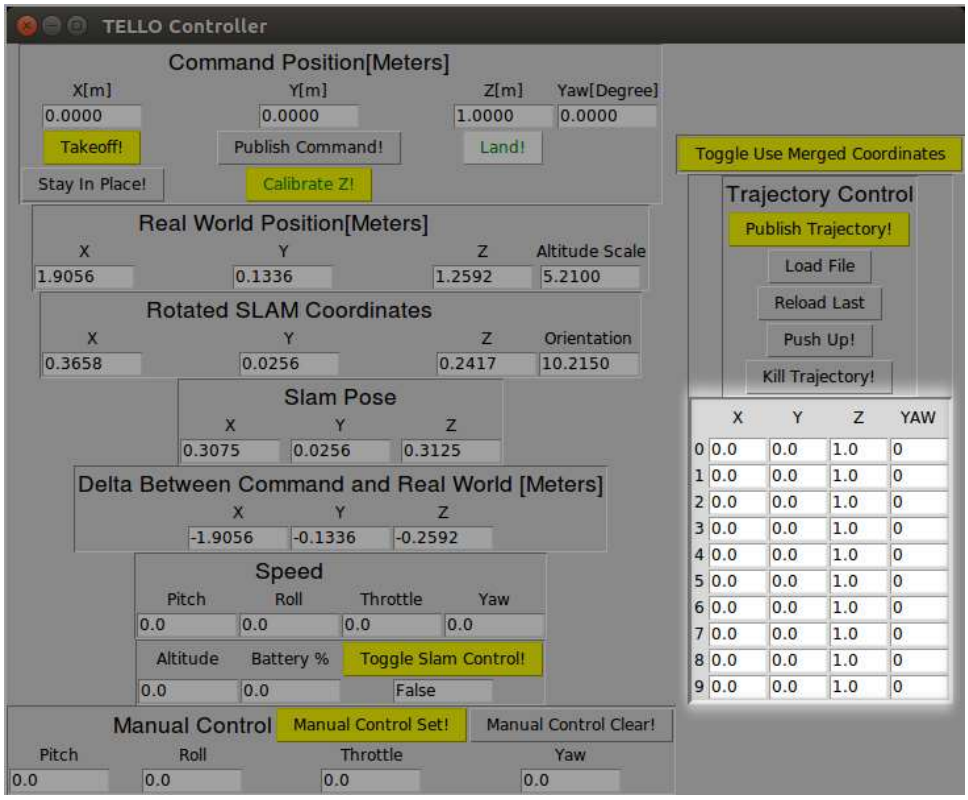



***Figure 8: Tello UI for Flight Path Generation***
***Source: Adapted from [13]***

Once the list of waypoints for a designed route have been published to the program, the user can then click on 'Toggle Slam Control' to get the drone to perform the route. The drone will then traverse from one waypoint to the next linearly. Through this method, the drones can fly around and map out a room.

## 4.2 Swarm Control in Station Mode for Air Performance

This section outlines the flight patterns created with the drones in station mode without camera feedback as depicted in the control structure in 3.1. Two different flight patterns were created to demonstrate the coordination and precisions of the drones.

### 4.2.1 ‘NTU’ Flight Pattern

In this flight pattern, three drones were used.

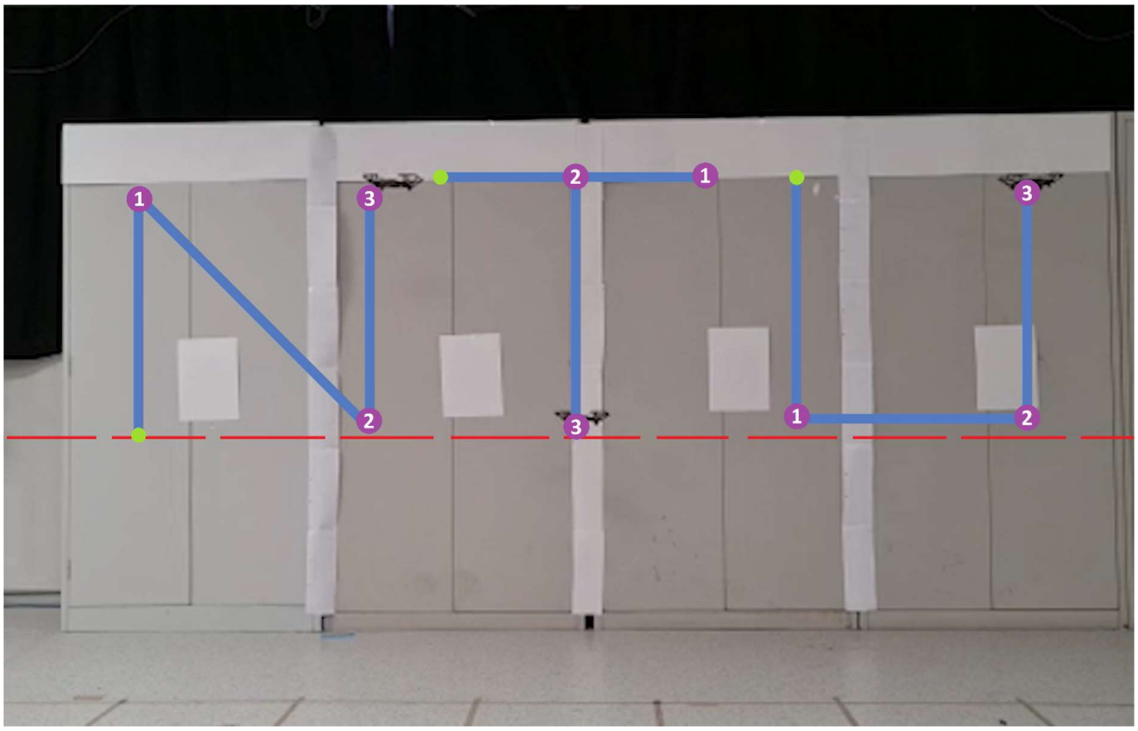



***Figure 9: Drones Spelling ‘NTU’***

Each of the drones would spell out a different letter mid-flight – ‘N’, ‘T’, and ‘U’ as seen in Figure 9, where drone 1 spells the letter ‘N’, drone 2 spells the letter ‘T’, and drone 3 spells the letter ‘U’. To achieve this, there are three following steps:

1. The drones take off and head to their respective starting positions.

   Marked by the green dot in Figure 9, the letters ‘T’ and ‘U’ have a higher starting position than the letter ‘N’. The ‘takeoff’ command brings drones to a common height, marked by the red dotted line in Figure 9. This means that while drone 1 will be in its correct starting position once all three drones have taken off, drone 2 and 3 will still have to rise further to reach their starting positions.

