UNIVERSITY OF HULL

Development of Autonomous Crazyflie Swarm

and

Physicomimetic Emergence

in

Swarm Topologies

Final report

Submitted for the BSc in

Computer Science

April 2020

by

Onayo Moro

Word count: 16,496

Abstract

This project explores the potential implementation of physicomimetic emergent algorithms in a Crazyflie swarm. Additionally, it explores the aspects of development involved in utilising the Crazyflie drones for said swarm, including developing appropriate modifications to the Crazyflie API, firmware and GUI for the Crazyflie client. This project also makes use of an emergent swarm simulation developed in the Unity Engine to serve as a digital twin to its physical Crazyflie counterpart. These simulated drones are regulated by a PID controller that has been iteratively fine-tuned to maintain their flight as well as the structure of the swarm. Different topologies that vary in their decision-making process are developed to discover an optimal solution for implementing emergent behaviour in a Crazyflie swarm. This project finds that, contrary to previous assumptions, topologies that implement the least interaction complexity perform better than topologies that consider more information and implement more complex decision-making methods. These conclusions are visualised and represented with empirical data showing different types of interactions within the swarm.

Acknowledgments

Firstly, especially before the effects of COVID-19 affected the course of research for this project, Ieaun Roberts' advice, and contribution to the field was certainly appreciated. Understanding how to interface with the Crazyflie API in particular, proved awkward in the beginning stages of development; the research Ieaun Roberts conducted in the prior academic year was certainly a helpful base to start from. Before drones could be acquired for this project, Roberts trustingly allowed one of his Crazyflie drones to be borrowed for initial testing in early stages of development. Hopefully, the drone could be returned in one piece. Regardless, the gesture was greatly appreciated.

As my secondary marker, Dr Amadou Gning's advice on how the methodology of this project should be carried out was incredibly helpful. This project's development certainly had unforeseen roadblocks and issues. Nevertheless, the advice Dr Gning provided helped to see an improvement in the project's management of both resources and time. Many thanks indeed.

Last but certainly not least, many thanks are in order for the overall guidance Dr Yongqiang Cheng was able to offer during the course of this project as my primary marker and module leader. Even after crashing one of his prized Crazyflie drones during testing, he was kind enough to repair the drone and return it into my trusting possession. Dr Cheng's counsel was much appreciated, whether it was discussing any problems the project encountered or technical advice during development.

To my mother, for all her support.

Effects of Covid-19

Unfortunately, the year 2020 faced an unforeseen viral threat known as COVID-19. As the UK government moved to place necessary restrictions on public movements, the University of Hull decided to limit access to the campus until further notice. This resulted in the severe limiting of the project's original technical scope as development could not be continued without using specific lab equipment, most notably the Vicon Vision System.

The project's original purpose was to make use of at least two Crazyflie Drones developed by Bitcraze. Retro reflective balls would then be attached to said drones in order to obtain their positional data via the Vicon Cameras. Development of PID controllers and emergent algorithms would then allow the drones to behave like a swarm whilst they moved towards a virtual visual trajectory using said cameras.

As the Vicon Cameras were responsible for providing the positional data of the Crazyflie drones and testing any software, many aspects of development became impossible. This is due to the positional data being responsible for allowing the movement of the drones, providing an input for the PID controller and feeding a visual trajectory for the drones.

In response, whilst the aims and objectives saw changes to avoid the need for specialised hardware, time suddenly became a constraint due to the timing of the outbreak. Nevertheless, the original main objective for this project will remain mostly the same.

Contents

1 Introduction

1.1 Purpose of this Project

As drone swarm technology advances, attempts are being made to move towards individual drones having full autonomy. This is opposed to previous methods that utilise external sensors to track and feedback drone positions to a single external computer system. This system processes the information then sends instructions to the individual drones in a swarm. As more sophisticated on-board systems are developed, drones become more capable of collecting more complex information about their environment. As a result, more complex decisions can be made without the use of dedicated external instruments or computer systems. This project aims to explore how such complex decisions may affect the emergent behaviour of a swarm.

1.2 The Pursuit of Autonomous Swarms

Whilst on one hand, there are valid humanitarian applications to support the development of autonomous drone swarms, the motivation appears to mainly concern military applications. Kallenborn (2018) proclaims that the era of the drone swarm is coming, and we need to be ready for it. As the autonomous capability of drones increase, their ability to obtain and share complex information will allow for more complex decision making.

1.2.1 Military Motivations

The aforementioned developments will have significant applications to almost every area of national and homeland security. Dispersing swarms to survey large areas of land and sea for hostile entities as well as detecting biological, chemical or radioactive threats could serve to preserve the lives of military personnel and the civilian population (Kallenborn, 2018).

It is also reasonable to assume that this technology could be used to autonomously discriminate hostile entities, allowing them to take either lethal or non-lethal action. Nevertheless, it was hypothesised that such autonomous weapons systems were unlikely to be developed in the near future following an expert meeting for Autonomous Weapon Systems, Technical, Military, Legal and Humanitarian Aspects in 2014. During a speaker's summary, Professor Noel Sharkey (2014) from the University of Sheffield, UK, stated that despite decades of research, current autonomous targeting methods only work in low cluttered environments. For instance, tanks in deserts or ships at sea present little environmental features making recognition systems better able to accurately distinguish targets. The capabilities of autonomous systems are unlikely to make significant advancements in the near to medium term future, although improvements are expected in the longer term (Sharkey, 2014).

There is however an incentive to use autonomous swarms for on the ground support during military operations. Defence research companies, most notably DARPA, are testing and investing in autonomous drone swarm technologies. DARPA's Offensive Swarm-Enabled Tactics program (OFFSET) intends small-unit infantry forces to use swarms of upwards of 250 small unmanned aircraft systems (UASs) and potentially small unmanned ground systems (UGSs) to accomplish diverse missions in complex urban environments (Dr Chung, n.d.).

1.2.2 Humanitarian Motivations

The aforementioned advancements in drone swam technology may also play a large role in supporting search and rescue efforts in otherwise precarious environments and vast areas of land. In natural disaster situations such as earthquakes or tsunamis, swarms can be deployed in mass to scout in and over cities, villages and towns for survivors, locating them for rescue. This would remove the need for multiple expensive manned aircraft, constantly burning fuel to stay in the air to scout out vast disaster areas. Autonomous systems could also facilitate in putting effective counter measures in place for

rescue teams by providing data on the scouted area (Tahir, 2019; Böling, 2019; Haghbayan, 2019; Toivonen, 2019; Plosila, 2019). The same is true for mountainous regions or deep cave systems where the hazardous terrain demands for the careful planning of a rescue.

1.2.3 Recreational Use

Since the inception of drone swarms, their ability to produce mesmerising light shows and form spellbinding patterns in the sky has captured the hearts and minds of many researchers in the field as well as other spectators. An applicable benefit of such technologies could be their use in the entertainment industry. Swarms could be deployed in events such as concerts for the audience's entertainment provided the drones are reliable enough to avoid malfunctioning over crowds.

1.3 Research Question

Based on the different decision-making methods an autonomous drone may utilise, how will the emergent behaviour of a drone swarm be affected?

This project hypothesises that the greater the interaction magnitude between autonomous drones within a swarm, the greater the swarm cohesion, structure and stability.

1.4 Aims and Objectives

Before it became necessary to change the project's objectives, much research exploring using the Crazyflie API to create emergent swarms had already been done. To reflect this, this project will detail the lessons learned prior to the changes to provide hypothetical solutions for achieving an emergent Crazyflie swarm. The new main objective will effectively serve as a Digital Twin to the original objective as it will serve to simulate the behaviours that could potentially occur in a physical solution.

1.4.1 Original Objective

This project's original objective was to develop algorithms to control the flight behaviour of a swarm of Crazyflie drones. The algorithms would invoke emergent behaviour in the drones whilst the swarm is given a virtual visual trajectory. Such emergent behaviour would prevent the drones from colliding with each other during flight whilst maintaining a cohesive, collective swarm structure.