2. Once all 3 drones have arrived at their correct starting positions, they will rotate in a full circle in the clockwise direction to signal the start of them drawing the letters.

3. Following which, the drones will spell out their respective letters. This is done in the program by specifying the waypoints of the letters – each letter has 3 waypoints that the drone will fly linearly to in ascending order from its starting position. The way points are marked and numbered by the purple dots in Figure 9.

4. Once the drones have completed spelling their letters, they will once again rotate in a full circle in the clockwise direction to signal the end of the drawing of letters. Subsequently, all three drones will land.

### 4.2.2 Criss Cross Flight Pattern

In this flight pattern, two drones were used.

Throughout this flight, the drones draw a vertical triangle as shown in Figure 10 where the drones are awaiting takeoff – each green dot represents a vertex of the triangle and the drones traverse from vertex to vertex in the order of the alphabets displayed (i.e. A to C, then back to A before landing).

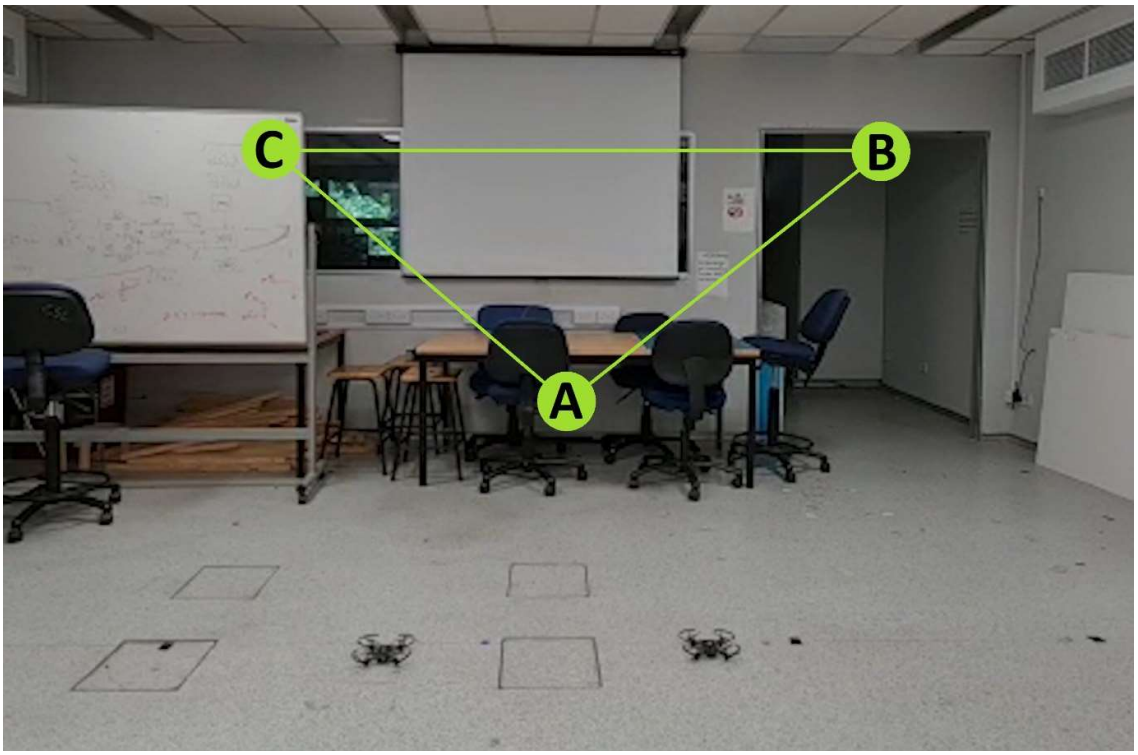



***Figure 10: Frontal View of Both Drone's Flight Route***

At each vertex, the drones will fly in a short criss cross pattern before moving to the next vertex, where they will move from a side-by-side formation to a front and back formation, or vice versa. Figure 11 shows the drones' movements at each vertex. The drone icons show the arriving position of the drones at each vertex. Following which, they carry out the movement shown by the blue lines before flying to the next vertex. Once they have completed the movement at the final vertex, they land.

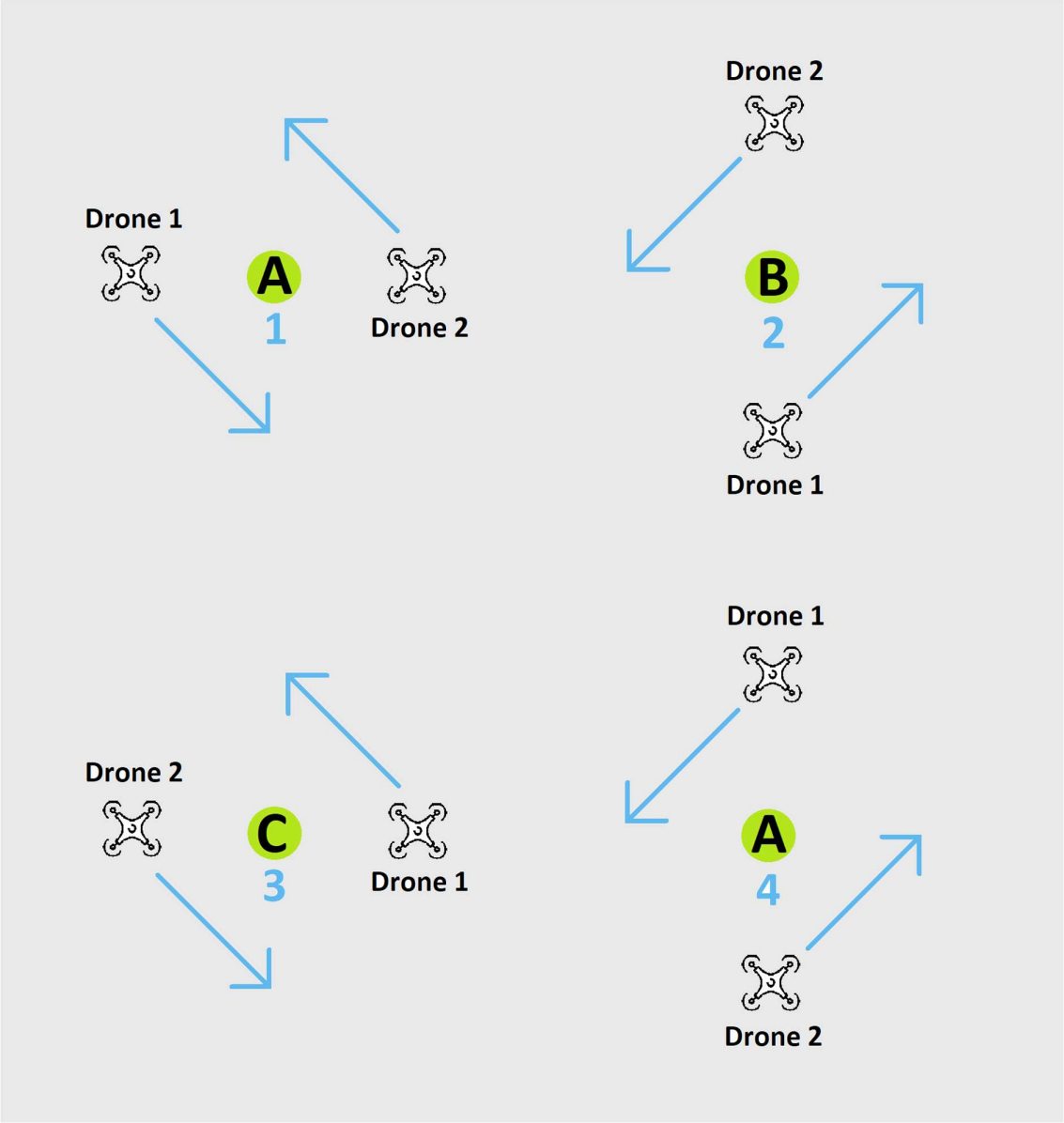



***Figure 11: Trajectory of Drones at Each Vertex***

As we can see, once the drones first arrive at vertex A, they move to a front and back formation from a side-by-side formation, where drone 1 is now directly in front of drone 2. Following which, they will ascend diagonally up to vertex B, where they will revert to a side-by-side formation, before flying to vertex C.

This pattern repeats itself – the drones move to a front and back formation at vertex C, then back to a side-by-side formation back at vertex A. Now back in their initial starting positions, both drones land, completing the flight pattern.

## 4.3 Swarm Control with Camera Feedback

In this section, two flight patterns were created, demonstrating the achievement of simultaneous control of drones while having camera feedback. The control structure of these flights are depicted in 3.3.

### 4.3.3 Facade Scan Flight Pattern

In facade scans, the surface of structures, usually buildings, are scrutinised.

Thus, this flight pattern was developed for the purpose of scanning the facade of buildings. Drones will start from the ground and work their way upwards in the following described manner:

1. Multiple drones take off from the ground, reaching a fixed height below the second floor of the building.
2. The drones fly to the right simultaneously, covering the entire horizontal length of the building's exterior surface. Once done, the drones ascend to the height of the next part of the surface to scan.
3. The drones then fly to the left simultaneously and cover the entire horizontal length of the building's exterior surface once again, before ascending to a new height.
4. Steps 2 to 3 repeat until the entire vertical surface of the building has been scanned.

Once the drones arrive at the height of the top of the building, the facade scan is completed and the drones will land. Figure 12 shows the trajectory of the flight when drones are performing the facade scan.

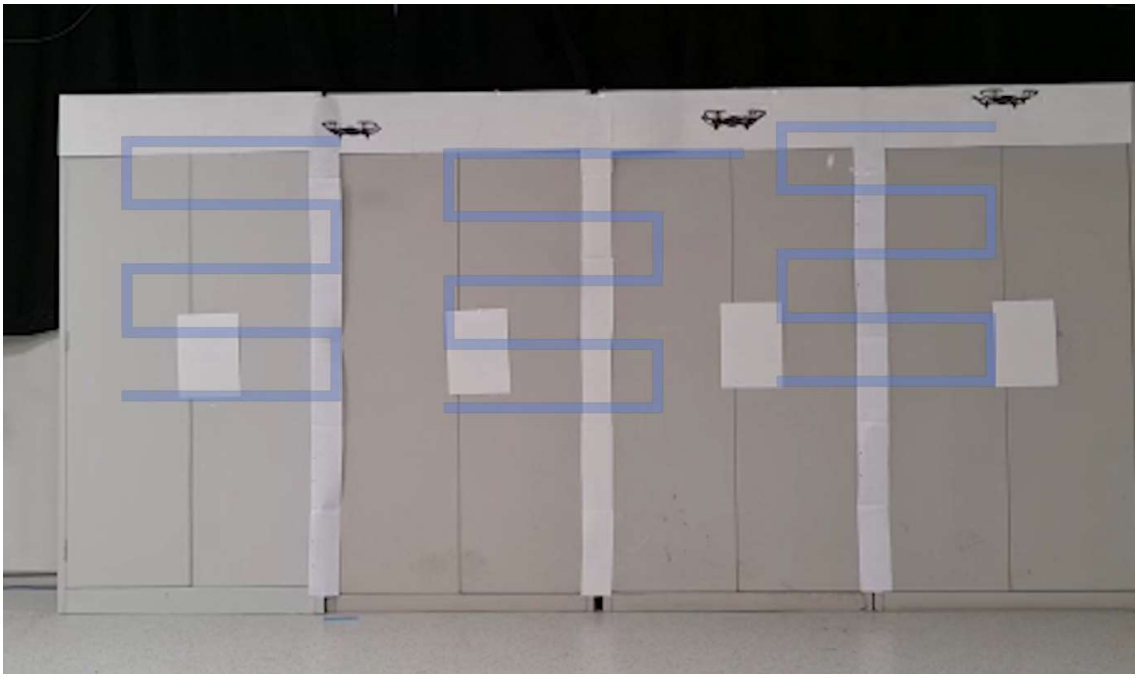


***Figure 12: Facade Scan Flight Pattern***

This method of flying can be useful in applications such as damage inspection of structures.