1.4.2 Main Objective

Swarm topology references the decision-making process that generates the network of communication between agents in a swarm (Bratton & Kennedy, 2007; Kennedy & Mendes, 2002). Iterative development of swarm topology algorithms will be carried out using the Unity 3D Physics Engine with the intention of realising an optimal topology for invoking desirable emergent behaviour. This behaviour must also be maintained by a Proportional, Integral and Derivative (PID) controller whilst the swarm is given a visual trajectory. Such emergent properties must allow the swarm to move through an environment without making contact with the other entities in the swarm to simulate drones preventing damage to themselves and other drones.

1.4.3 Primary Objectives

- A user interface to manage the visual trajectory of a single Crazyflie drone will be developed using the Crazyflie API and QT designer software.
- The User interface will be iteratively developed to manage the visual trajectory of a swarm of drones.
- The simulated drone's trajectory must be calculated in order to invoke movement in a given direction. Its' trajectory, for now, will be dictated by an end position.

- The PID controller, which is responsible for error correction during flight, must be optimised to stabilise the flight of the drone. Without this stability, the drone may be at the mercy of the physics engine, making them liable to fly off course and out of control.
- Whilst optimising the PID controller, qualitative research will be carried out to gain insight on the performance of different configurations in order to optimise and fine turn the controller.
- After gaining an understanding of how to manipulate the drone in 3D space and optimising the PID controller, efforts can be made to scale up testing to multiple drones. A drone's trajectory will now also depend on the end position as well as its proximity to other drones in the swarm.
- Emergent algorithms will be iteratively developed to maintain a structure within the swarm.
- Qualitative research on swarm topologies will also be carried to optimise the emergent behaviour in the simulation.
- To add some realism to the simulation, white noise will be added to the values responsible for positioning and movement using the Box Muller method. This will emulate the noise a system would encounter when gathering data on its environment as well as the noise within its own system.
- There will be a detailed comparison between the emergent behaviours of the different tested swarm topologies.

1.4.4 Secondary Objectives

- Graphing UI elements may be used to track the inertia measurements of a drone within the Crazyflie user interface.
- Other threats to the integrity of the swarm may come from the environment itself. Simulated wind may be implemented to test the integrity of the emergent algorithms as well as the error correction capabilities of the PID controller.
- Other challenging environmental hazards may also be implemented to test the obstacle avoidance capabilities of the emergent algorithms. This will also show how well the structure of the swarm is maintained by the algorithms.
- Qualitative research on other research papers on certain aspects of the project may be conducted.

1.5 Project Organisation

The structure of this project will be organised as follows:

1. Theorise methods of executing an autonomous swarm using the Crazyflie drones developed by Bitcraze.
2. Develop a user interface responsible for setting up and controlling a Crazyflie swarm using the Crazyflie API released by Bitcraze. It must be intuitive enough to use whilst also displaying important diagnostics to the user.
3. Create a rudimentary simulation of a drone, controlled by a PID controller, moving to a virtual trajectory.
4. Iteratively develop different emergent behaviours to simulate drones in a swarm setting whilst also moving to a visual trajectory.
5. There will be an evaluation of the project's achievements.
6. A conclusion will then summarise the projects results.

2 Literature Review

2.1 Turn Virtual Trajectory into Reality by Flying Drones

Since the opensource Crazyflie platform was released in early 2013, it has become a prime platform for autonomous UAV development. Before an autonomous swarm can be achieved, one must understand the methods necessary to turn virtual trajectory into reality. Ieaun Roberts details the techniques he used to achieve this using the Crazyflie 2.0 platform. At the time, the 2.0 platform appeared to be a prime choice for drone research due to its widespread adoption in academic institutions (Roberts, 2019). As a result, support for Crazyflie 2.0 versions or above can easily be found as opposed to the earlier 1.0 that was discontinued following the release of 2.0 in 2014.

Ieaun Roberts was able to develop a PID controller to regulate the pitch, yaw, thrust and roll of a Crazyflie drone with the goal of moving the drone from one position to another. For the PID controller to be able to deliver an output, the position of the Crazyflie drone is obtained using the Vicon Vision Cameras. PID controllers can be described as a non-linear control loop where the difference between the desired end position and the drone's current position is calculated (Roberts, 2019). This difference, known as the error, influences the PID's output, in turn, influencing the flight of the Crazyflie; moving it closer to the desired end position. This is an integral function that allows dynamic correction of a flight path in three-dimensions.

To obtain the Crazyflie's positional data, the Vicon Vision Cameras provided the necessary indoor tracking capabilities. They function by only being able to track retro-reflective surfaces as opposed to conventional cameras that see reflective surfaces. Maliszewski (2015) states, reflective surfaces normally bounce light away from the light source when it's reflected. On the other hand, retro-reflective surfaces bounce light back in the direction the light came. The most common way it does this is by using tiny glass beads embedded within fabric. Several retro-reflective mo-cap markers were attached to the Crazyflie allowing precise tracking of the drone in 3D space, thus providing the PID controller with an input.

Ieaun Roberts was able to develop methods of interfacing with the Crazyflie through the use of the Crazyflie API and the Crazyflie Client provided by Bitcraze. Using the provided QT Designer software, a custom user interface could be developed as a way for users to interface with custom modifications made to the Crazyflie Client software. The final version of Roberts' custom user interface was able to connect to the Vicon Vision System to stream the Crazyflie's positional information. A wireless UDP connection was used to achieve this with the port and IP address able to be manually changed by the user. Furthermore, custom x, y and z coordinates could be given to command the Crazyflie to move to a certain virtual trajectory. In addition to this, system stats concerning the state of the drone during flight were displayed. Temperature, thrust, roll, pitch, yaw as well as the drone's current position were all displayed for diagnosis during runtime.

Before the drone's positional data can be utilised, the data packets received from the Vicon Vision Systems are decoded in order to interpret its data. These packets contained information concerning the drone's properties for example rotation and transformation. To achieve this, a function called Connect UDP was created that assigned values and searched for the selected subnet mask according to what the user had selected within the client. The function then attempts to bind the specified address and port. If successful, the data from the stream can then begin to be parsed (Roberts, 2019). Essentially, parsing the data stored in the packets allows that data to be saved as variables that can be used for other systems, most notably the PID controller.

"b'3\x89.\x00\x01\x00H\x00Glasses02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00{\x18\xa1_R\xee\x92@l\xeb~\xab:\xc5\x93\xc0KM\x11\xb6&\xc6\x87@4\x91W\r\xfd\xb5\x90?\xdd"?\xaa\xd6\x90?\x02\xadMI\x99O\x06\xc0\x00"

Figure 1 Raw 256-byte width UDP data from Vicon Broadcast (Roberts, 2019)

```python
#parse UDP datastream into variables
def Parse(self, data):
    FrameNumber = data[1] + data[2] + data[3]
    Itemsinblock = data[4]
    ItemNamez = data[8:31]
    TransX = TransCoverter(self,32, 39)
    TransY = TransCoverter(self,40, 47)
    TransZ = TransCoverter(self,48, 55)
    RotX   = TransCoverter(self,56, 63)
    RotY   = TransCoverter(self,64, 71)
    RotZ   = TransCoverter(self,72, 79)
    return FrameNumber, Itemsinblock, ItemNamez, TransX, TransY, TransZ, RotX, RotY, RotZ
```

Figure 2 Parse UDP Data Stream (Roberts, 2019)

Overall, Ieaun Roberts' research lays the technical groundwork for how the Crazyflie platform can be incorporated with indoor tracking systems like the Vicon Vision Systems. In the future, the techniques used here could potentially be used to scale up the number of drones to begin testing emergent swarm behaviour whilst moving to a virtual visual trajectory.