### 4.3.4 Area Scan/Patrol

In this flight pattern, the drones scan the area inside a laboratory. Figure 13 shows a plan view of the laboratory, where the grey areas are to be scanned.

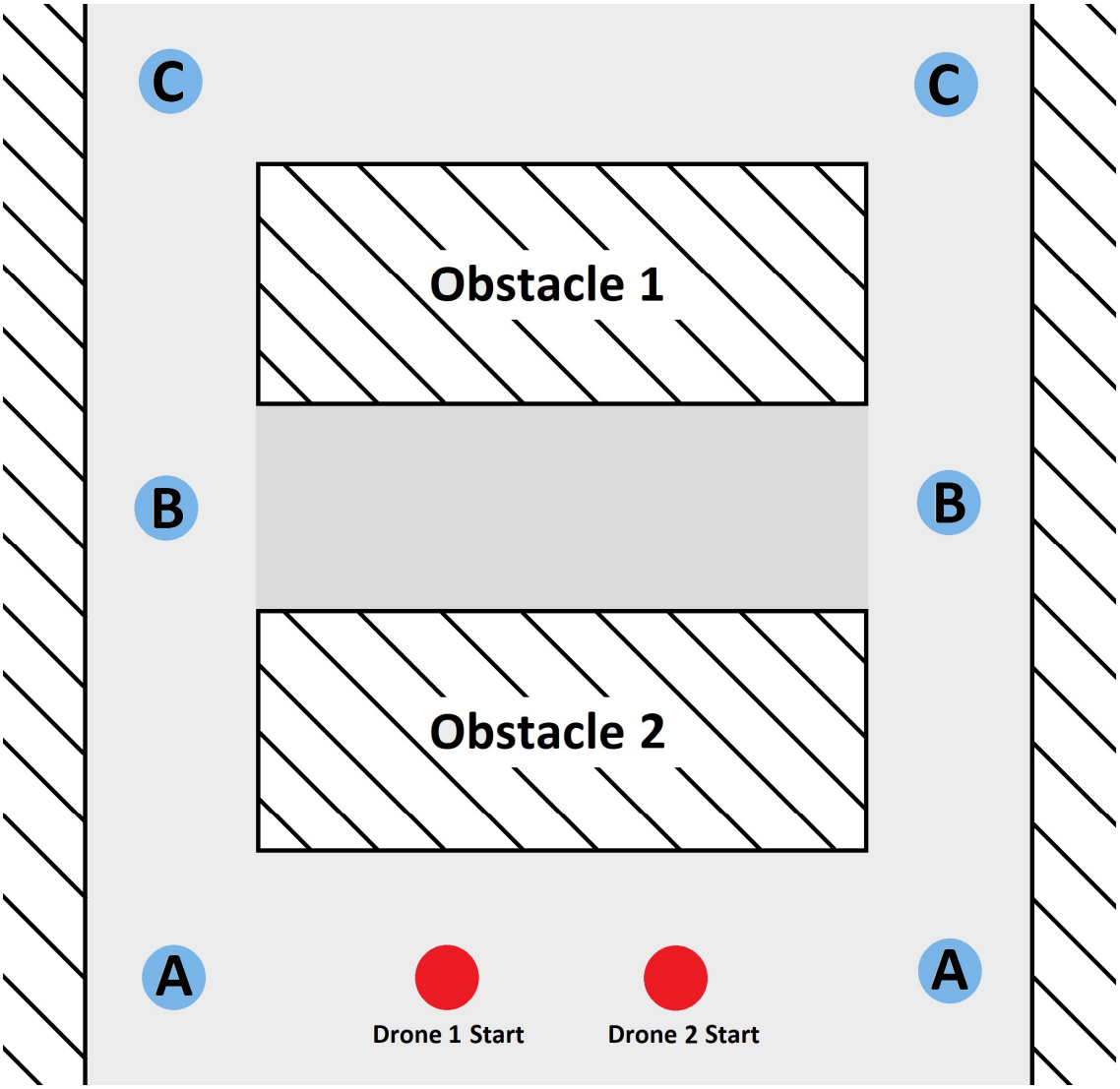



***Figure 13: Plan View of Area to be Scanned***

Two drones are used in the scanning of this area, and the routes the drones will be taking are shown in green. The starting point is shown by the red dots, where the drones will take off from. Blue dots along the routes represent points where the drones will stop to hover at and

rotate to view areas in greater detail. The following is a clearer explanation of the route, where drone 2 mirrors the flight pattern of drone 1:

1. Drone 1 takes off from the red dot emphasised in Figure 13.
2. Drone 1 then flies to blue dot A, scanning the area between blue dot A and the starting point.
3. Following which, Drone 1 turns to the right by 90 degrees and flies to blue dot B.
4. Once at blue dot B, the drone will rotate 90 degrees to the right again to view the area beween the two obstacles, marked by the darker grey colour.
5. Then, the drone rotates to the left by 90 degrees and head towards blue dot C. At this step, the drones would have finished scanning the entire area except between the two blue dot Cs.
6. Finally, Drone 1 rotates 90 degrees to the right to view the last unscanned area.

Once the entire area has been scanned, both drones will return to their starting points and land. Figure 14 shows screenshots of the two drones' camera feedback viewed on the project PC during a test flight. The positions of the two drones are shown in the figure and the blue cone represents the direction the drone's camera is facing.

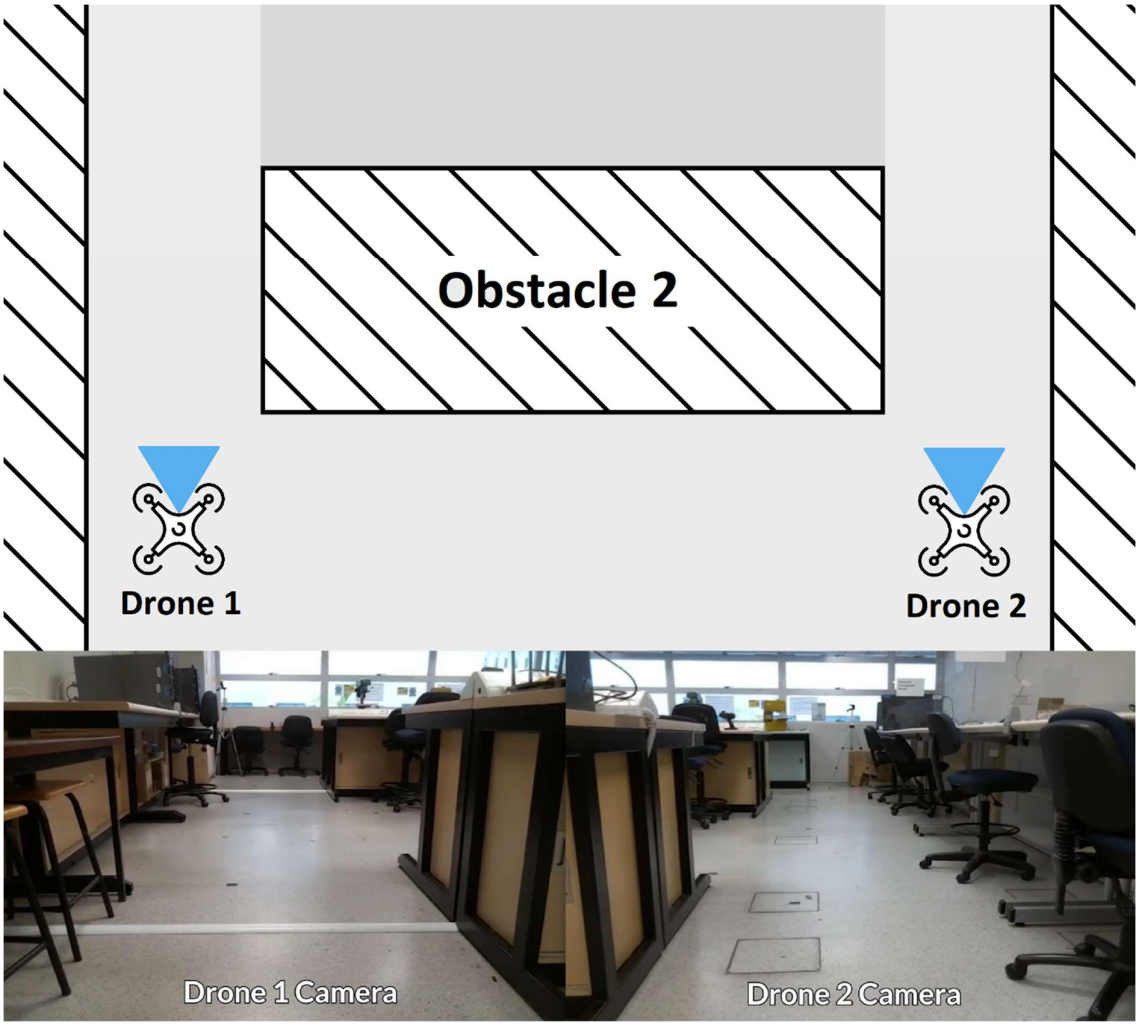



***Figure 14: Drones' Camera View During Patrol***

The flight pattern can be enhanced to accommodate for different areas and thus be applicable to a wide variety of area scans and patrols.

# Chapter 5 – Conclusion and Future Work

## 5.1 Conclusion

In summary, low cost COTS drones are a feasible alternative to the more expensive custom-made drones conventionally used in drown swarm operations.

This is achieved by being able to control multiple drones simultaneously and obtaining discrete camera feedback from each drone. Control of one DJI Tello EDU drone while performing ORB-SLAM was accomplished. Additionally, multiple drones (in AP mode) were controlled simultaneously by connecting them to a common access point with the project PC. Portforwarding and iptables commands enabled each of these drones to send their camera feedback via Raspberry Pis to the project PC and viewed separately. Drones in station mode could also perform swarm operations such as flights for drone performances, albeit without camera feedback.

Various tests were performed to apply these capabilities on facade scans, area scans, flight performances, and have shown that low cost COTS drones were able to perform such drone swarm operations, which could previously only be done with expensive, custom-made drones.

## 5.2 Future Work

### 5.2.1 SLAM with Multiple Drones

Currently, the ORB-SLAM functionality mentioned in 3.1 could only be carried out with a single drone.

In order to use multiple drones for ORB-SLAM, the network transmission model mentioned in 3.3.1 could be extended to this use and enable collaborative mapping operations between multiple drones.

This is achieved when at least two drones are able to perform the ORB-SLAM simultaneously, with each camera feedback relayed to the project PC separately.

### 5.2.2 Changing the IP Addresses of Drones in AP Mode

Each drone contains the same IP address from their manufacture.

This gave rise to issues preventing simultaneous control of all drones and obtaining discrete camera feedback from each drone when the drones are in AP mode and connected directly to the project PC.

Portforwarding was used to address this, but a more direct solution would be to change the IP addresses of each drone individually. However, as mentioned previously, the drones are manufactured with the same IP addresses while in AP mode. While attempts at changing the manufactured IP addresses were explored earlier in the project, it could not be achieved.

Further research of the issue may lead to a solution in the future.

# References


[1] SKYMAGIC, “Drone Light Show Technology,” *SKYMAGIC*. [Online]. Available: https://skymagic.show/about/technology/. [Accessed: 09-Nov-2020].