2.2 Optimised Flocking of Autonomous Drones in Confined Environments

The pursuit for autonomous robotic behaviour brought about the further development of embedded technologies that allowed for environmental awareness within a robotic system. These technologies may have provided such systems with the ability to acknowledge objects and their state, the distance of said objects as well as their relative position and rotation in 3D space. With this information, autonomous vehicles, including UAVs, now have the opportunity to make predictions of what may occur around them in the near future. Such an ability would allow an autonomous system to then make decisions based on these predictions.

Desirable swarm intelligence is based upon the principle of self-organisation as observed in insects (Hallinan, 2013). Arguably one of the most mesmerising demonstrations of swarming intelligence is the autonomous drone flocking research conducted by G. Vásárhelyi, Cs. Virágh, G. Somorjai, T. Nepusz, A. E. Eiben and T. Vicsek in 2018. A fleet of 30 autonomous drones were able to display emergent flocking behaviours within a confined environment whilst avoiding collisions with other drones in the swarm. Inspiration during development came from studying the flocking behaviours exhibited by other animals in nature. Such collective behaviour observed in, for instance, birds or fish, control themselves by making decisions using information gathered from their neighbours and local measurements (Nepusz, 2018). By creating bio-inspired algorithms, the drones were able to maintain constant communication whilst airborne without being controlled using a central ground station.

Similar to natural systems, the drones form coherent groups, split and re-unit around obstacles and turn collectively around walls (Vásárhelyi, 2018). The drones were able to mitigate crowding when

approaching closed off areas as the drones in the front of the swarm were able to notify the drones in the back of their intentions. Testing of the algorithms used took place within a realistic simulation where the algorithm's parameters were fine-tuned. The solution was also optimised using an evolutionary algorithm to optimise the speed and coherence of the flock whilst minimising oscillations and collisions (Vásárhelyi, 2018). To visualise the movement of a drone, a light attached to its base changes colour depending on the drone's direction; giving credit to the experiment's visually mesmerising reputation.

The swarm's behaviour during the beginning stages of flight show hesitation before a consensus of the direction of the swarm is made and they fly away smoothly (Nepusz, 2018). The performance of the algorithmic model is assessed by how efficiently the swarm is able to make collective decisions. This is signified by the natural emergence of collective motion patterns during flight (Nepusz, 2018).

The research G. Vásárhelyi and his peers conducted do not demonstrate the swarm's behaviour when given a visual trajectory by the user. More importantly, on the other hand, their emergent algorithms demonstrate how the swarm can seamlessly cooperate in order to maintain their collective structure whilst avoiding obstacles. These important achievements also show how observing nature can give inspiration to developing the rules used to achieve these emergent behaviours. Furthermore, their research shows how evolutionary algorithms can be used to improve a swarm's emergent algorithms.

2.3 Controlling Multiple Crazyflies

Whilst Ieaun Roberts' research detailed the techniques used to achieve virtual visual trajectory flight, there have been other studies that instead focus on connecting and controlling multiple Crazyflie drones simultaneously. One such study by Yifan Zhang from the University of California Riverside, was able to synchronise eight Crazyflie drones using the Crazyflie API. The thesis focused on the experimental implementation and optimisation of distributed time-varying cost function algorithms. These algorithms were tested by flying the eight Crazyflie drones in formation whilst following a trajectory.

Zhang (2019) states that distributed optimization represents an interesting aspect of a multi-agent system where all the agents work cooperatively to find the optimal solution for the whole team using only local information and local communication. The local information is provided by the Vicon System, responsible for feeding back the drone's positions, whereas the local communication references the drone's proximity to the radio transmitter. Essentially, Zhang's research looks for the optimal algorithm to use to send instructions to the eight drones using a single Crazyflie Radio. This algorithm also dictates the topology for communication with the swarm.

Achieving control of multiple Crazyflie drones with one radio was made possible through the Crazyflie Client allowing the drone's frequency address to be changed. The radio channel parameter should be another value between 0-100 when the number of Crazyflie drones exceeds 15 (Preiss et al., 2016). Each Crazyflie drone requires a different address, however other parameters can stay the same. This is true for when using the Crazyswarm or the Crazyflie ROS platform which were used in the experiment (Zhang, 2019).

2.4 Crazyflie 2.0 Nano-Quadcopter Simulation

To make this project's simulation more accurate, simulating the actual functions of a drone would give insight into the potential systems necessary for autonomous flight as well as their effect on flight behaviour. A study by Giuseppe Silano and Luigi Iannelli aimed to model a Crazyflie 2.0 nano-quadcopter in a physics-based simulation environment (Silano & Iannelli, 2018). This was achieved using ROS (Robot Operating System) with the CrazyS extension of the ROS package RotorS. To develop

the Crazyflie 2.0 physical model, systems imperative for flight, including the flight control system and on board IMU (Inertia Measurement Unit) had to be modelled.

When designing the flight control system, a common architecture for controlling the position of a quadrotor was considered (Silano & Iannelli, 2018). A Reference Generator was created which takes the position to reach (x_r, y_r, z_r) and the desired yaw angle ψ_r to generate command signals $(\theta_c, \varphi_c, \Omega_c, \psi_c)$ that are inputs for the onboard control architecture for the Crazyflie (Silano & Iannelli, 2018). In a real-life system, the drone's position and velocity may come from a motion capture system, for example Vicon, Optitrack or Qualisys (Silano & Iannelli, 2018). In this simulation, the data came from a virtual odometry sensor which can indicate change in position, or how far a robot has travelled, over time (LaValle, 2012).

	Sym.	Unit	Output limit
Roll command	ϕ_c	rad	$[-\pi/6,\ \pi/6]$
Pitch command	θ_c	rad	$[-\pi/6,\ \pi/6]$
Yaw rate command	$\dot{\psi}_c$	$\mathrm{rad\,s^{-1}}$	$[-1.11\pi,\ 1.11\pi]$
Thrust command	Ω_c	UINT16	$[5156,\ 8163]$

Table 1 Physical constraints of the Crazyflie 2.0 nano-quadcopter (Silano & Iannelli, 2018).

The simulated onboard controller is made of two parts, the altitude and rate controller. The altitude control system is designed to increase the positioning accuracy in flight on the z axis of the inertial system in a UAV (Vidan et. al., 2018). The opensource Crazyflie firmware released by Bitcraze allowed the accelerometer and gyroscope data to be used to estimate the drone's altitude and angular velocities used by the onboard control loop (Silano & Iannelli, 2018).

Through the efforts of creating a viable flight control system and modelling the functionalities of the Crazyflie 2.0, the project has illustrated how to expand on the functionalities of the ROS package RotorS for integrating a nano-quadcopter (Silano & Iannelli, 2018). Additionally, the simulation shows how the platform can be used to understand the behaviour of a flight control system through the evaluation of different indoor and out-door scenarios (Silano & Iannelli, 2018).

2.5 Network Topology Inference

Swarm systems consider the propagation of local interactions in a communication network in order to express a global behaviour like reaching a goal (Vasquez, 2018; Barca, 2018). The swarm's topology is dictated by how the drones in the swarm interact locally. The study, completed by Bruno Luis Mendivez Vasquez, 2018 and Jan Carlo Barca, 2018 focused on constructing a topological graph to visualise both the magnitude and orientation of swarm interactions. Such a structure is used obtain information concerning the global parameters within a swarm, such as leadership, and to identify correlations between the distribution of swarm interaction magnitudes and swarm parameters (Vasquez, 2018; Barca, 2018).

Interaction magnitudes are a derivative of trajectory distances from nearest neighbours whilst leadership was dictated by the position vector orientation in local neighbourhoods (Vasquez, 2018; Barca, 2018). In essence, the closer objects are in the swarm and the greater the number of objects, the greater their interaction, thus dictating interaction magnitude.

In this study, when testing the flocking controller, interaction magnitude and swarm density were modelled in different scenarios to identify relationships. It was found that when swarming around a

point, the interaction magnitude increases as the swarm size increased. This is likely due to the proximity of objects in the swarm when they are forced to swarm around a point.