[2] The Drone Shop, “Ryze Tech Tello EDU,” *The Drone Shop*. [Online]. Available: https://thedroneshop.co/ryze-tech-tello-edu.html. [Accessed: 09-Nov-2020].

[3] Ryze Tech, “Tello EDU,” *Ryze Robotics*. [Online]. Available: https://www.ryzerobotics.com/tello-edu?site=brandsite. [Accessed: 09-Nov-2020].

[4] ArduPilot, “Swarming,” ArduPilot. [Online]. Available: https://ardupilot.org/planner/docs/swarming.html. [Accessed: 09-Nov-2020].

[5] Baldwin, D. (2019, March 9). *dbaldwin/DroneBlocks-TelloEDU-Python*. GitHub. https://github.com/dbaldwin/DroneBlocks-TelloEDU-Python.

[6] Mulgaonkar, Yash, Gareth Cross, and Vijay Kumar. "Design of small, safe and robust quadrotor swarms." 2015 IEEE international conference on robotics and automation (ICRA). IEEE, 2015.

[7] Bingler, Andrew, and Kamran Mohseni. "Dual radio autopilot system for lightweight, swarming micro/miniature aerial vehicles." Journal of Aerospace Information Systems 14.5 (2017): 293-306.

[8] Zhang, Y., Wang, N., Weng, S., Li, M., Mou, D., & Han, Y. (2020, September 15). Emergency Positioning Method of Indoor Pedestrian in Non-Cooperative Navigation Environment Based on Virtual Reference Node Array/INS. https://ieeexplore.ieee.org/document/9093822.

[9] Vicon, “Vicon in Use: Case Studies: Motion Capture Systems,” Vicon, 28-May-2020. [Online]. Available: https://www.vicon.com/resources/case-studies/going-deeper-underground-the-role-of-vicon-virtual-reality-and-serious-gaming-in-sts3ds-training-for-miners/. [Accessed: 26-Mar-2021].

[10] Vicon, “Engineering,” Vicon. [Online]. Available: https://www.vicon.com/applications/engineering/. [Accessed: 26-Mar-2021].

[11] ROS, “About ROS,” *ROS*. [Online]. Available: https://www.ros.org/about-ros/. [Accessed: 09-Nov-2020].

[12] A. Dattalo, “Wiki: ROS/ Introduction,” *ros.org*, 08-Aug-2018. [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Accessed: 09-Nov-2020].

[13] Tau-Adl, “tau-adl/Tello_ROS_ORBSLAM,” *GitHub*, 07-Jun-2020. [Online]. Available: https://github.com/tau-adl/Tello_ROS_ORBSLAM. [Accessed: 09-Nov-2020].

# Appendix

## Port Forwarding Code (On Raspberry Pi)

**Information:**

Project PC IP Address: 192.168.1.100

Drone IP Address: 192.168.10.1

**Code:**

```
sudo iptables -t nat -A PREROUTING -s 192.168.1.100 -i eth0 -j DNAT --to-destination 192.168.10.1

sudo iptables -t nat -A PREROUTING -s 192.168.10.1 -i wlan0 -j DNAT --to-destination 192.168.1.100

sudo bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

sudo iptables -A FORWARD -i wlan0 -o eth0 -j ACCEPT

sudo iptables -A FORWARD -i eth0 -o wlan0 -j ACCEPT

sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

sudo iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
```

# Iptables Command – Redirect Video Feed (On Project PC)

**Information:**

Raspberry Pi IP Address: 192.168.1.104

Port Where Video Feed is Received: 11111

Port Where Video Feed is Changed to: 11118

**Code:**

```
sudo iptables -t nat -A PREROUTING -s 192.168.1.104 -p udp --dport 11111 -j REDIRECT --to-port 11118
```

# FFplay Command to Play Video Feed (On Project PC)

**Information:**

Port to Play Video Feed: 11118

Project PC IP Address: 192.168.1.100

**Code:**

```
ffplay -f h264 -fflags nobuffer -flags low_delay -framedrop udp://192.168.1.100:11118 -framerate 30
```

# ‘NTU’ Flight Pattern (On Project PC)

**Code:**

```
# Import the necessary modules

import socket

import threading

import time

# IP and port of Tello

tello1_address = ('192.168.1.101', 8889)

tello2_address = ('192.168.1.102', 8889)

tello3_address = ('192.168.1.103', 8889)

# IP and port of local computer

local1_address = ('', 9010)

local2_address = ('', 9011)

local3_address = ('', 9012)

# Create a UDP connection that we'll send the command to

sock1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the local address and port

sock1.bind(local1_address)

sock2.bind(local2_address)

sock3.bind(local3_address)
```

```
def command(delay):
  # Try to send the message otherwise print the exception
  message = "command"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def battery(delay):
  # Try to send the message otherwise print the exception
  message = "battery?"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
```

```
def takeoff(delay):
  # Try to send the message otherwise print the exception
  message = "takeoff"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def land(delay):
  # Try to send the message otherwise print the exception
  message = "land"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
```

```
def startposition(delay):
  # Try to send the message otherwise print the exception
  message = "up 80"
  try:
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def spin(delay):
  # Try to send the message otherwise print the exception
  message = "cw 360"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def one(delay):
```

```
  # Try to send the message otherwise print the exception
  message1 = "up 80"
  message2 = "left 80"
  message3 = "down 80"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    sock3.sendto(message3.encode(), tello3_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def two(delay):
  # Try to send the message otherwise print the exception
  message1 = "go 0 80 -80 40"
  message2 = "right 40"
  message3 = "left 80"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    sock3.sendto(message3.encode(), tello3_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
```

```
  # Delay for a user-defined period of time
  time.sleep(delay)


def three(delay):
  # Try to send the message otherwise print the exception
  message1 = "up 80"
  message2 = "down 80"
  message3 = "up 80"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    sock3.sendto(message3.encode(), tello3_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


# Receive the message from Tello
def receive():
  # Continuously loop and listen for incoming messages
  while True:
    # Try to receive the message otherwise print the exception
    try:
      response1, ip_address = sock1.recvfrom(128)
      response2, ip_address = sock2.recvfrom(128)
```

```
        response3, ip_address = sock3.recvfrom(128)
        print("Received message: from Tello EDU #1: " + response1.decode(encoding='utf-8'))
        print("Received message: from Tello EDU #2: " + response2.decode(encoding='utf-8'))
        print("Received message: from Tello EDU #3: " + response3.decode(encoding='utf-8'))
    except Exception as e:
        # If there's an error close the socket and break out of the loop
        sock1.close()
        sock2.close()
        sock3.close()
        print("Error receiving: " + str(e))
        break

# Create and start a listening thread that runs in the background
# This utilizes our receive functions and will continuously monitor for incoming messages
receiveThread = threading.Thread(target=receive)
receiveThread.daemon = True
receiveThread.start()

#All takeoff
command(3)
battery(5)
takeoff(8)
startposition(6)
spin(10)
one(6)
two(8)
```

```
three(5)

spin(10)

land(2)


# Print message

print("Mission completed successfully!")


# Close the socket

sock1.close()

sock2.close()

sock3.close()
```

# Criss Cross Flight Pattern (On Project PC)

**Code:**

```
# Import the necessary modules

import socket

import threading

import time

# IP and port of Tello

tello1_address = ('192.168.1.101', 8889)

tello2_address = ('192.168.1.103', 8889)

# IP and port of local computer

local1_address = ('', 9010)

local2_address = ('', 9012)


# Create a UDP connection that we'll send the command to

sock1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the local address and port

sock1.bind(local1_address)

sock2.bind(local2_address)
```

```
def command(delay):
  # Try to send the message otherwise print the exception
  message = "command"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


def battery(delay):
  # Try to send the message otherwise print the exception
  message = "battery?"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
```

```
  # Delay for a user-defined period of time
  time.sleep(delay)


def takeoff(delay):
  # Try to send the message otherwise print the exception
  message = "takeoff"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


def land(delay):
  # Try to send the message otherwise print the exception
  message = "land"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
```

```
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


def cross1(delay):
  # Try to send the message otherwise print the exception
  message1 = "go 50 50 0 30"
  message2 = "go -50 -50 0 30"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


def cross2(delay):
  # Try to send the message otherwise print the exception
```

```
  message1 = "go -50 50 0 30"
  message2 = "go 50 -50 0 30"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def cross3(delay):
  # Try to send the message otherwise print the exception
  message1 = "go 50 50 0 30"
  message2 = "go -50 -50 0 30"
  try:
    sock1.sendto(message2.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
```

```
  # Delay for a user-defined period of time
  time.sleep(delay)


def cross4(delay):
  # Try to send the message otherwise print the exception
  message1 = "go 50 -50 0 30"
  message2 = "go -50 50 0 30"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


def diagonalupleft(delay):
  # Try to send the message otherwise print the exception
  message1 = "go 0 80 80 30"
  try:
    sock1.sendto(message1.encode(), tello1_address)
```

```
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def right(delay):
  # Try to send the message otherwise print the exception
  message1 = "right 160"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def diagonaldownright(delay):
  # Try to send the message otherwise print the exception
```

```
  message1 = "go 0 80 -80 30"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


# Receive the message from Tello
def receive():
  # Continuously loop and listen for incoming messages
  while True:
    # Try to receive the message otherwise print the exception
    try:
      response1, ip_address = sock1.recvfrom(128)
      response2, ip_address = sock2.recvfrom(128)
      print("Received message: from Tello EDU #1: " + response1.decode(encoding='utf-8'))
      print("Received message: from Tello EDU #2: " + response2.decode(encoding='utf-8'))
    except Exception as e:
```

```
        # If there's an error close the socket and break out of the loop
        sock1.close()
        sock2.close()
        print("Error receiving: " + str(e))
        break

# Create and start a listening thread that runs in the background
# This utilizes our receive functions and will continuously monitor for incoming messages
receiveThread = threading.Thread(target=receive)
receiveThread.daemon = True
receiveThread.start()

#All takeoff
command(3)
battery(5)
takeoff(8)
cross1(8)
diagonalupleft(8)
cross2(8)
right(8)
cross3(8)
```

```
diagonaldownright(8)

cross4(8)

land(4)



# Print message

print("Mission completed successfully!")



# Close the socket

sock1.close()

sock2.close()
```