Morphogenesis means the generation of; in the context of biology, it means the generation of tissue organisation (Bard, 2008). It describes how cells are able to organise themselves to create complex structures (Vasquez, 2018; Barca, 2018). When testing the morphogenesis controller, communication range was the independent variable, however, this did not appear to influence the distribution of interactions in the swarm.

Test were also conducted using a physicomimetics controller. Physicomimetics is based on an artificial representation of physics where agents behave like point mass particles responding to artificial forces generated by local interactions with other nearby particles (Potter, n.d.). With the physicomimetics controller, goal-based (virtual trajectory) scenarios showed no difference in interaction magnitudes across different swarm sizes. Furthermore, communication ranges appeared to narrow interaction distribution to higher values (Vasquez, 2018; Barca, 2018).

Part of their conclusion saw that for the controllers tested, communication ranges did not have an influence on the distribution of interactions. This held true when testing for different swarm sizes in goal-based, swarming around a point, and obstacle avoidance scenarios.

Whilst the interaction magnitude may indicate how effective different topologies are at achieving certain goals, more direct testing detailing this aspect of performance could be helpful. Obstacle course training grounds to test the cooperative abilities of different topologies could give insight into how well swarms can maintain a cohesive structure and split apart when necessary.

2.6 Digital Twin

Thanks to the advent on the Internet of Things (IoT), the concept of a digital twin has become more cost effective to implement and develop. A digital twin describes a virtual system or model created to emulate a system in the real world. This allows the analysis of data and monitoring of systems to foresee potential problems before they have a chance to occur in the real world (Marr, 2017). In a sense, the studies that created simulations of drone systems or swarm behaviours are digital twins of their physical real-world counter parts; simulations designed to experiment with methods before physical implementation. The twin would represent a UAV's individual components as well as its integrated whole, using physics-based models to capture the details of its behaviour (Flaherty, 2019).

The ability for researchers to test these systems, without needing to spend more time and money on physical tests, adds to the viability of the digital twin. The digital twin approach was used in the creation of a prototyping-production of a functioning UAV. With this method, the construction and optimisation of new and existing aircraft models was now more feasible (Tomsk State University, 2019).

In the context of this project, the emergent behaviours modelled may be indicative of the behaviours of a real-world implementation. The iterative swarm tests could potentially serve as a digital twin if the same algorithms were realised in, for example, the creation of a Crazyflie swarm.

3 Requirements for Crazyflie API

Due to the reasons stated in the Effects of COVID-19 section of this report, the constraints on research warranted a refocusing of the objectives, and subsequently the deliverables, for this project. As a result, there will be separate sections in this document dedicated to two aspects of development. One section will be the on the theoretical requirements for the Crazyflie Swarm with diagrams showing how the system may have functioned. Another will discuss the theoretical development of the emergent swarm simulation.

3.1 Product Requirements

3.1.1 Crazyflie Swarm

An important requirement for controlling the Crazyflie drones is to effectively give the drones a coherent three-dimensional understanding of its environment. This is so that the drones can be controlled within the confines of the environment they are operating in. This also allows the drones to effectively track their positions as well as the positions of other drones in the swarm. This information will be provided by the Vicon Cameras that will stream the drone's positional data back to the Crazyflie API.

To maintain steady and stable flight, the drone's velocity will be moderated by a PID controller. This must ensure that the drone maintains its course during flight. Furthermore, acceleration will also be maintained to mitigate overshoot which may cause the Crazyflie to fly past its target or potentially crash. The error correction capabilities must also be able to stabilise the drone's flight against simulated weather effects such as wind.

Swarm cohesion must also be maintained by the PID controller. This requires the PID controller to maintain a desired distance between a Crazyflie drone's nearest neighbours. If a drone either moves to far or too close to a neighbouring drone, the PID controller must work to correct that distance.

3.1.2 Crazyflie GUI

Mainly functioning as a proxy, the Crazyflie GUI, integrated into the Crazyflie API, will be responsible for handling communications between the Vicon Systems, Crazyflie API and the Crazyflie drones themselves. The GUI must also be able to have some level of debugging capabilities in case diagnosis is needed during a malfunction. This could simply be testing the connection between the client and the Crazyflie or the Vicon System.

Upon connecting to both the Vicon System and the Crazyflie drone, the GUI must be able to display certain system stats to inform the state of the different systems to the user. This will allow the user to better analyse the system during runtime and monitor performance.

Where the Crazyflie Client is responsible for connecting the drone to the radio, the GUI will be responsible for connecting the Crazyflie Client and the Vicon System. This will be done using a port number and IP address.

Regarding giving the drone a virtual visual trajectory, the GUI must, at least, be able to use (x, y, z) coordinates to tell a drone where to move in one-dimensional space. Perhaps during later stages of development, it may be possible to fully integrate pathed movement. This would be similar to giving the drones a series of one-dimensional trajectories during flight, moving from one position to the next.

Once the functionality for one drone is met, modifications to the GUI should enable it to manage being connected to multiple drones. Displaying drone stats, for instance pitch, yaw and roll, for the whole swarm plays an imperative role in monitoring performance.

3.2 Functional Requirements

3.2.1 Crazyflie Swarm

3.2.1.1 Functional Capabilities

Before the drones can be flown, the ability to connect to the Vicon System and receive its packets must be implemented into the Crazyflie API. In order to interpret the incoming packets from the Vicon System, the Crazyflie API must be able to parse the incoming data for later use.

The PID controller must be able to use the drone positional data obtained from the Vicon System to error correct the flight path of the drones.

The Crazyflie Client must be able to change the radio addresses of each Crazyflie drone with the purpose of configuring unique addresses for each drone.

The Crazyflie Radio Firmware must be able to loop through the unique radio addresses of each drone. This allows the system to communicate and control multiple Crazyflies simultaneously.

3.2.1.2 Performance Requirements

It is imperative that the drones are able to quickly and safely move through their environment and within the swarm as well correct their flight path if any deviations occur. This requires the configuration of the PID controller and the values responsible for its error correction capabilities to be fine-tuned to increase its proficiency.

Latency in radio communication between the Crazyflie drones and Crazyflie Client must be minimal. This is to ensure that any latency does not cause a lapse in communication. Such a scenario may result in a drone not receiving a crucial instruction fast enough which may lead to unstable flight or a collision.

As the Crazyflie API runs on a Linux Virtual Machine, system performance may decrease as a result of emulating the Linux Operating System. To mitigate this, configurations to memory allocation and CPU core configuration, as well as graphical settings, can improve performance, and subsequently, the performance of the Crazyflie API.

3.2.1.3 Data Structures/Elements

The data packets from the Vicon System will contain the positional data of the drones that are currently being tracked and how their position has transformed.

After receiving the data packets from the Vicon System, the data parsed will simply be stored as global variables. Storing these variables globally will allow them to easily be accessed by other methods within the API.

3.2.1.4 Safety

To mitigate any damage to the Crazyflie, its environment or any persons that be present, the flight control algorithms, which include the PID controller and the emergent swarm algorithms, must work to keep the drones within bounds of the dedicated air space.

3.2.1.5 Reliability

The PID controller's configuration must be reliable enough to be utilised by every drone within the swarm. This could be judged by how well the drone is able to maintain its course and how accurately is lands at its destination.

3.2.1.6 Quality

Regarding the accuracy of the system, having high quality communication with the Vicon System or the Crazyflie drones will minimise the chances of the system experiencing packet loss. Such an experience may require the same data packet to be sent again if using a TCP (Transmission Control Protocol) connection. Loosing information or not receiving data on time may undermine the systems accuracy, resulting in less accurate decisions being made with out of date data by said system.

3.2.1.7 Constraints and Limitations

Depending on the size of the air space provided and the proficiency of the PID controller, it may be reasonable to limit the speed of the Crazyflie drones during flight through the Crazyflie Client. This is to reduce the chances of a drone flying too fast for the PID controller to account for, potentially putting the drone at risk.

3.2.2 Crazyflie GUI

3.2.2.1 Interface

The interface must display the relevant information clearly and concisely. Functions dedicated to similar tasks will be kept close to each other in boxed off sections. This will clearly indicate which UI elements, for example buttons or, command boxes, are related to which task.

Furthermore, sections related to user inputs will occupy one side of the interface whilst sections dedicated to displaying system information and stats will occupy another side. In theory, this method of segregation will make navigating the interface less confusing.

3.2.2.2 Functional Capabilities

The Crazyflie GUI must have a section dedicated to connecting with the Vicon System. This section will offer a drop-down menu to apply a default port number and IP address. However, the user must also have the option to manually insert their own port number and IP address and possibly save these values for later use in the same drop-down menu.

The GUI must accept (x, y, z) coordinates from the user to allow the drone to move in one-dimensional space. The user would acquire the coordinates using the Vicon System and manually type in the (x, y, z) values into the interface. Further on in development, instead on only being able to accept one set of coordinates, a three-dimensional graph could be used to set a route for the drone to follow. If a three-dimensional graph is not available, multiple coordinates could be typed in by the user and saved as an array to plan out a flight path.

Commands to control the motors of a Crazyflie will be implemented. Its main function will be to test the connection between the Crazyflie Client and the drone itself. These commands would control the drone's pitch, yaw, thrust and roll as well as a function to disable or enable the motors all together.

Received information about the state of the drone would come from both the Vicon System and the Crazyflie drone itself. The Vicon System is responsible for streaming data packets carrying information about a drone's position. The modified Crazyflie API would then parse the data and retrieve the drone's current position, displaying it in the GUI for the user to monitor. The Crazyflie drone also sends back data regarding its current state, allowing the GUI to display the drone's roll, pitch, yaw and thrust values in real time.

3.3 Design Constraints

3.3.1 Crazyflie Swarm

As the Vicon System can only be used in an indoor lab, the Crazyflie drones cannot be tested outside to test the algorithm's reliability against real world weather conditions. To mitigate this issue, fans could be used to simulate the effects of wind at different intensities.

Due to the Crazyflie's small size and meagre weight limit, the drone is unable to carry any extra embedded sensors that could be used to navigate their environment or detect the presence of other Crazyflie's in the swarm. This makes the use of an indoor tracking system, like the Vicon System, necessary. However, this would limit where the Crazyflies can be operated.

3.3.2 Crazyflie GUI

As the Vicon Systems lab was unavailable during important stages of this project's development, a major constraint is that most of the GUI's functionality cannot be tested. Therefore, most of its development is mostly focused on the design of its interface and theoretical functionality.

When scaling up the number of drones the GUI would need to manage, certain functions may need to adapt the way they display information depending on the number of drones connected. The limitations on how many UI elements QT Designer offers may impact the integrity of the interface design making the GUI less intuitive for users.

3.4 Crazyflie GUI Use Case Diagram

The diagram bellow shows how the Crazyflie GUI may interact with the rest of the system if implemented. It also shows what functions may occur within the interactions between Vicon System, Crazyflie drone and the user.

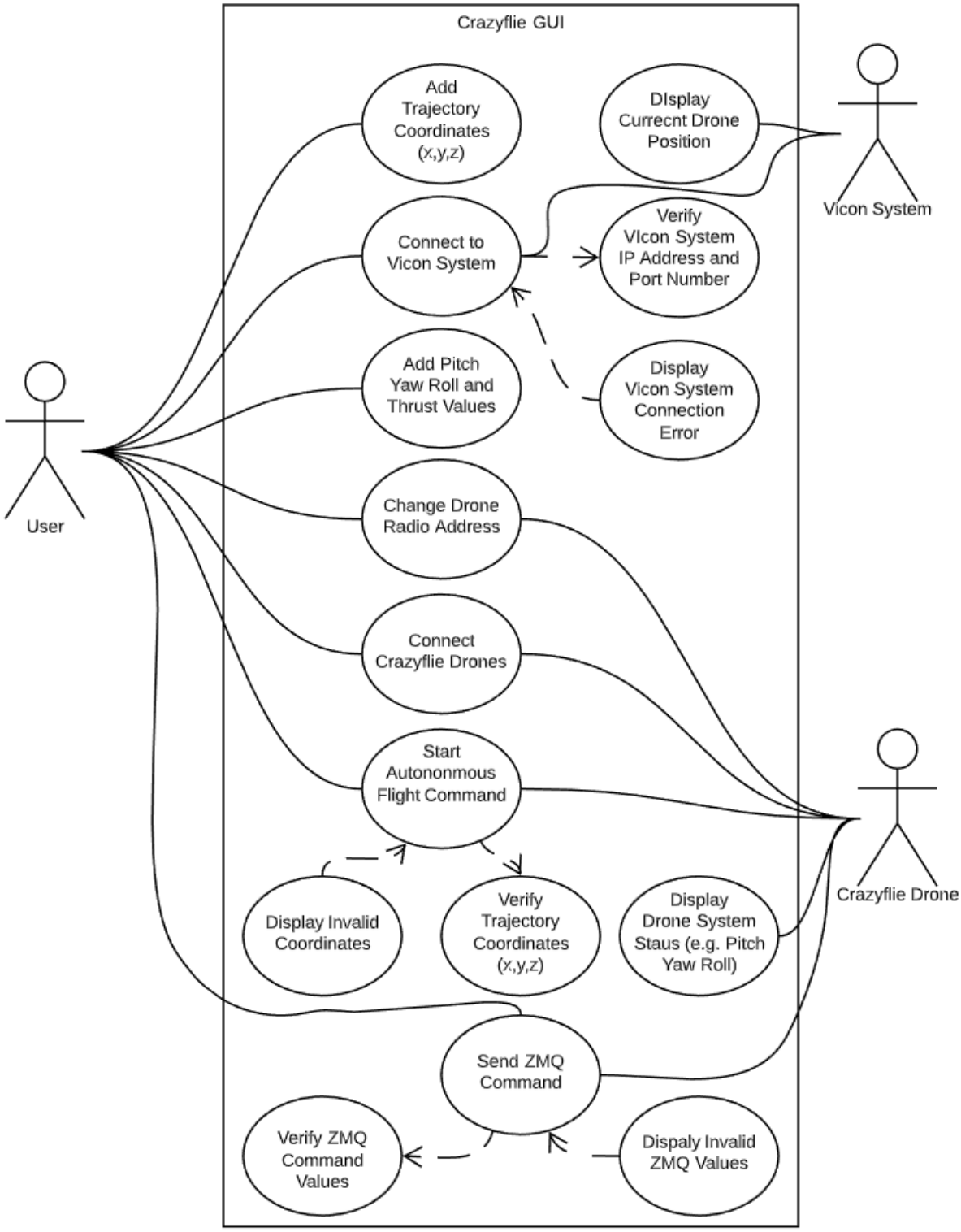

Figure 3 Crazyflie GUI Use Case Diagram

4 Theoretical Development of Emergent Swarm

As explored in the study on swarm topologies by Bruno Luis Mendivez Vasquez, 2018 and Jan Carlo Barca, 2018; there are many methods of invoking emergent behaviour. Nevertheless, while the methods may vary, the overall objective of these methods are largely the same; to maintain a cohesive swarm structure whilst completing an objective.

These emergent behaviours will be derived from a set of simple conditions that that individual drones must abide by. Such conditions may involve parameters concerning distance from neighbouring drones and velocity. This falls in line with the current philosophy of how swarms interact; that complex swarm behaviours emerge from simple rules adhered to by a group of locally interacting agents (Spector et al., 2005; Reynolds, 1987).

4.1 Theoretical Requirements

The most important main requirement is to simulate the movement of a drone using a PID controller. The PID controller will be responsible for calculating the drone's velocity using an error. This error will be derived from the distance between the drone and its destination. The idea behind the PID controller is that by measuring the change in error over time, the drone's velocity will change depending on how great that change in error is. In short, the greater the distance, and subsequently the error, the greater the drone's acceleration. On the other hand, as the distance, and therefore the error, decreases, the drone will begin to decelerate.