# Facade Scan Flight Pattern (On Project PC)

**Code:**

```
# Import the necessary modules

import socket

import threading

import time

# IP and port of Tello

tello1_address = ('192.168.1.101', 8889)

tello2_address = ('192.168.1.102', 8889)

tello3_address = ('192.168.1.103', 8889)

# IP and port of local computer

local1_address = ('', 9010)

local2_address = ('', 9011)

local3_address = ('', 9012)

# Create a UDP connection that we'll send the command to

sock1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the local address and port

sock1.bind(local1_address)

sock2.bind(local2_address)

sock3.bind(local3_address)
```

```
def command(delay):
  # Try to send the message otherwise print the exception
  message = "command"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def battery(delay):
  # Try to send the message otherwise print the exception
  message = "battery?"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
```

```
def takeoff(delay):
  # Try to send the message otherwise print the exception
  message = "takeoff"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def land(delay):
  # Try to send the message otherwise print the exception
  message = "land"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    sock3.sendto(message.encode(), tello3_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
```

```
def facadescan(delay):
  # Try to send the message otherwise print the exception
  message1 = "right 70"
  message2 = "up 25"
  message3 = "left 70"
  for i in range(2):
    try:
      sock1.sendto(message1.encode(), tello1_address)
      sock2.sendto(message1.encode(), tello2_address)
      sock3.sendto(message1.encode(), tello3_address)
      print("Sending message: " + message1)
    except Exception as e:
      print("Error sending: " + str(e))
    #Delay for a user-defined period of time
    time.sleep(delay)
    try:
      sock1.sendto(message2.encode(), tello1_address)
      sock2.sendto(message2.encode(), tello2_address)
      sock3.sendto(message2.encode(), tello3_address)
      print("Sending message: " + message2)
    except Exception as e:
      print("Error sending: " + str(e))
    #Delay for a user-defined period of time
    time.sleep(delay)
    try:
      sock1.sendto(message3.encode(), tello1_address)
```

```
      sock2.sendto(message3.encode(), tello2_address)
      sock3.sendto(message3.encode(), tello3_address)
      print("Sending message: " + message3)
    except Exception as e:
      print("Error sending: " + str(e))
    #Delay for a user-defined period of time
    time.sleep(delay)
    try:
      sock1.sendto(message2.encode(), tello1_address)
      sock2.sendto(message2.encode(), tello2_address)
      sock3.sendto(message2.encode(), tello3_address)
      print("Sending message: " + message2)
    except Exception as e:
      print("Error sending: " + str(e))
    #Delay for a user-defined period of time
    time.sleep(delay)

def finish(delay):
  # Try to send the message otherwise print the exception
  message1 = "right 80"
  message2 = "land"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    sock3.sendto(message1.encode(), tello3_address)
    print("Sending message: " + message1)
```

```
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
  try:
    sock1.sendto(message2.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    sock3.sendto(message2.encode(), tello3_address)
    print("Sending message: " + message2)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

# Receive the message from Tello
def receive():
  # Continuously loop and listen for incoming messages
  while True:
    # Try to receive the message otherwise print the exception
    try:
      response1, ip_address = sock1.recvfrom(128)
      response2, ip_address = sock2.recvfrom(128)
      response3, ip_address = sock3.recvfrom(128)
      print("Received message: from Tello EDU #1: " + response1.decode(encoding='utf-8'))
      print("Received message: from Tello EDU #2: " + response2.decode(encoding='utf-8'))
      print("Received message: from Tello EDU #3: " + response3.decode(encoding='utf-8'))
```

```
    except Exception as e:
      # If there's an error close the socket and break out of the loop
      sock1.close()
      sock2.close()
      sock3.close()
      print("Error receiving: " + str(e))
      break

# Create and start a listening thread that runs in the background
# This utilizes our receive functions and will continuously monitor for incoming messages
receiveThread = threading.Thread(target=receive)
receiveThread.daemon = True
receiveThread.start()

#Code
command(3)
battery(5)
takeoff(8)
facadescan(8)
finish(5)

# Print message
print("Mission completed successfully!")

# Close the socket
sock1.close()
```

```
sock2.close()

sock3.close()
```

# Area Scan/Patrol Flight Pattern (On Project PC)

**Code:**

```
# Import the necessary modules

import socket

import threading

import time

# IP and port of Tello

tello1_address = ('192.168.1.105', 8889)

tello2_address = ('192.168.1.104', 8889)

# IP and port of local computer

local1_address = ('', 9010)

local2_address = ('', 9011)


# Create a UDP connection that we'll send the command to

sock1 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the local address and port

sock1.bind(local1_address)

sock2.bind(local2_address)


def command(delay):

  # Try to send the message otherwise print the exception

  message = "command"

  try:

    sock1.sendto(message.encode(), tello1_address)
```

```
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def battery(delay):
  # Try to send the message otherwise print the exception
  message = "battery?"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def takeoff(delay):
  # Try to send the message otherwise print the exception
  message = "takeoff"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
```

```
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def land(delay):
  # Try to send the message otherwise print the exception
  message = "land"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def streamon(delay):
  # Try to send the message otherwise print the exception
  message = "streamon"
  try:
    sock1.sendto(message.encode(), tello1_address)
    sock2.sendto(message.encode(), tello2_address)
    print("Sending message: " + message)
  except Exception as e:
    print("Error sending: " + str(e))
```

```
  # Delay for a user-defined period of time
  time.sleep(delay)

def turn1(delay):
  # Try to send the message otherwise print the exception
  message1 = "ccw 90"
  message2 = "cw 90"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def turn2(delay):
  # Try to send the message otherwise print the exception
  message1 = "cw 90"
  message2 = "ccw 90"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message2.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
```

```
  # Delay for a user-defined period of time
  time.sleep(delay)

def one(delay):
  # Try to send the message otherwise print the exception
  message1 = "forward 110"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def two(delay):
  # Try to send the message otherwise print the exception
  message1 = "forward 364"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)
```

```
def three(delay):
  # Try to send the message otherwise print the exception
  message1 = "forward 308"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def four(delay):
  # Try to send the message otherwise print the exception
  message1 = "forward 672"
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)

def forward(distance, delay):
```

```
  # Try to send the message otherwise print the exception
  message1 = "forward " + distance
  try:
    sock1.sendto(message1.encode(), tello1_address)
    sock2.sendto(message1.encode(), tello2_address)
    print("Sending message: " + message1)
  except Exception as e:
    print("Error sending: " + str(e))
  # Delay for a user-defined period of time
  time.sleep(delay)


# Receive the message from Tello
def receive():
  # Continuously loop and listen for incoming messages
  while True:
    # Try to receive the message otherwise print the exception
    try:
      response1, ip_address = sock1.recvfrom(128)
      response2, ip_address = sock2.recvfrom(128)
      print("Received message: from Tello EDU #1: " + response1.decode(encoding='utf-8'))
      print("Received message: from Tello EDU #2: " + response2.decode(encoding='utf-8'))
    except Exception as e:
      # If there's an error close the socket and break out of the loop
      sock1.close()
      sock2.close()
      print("Error receiving: " + str(e))
```

```
    break

# Create and start a listening thread that runs in the background
# This utilizes our receive functions and will continuously monitor for incoming messages
receiveThread = threading.Thread(target=receive)
receiveThread.daemon = True
receiveThread.start()

#All takeoff
command(3)
battery(5)
streamon(30)
takeoff(6)
forward("110", 8)
turn1(6)
forward("364", 8)
turn1(6)
turn2(6)
forward("308", 8)
turn1(6)
turn1(6)
forward("672", 10)
land(2)

# Print message
print("Mission completed successfully!")
```

```
# Close the socket

sock1.close()

sock2.close()
```