Before scaling up the number of drones, in order to simulate the function of certain embedded sensors used by autonomous drones, the simulated drones will only be able to detect other drones within a certain radius. Once a drone detects the presence of another drone that is within radius, the PID controller will take into account the trajectory and distance between them.

Once the PID controller is able to regulate the velocity of a drone moving to a single trajectory, the same PID configuration should be able to maintain a specified distance between at least two drones whilst still moving towards an objective. This will be the beginning stages of forming a cohesive swarm structure.

In theory, adding a detection radius to the drones may allow swarm separation to occur as an emergent property. If the swarm's path was in some way interrupted by an obstacle, the swarm may be forced to split into separate flocks. Similar emergent behaviour was achieved in the flocking simulation conducted by G. Vásárhelyi in 2018.

In nature, for example a shoal of fish, one of the ways in which evolved swarm behaviours could protect a group from predators could be increasing a swarm's effective vigilance (Treherne, 1981). This may suggest that not only are there simply more eyes looking out for predators, but fish are also aware of the changes in behaviour of other fish in the shoal. Another potential benefit of swarm behaviour could be distributed information processing abilities (Couzin, 2009). When a predator is near, other fish may recognise the erratic behaviour of their neighbours and decide to react. This effectively sends a message throughout the swarm, notifying others of a threat and roughly which direction the message, and therefore the threat, came from. The same principle could be exhibited by the simulation's emergent behaviour. If a drone detects something within its radius that it may want to move towards or away from, the swarms desire to maintain cohesion may cause the swarm to effectively follow that drone as if they were a leader. Theoretically, the more drones that move towards a similar visual trajectory, subsequently increasing the number of leaders, the more influenced the swarm will be to move in that similar direction.

In order to better maintain swarm cohesion, the PID controller must be fine turned in such a way that the drones are able to adapt to a change quickly enough whilst minimising overshoot. Too slow of a reaction time is likely to cause its neighbours to move out of detection radius, decreasing cohesion within the swarm and fracturing formations. Moreover, a PID configuration that is too quick to adapt is prone to overshoot, likely resulting in the collision of drones and their neighbours.

If the simulation is able to adhere to most of these requirements, it must continue to perform after white noise is added to the necessary values used in the PID controller. This will give insight into how the swarm may perform if the information came from physical systems that must interact with noise from their environment such as proximity sensors and ultra-sonic range finders.

4.2 Design Constraints

When the decision to develop a simulation was made in response to the university campus being off limits, time suddenly became a notable constraint. To test some of the theoretical requirements would demand some secondary objectives to be met. However, with the limited time left for development, achieving secondary objectives may prove challenging.

As these simulations will be run in real time and the number of drones increase, it is possible that a decrease in system performance may occur, decreasing the speed of the simulation. This is in part due to the simulation being run on a laptop with limited performance and fitted with a mobile cooling solution. This bottleneck may impact the ability to judge swarm performance on a time basis. To mitigate this, it may be reasonable to limit the swarm to a manageable size to allow the system to run the simulation smoothly. On the other hand, measuring a swarm topology's capabilities whilst utilising a limited swarm size may provide less reliable results.

5 Design

Much of the design for the aspects of this project involving the Crazyflie API is inspired by the project completed by Ieaun Roberts. Much of the work he completed lays the foundation for the systems required to control a Crazyflie drone. This study will focus on scaling the design for controlling multiple Crazyflies.

5.1 Overall System

The class diagram below details the structure and data flow of the overall system. It serves to highlight functions involved in the process of sending commands to multiple drones. One thing to note is that the Vicon system sends the data for each drone in a single packet. This allows the Crazyflie API to match a Crazyflie's radio address with the object ID in the data packet.

The functionality of various sections in this class diagram will be elaborated on further in the document.

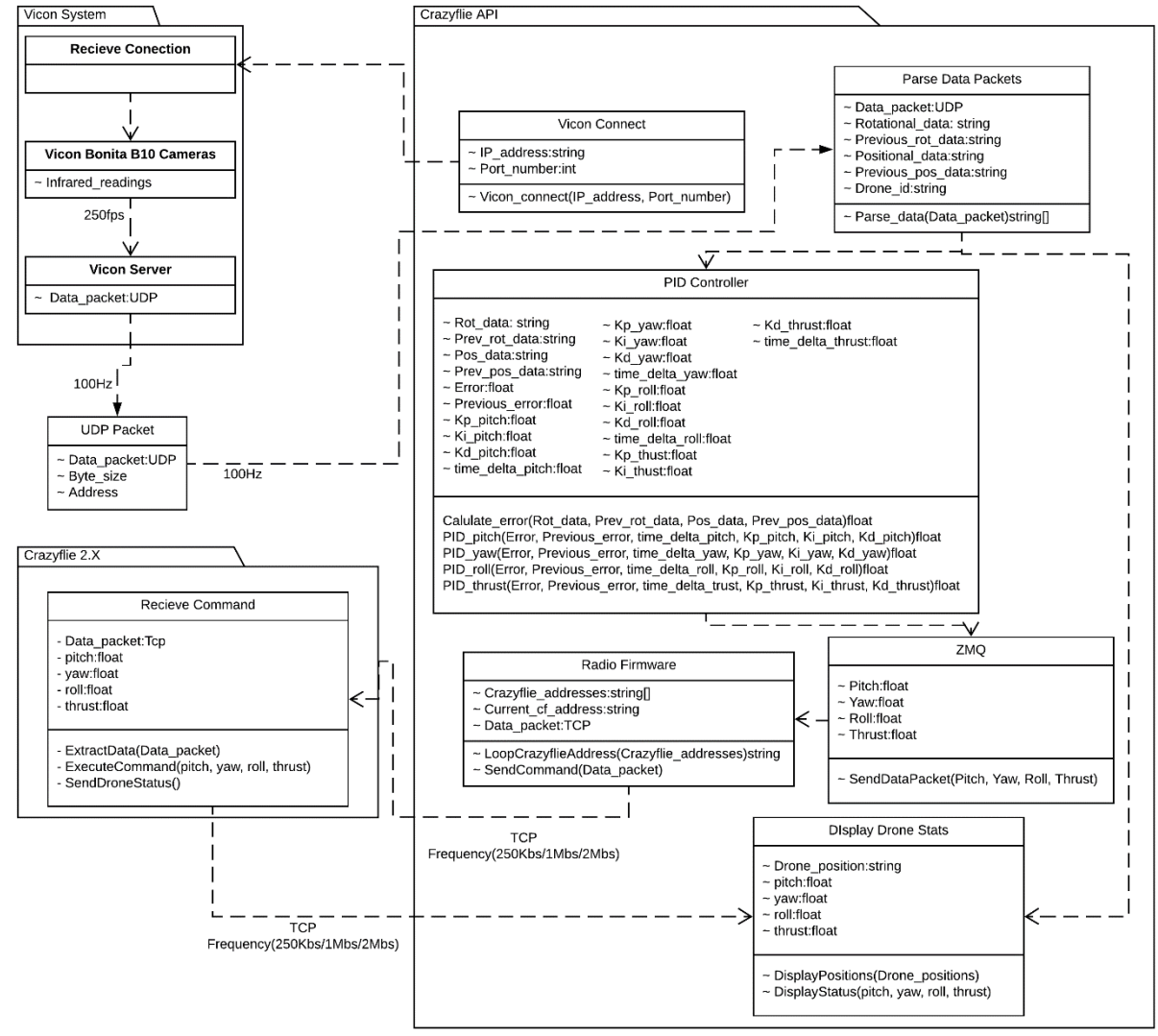

Figure 4 Crazyflie Swarm Class Diagram

5.2 Methodology

Due to the different aspects of development involved when working with the Crazyflie API, it became necessary to utilise multiple methodologies. Initially, the Evolutionary Prototype Method was considered for the development of the algorithms responsible for emergent behaviour. However, before developing these algorithms, functionality would need to be implemented into the Crazyflie API to allow said algorithms to operate effectively.

A more appropriate methodology for developing most of the Crazyflie API's functions would be the Waterfall Development Method introduced by Dr Winston, W Royce, (1970). As the functions required for developing the Crazyflie API can be easily compartmentalised into sections that will usually operate a single function, a linear rigid model consisting of sequential phases would be appropriate (Synopsys Editorial Team, 2017). Breaking down these components and developing them using a linear model makes the goal of each component easier to understand and manage. This aspect of the project's development will have clear and concise objectives whose requirements are unlikely to change in any major way.

The Evolutionary Prototype Method is still very much needed for improving on the emergent algorithms, PID controller and the intuitive design of the Crazyflie GUI. This method is a variation of the Prototype Method where a prototype is built and constantly reworked until acceptable performance is achieved, Sherrell L, (2013). This would allow a developer to gain a deeper understanding of how certain parameters would affect behaviour or performance. Additionally, wasting gratuitous amounts of time working on separate prototypes for each iterative change would not be necessary.

Regarding the emergent algorithms and the PID controller, considering many of the project's primary objectives focus on requirements that necessitate fine tuning, the Evolutionary Prototype Method would be appropriate. A supporting factor in this decision is that the nature of emergent properties means that incremental optimisations are crucial when attempting to make progressive improvements. Other methodologies, on the other hand, would be less suited as they often focus on defined requirements that do not require significant amounts of iterative improvements. Moreover, the Evolutionary Prototype Method is well suited for projects that may have well defined requirements, but the final solution, results or output may be unclear. For instance, the configuration of a PID controller may produce admirable results that correspond with the project's requirements. Even so, improvements to responsiveness and accuracy could still be made, although, the requirements for such improvements may be unclear.

The table below details which Crazyflie API functions will utilise which development methods. These are functions that are not included in the Crazyflie API so they must be implemented. Most of the functions displayed require the Waterfall Development Method. However, the Evolutionary Prototype Method is necessary for the more experimental aspects of the project crucial for the successful autonomous control of the Crazyflie drones. This is because of the fine tuning required to achieve optimal performance.

Functions	Methodologies
Vicon Functions	
Connect to Vicon System	Waterfall Method
Receive Data Packets	Waterfall Method
Parse Data Packets	Waterfall Method
Ping Vicon System	Waterfall Method
Crazyflie Drone	
Send Pitch, Yaw, Roll and Thrust Radio Command	Waterfall Method
Receive Drone System Diagnosis Data	Waterfall Method
Radio Functions	
Loop Through Crazyflie Radio Address	Waterfall Method
Autonomous Drone Control	
Calculate Distances to Trajectories and Drones	Waterfall Method
Calculate Vectors to Trajectories and Neighbouring Drones	Waterfall Method
PID controller	Evolutionary Prototype Method
Emergent Behaviour Algorithms	Evolutionary Prototype Method
Crazyflie GUI	
Display Drone Diagnosis Data	Waterfall Method
Display Drone's Current Position	Waterfall Method
Display System Messages in Console Window	Waterfall Method
Input Trajectories for Drone Movement	Waterfall Method
Input Drone Pitch, Yaw, Roll and Thrust Commands	Waterfall Method
Input IP Address and Port Number to Connect to Vicon System	Waterfall Method

Table 2 Methodologies for Different Functions

5.3 Hardware

This section of the report will detail the specifications of the hardware crucial to the functionality of the Crazy Swarm and how they may be integrated.

5.3.1 Crazyflie Drones

Initially this project was to use the Crazyflie 1.0 drone released in 2013. However, this model was quickly discontinued following the release of the Crazyflie 2.0 in 2014. The incredibly short-lived lifespan of the 1.0 meant that academic support, as well as adoption in the opensource community, was incredibly limited. This was made apparent during the research stage of development. Many resources available online mostly referenced the Crazyflie 2.0 and 2.1 models, mostly referred to as 2.X. This presented an issue when developing the 1.0 firmware as it differs in structure and capabilities to the 2.X. Developing the firmware for the 1.0 would have been unreasonable with the minimal documentation available online. Therefore, the 2.1 model, released in 2019, was acquired to mitigate this issue; as the 2.0 and 2.1 drones share the same firmware, plenty of research papers and documentation exist as resources to aid in development.

The Crazyflie 1.0 and 2.1 drones also differ greatly in hardware specifications. Whilst compared to the 2.1's 27g, the 1.0 may weigh less at only 19g, however, the 2.1 is fitted with improved 4 x 7mm coreless DC-motors that give a maximum take-off weight of 42g. This allows the drone to fly more aggressively when needed, allowing for quicker responsiveness and torque (Bitcraze, 2019). This is opposed to the 1.0's weaker 1 x 6mm DC-motor. The extra carry weight of the Crazyflie 2.1 will also help the drone's manoeuvrability when carrying the mo-cap markers necessary for position tracking.

Crazyflie 1.0	**Crazyflie 2.1**
	
1 x 6mm DC-motor	4 x 7mm DC-motor
Weight: 19g	Weight: 27g
Dimensions: $9mm^2$	Dimensions: $9mm^2$
Flight Time: 7 minutes	Flight Time: 7 minutes
Expansion Deck: No	Expansion Deck: Yes

Table 3 Crazyflie Drone Generation Comparison

5.3.2 Vicon System

The four Vicon Cameras being used for tracking the position of the drones are the Bonita B10. Each Camera would be situated at the 4 corners of the room being used for the drone's air space. Using the Tracker software, the Vicon Cameras will be able to track the position of the 3 retroreflective mo-cap markers attached to each drone using infrared light. Three mo-cap markers are necessary for each object being tracked as it is the minimum number needed for the Tracker software to be able to recognise the transforming position of an object. When tracking an object, the Cameras will be able to relate the distance between each mo-cap marker. This will allow the Vicon System to track multiple objects minimal confusion as the markers are cross referenced with objects marker distance information (Roberts, 2019).

Once the Vicon System is able to track the positions of the Crazyflie drones, the position data will be streamed back to the Crazyflie API. Here, the PID controller can utilise this data to attempt to correct the drone's flight path by obtaining any deviations in trajectory.

5.4 Software Design

This section will serve to outline the theoretical system architecture and implementation level surrounding the Crazyflie swarm and the Crazyflie GUI. It will also serve to detail the software design for the emergent drone swarm simulation and how it will serve as a digital twin for the Crazyflie swarm.

5.4.1 Crazyflie API

The Crazyflie platform developed by Bitcraze was a prime platform to develop on with the support of its widespread opensource adoption and plethora of development tools. Many features also exist within the Crazyflie API that make development and debugging more intuitive (Bitcraze, 2019). This includes:

- Bootloading firmware via the radio
- A parameter framework to edit and read data via the radio
- A framework for logging data via the radio
- Lots of left-over CPU cycles as well as flash and RAM free to use
- JTAG interface
- An expansion interface to attach new hardware (Crazyflie DECK)

The Crazyflie API comes in the form of a modified distribution of the Linux operating system. The API offers many tools and resources used to modify many aspects of the API as well as multiple IDEs (Integrated Development Environment) to interface with the API's firmware.

Firmware	IDE	Language
Crazyflie Client	PyCharm	Python
Radio Firmware	Eclipse	C
Crazyflie firmware	Eclipse	C

Table 4 Crazyflie API Languages and IDEs

5.4.2 Oracle VM VirtualBox

VirtualBox is an opensource application developed by Oracle that allows operating systems to be run as a virtual machine within an operating system (Oracle, 2007). It supports a large number of guest operating systems, however, for the purposes of this project it will be used to run the Crazyflie API which is where the Crazyflie swarm is to be developed.

5.4.3 Crazyflie Radio Firmware

The Crazyradio PA 2.4GHz USB dongle will be used to send commands to the Crazyflie drones and receive the drone's diagnostic data. With a clear line of sight, the radio signal range can extend up to 2km with the Crazyflie 2.X which is more than enough for the allocated airspace. This range also makes it a perfect alternative to Wi-Fi as depending on the Wi-Fi configuration, most networks may not have the required bandwidth or range (Bitcraze, n.d.).

Whilst the Crazyradio is perfectly capable of sending and receiving signals from a single Crazyflie drone, there is currently no official support for communicating with multiple Crazyflies with a single Crazyradio. Nevertheless, many researchers and enthusiasts, such as Yifan Zhang, have attempted to modify the Crazyradio firmware to allow it to loop through each drone's unique URI (Uniform Resource Identifier) to send each drone its own command. Each drone has its own URI to interface with the

Crazyradio by telling the system the USB dongle number, radio channel and radio speed (e.g. radio://0/10/250k) (Bitcraze, 2020).

According to a 2013 blog post on the Bitcraze Forums by a Bitcraze employee known as Arnaud, there are at least two ways of controlling many Crazyflies with only one computer. The first is using one Crazyradio per Crazyflie where each drone has its own channel. The second is using only one crazy radio but setting up each drone with different addresses (Arnaud, 2013). At the time of researching methods for multiple Crazyflie control, only one Crazyradio was able to be acquired. As a result, the single radio option became necessary.

Arnaud (2013) stresses, whilst the radio chip has addressing capabilities, it currently always uses the default address. One would not only need to give each drone a unique URI but set up a data transfer loop in the python radio link object to handle multiple Crazyflies. This would mean that the channels and addresses would need to be specified between data packets being sent.

When modifying the Crazyflie's address through the Crazyflie client, a config file located in "//home/bitcraze/.config/cfclient" saves the drones address as a variable called "link_uri" for the Crazyradio firmware to read as a default parameter. Theoretically, another variable ("link_uri2") containing another Crazyflie's address could be created for the loop inside the Crazyradio firmware to utilise.

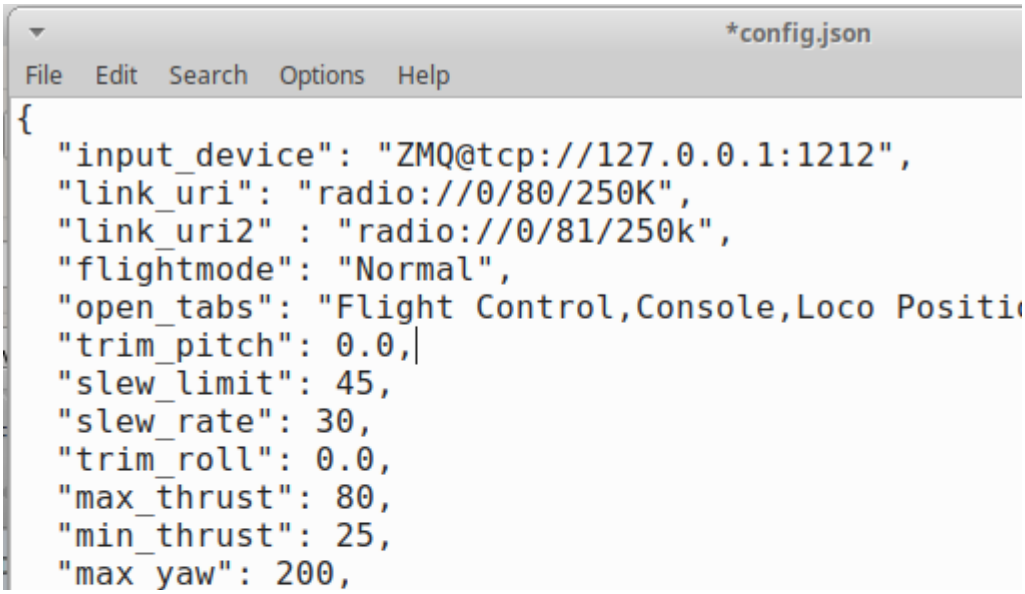

Figure 5 Crazyflie Link URI Example

Other more recent blog posts and articles discussing similar endeavours also cement the concept of utilising a loop for control of multiple Crazyflies (Kimberly, 2019; Hoenig, Preiss 2018).

Theoretically, depending on how many Crazyflies are communicating through a single radio, the latency in communication may be substantial enough to warrant using more than one radio. In terms of Big-O complexity, potential latency could be $0(n)$ where n is the number of Crazyflie drones.

5.4.4 PID Controller

The PID Controller is a mathematical approach to controlling the output of a system whether it be for climate control, a valve controlling the flow of water or a robotic arm (Maldini, 2018). Whilst there are many different types of control systems out there, PID controllers excel at stabilising the translational and rotational motion of an object with six degrees of freedom (Najm & Ibraheem, 2019).

The PID controller is not only a relevant component in both the Crazyflie swam and the emergent swarm simulation but plays the important role of maintaining a drone's trajectory. The PID controller for the simulation, however, will serve as a digital twin to demonstrate the potential performance of the controller's configuration.

A major difference between the utilisation of the PID controller is that in the simulation, one controller will be used to manage the drones force in a given direction. On the other hand, in the Crazyflie swarm, four PID controllers would be used to manage a drone's pitch, yaw, roll and thrust. This would require the configuration of the PID's parameters (K_p, K_i, K_d) as well as time (t) for each of the Crazyflie's four parameters totalling in sixteen PID parameters (Salih et al., 2010).

PID Parameter	Symbol
Proportional	K_p
Integral	K_i
Derivative	K_d
Error	e
Time	t
Output	u

Table 5 PID Parameter Symbols

The PID controller operates by calculating the sum of the proportional, integral and derivative of an error. These calculations take into account the previous error as well as the time delta to essentially figure out how much the error has changed within a given amount of time. Depending on how much has changed, the PID output can either increase or decrease (Chien, 1988).

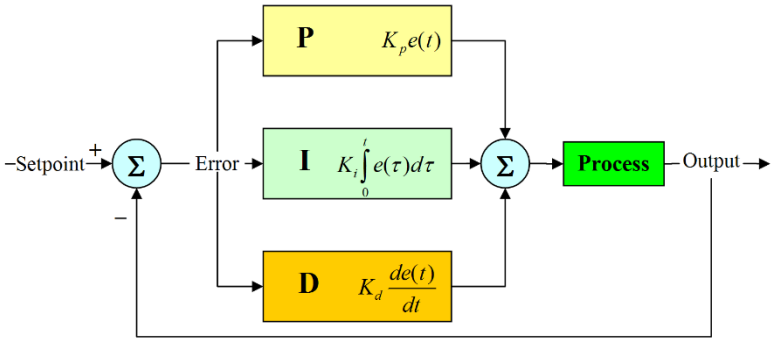

Figure 6 PID Controller Diagram (Maldini, 2018)

The Plant, or in the case of the diagram above, Process, is the system the PID controller tries to regulate through actuating signals (input) to produce a viable controlled variable (output). The PID controller will continue to do this until $e = 0$. The equation bellow details how the output is calculated.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(e)}{dt}$$

Equation 1 PID Output

5.4.4.1 Proportional Control

The proportional controller functions by looking at the present error to calculate how large the output should be and how quickly it should change. This is achieved by multiplying the present error by the gain value K_p. In other words, the initial acceleration of the drone highly depends on the value of K_p. The greater the gain the quicker the acceleration. As the error decreases however, the proportional output decreases causing the acceleration of the drone to decrease. The pseudo code below shows how the proportional control is implemented.

$$proportional\_output = gain * error$$

One detrimental characteristic of proportional controllers is that multiplying the error by a gain means the error will never truly reach zero. Instead, the error will continue to asymptotically decrease getting closer and closer to zero (Douglas, 2012). This phenomenon is called steady state error, the only way to mitigate this is by utilising controller capable of acknowledging the past error.

An example of steady state error can be seen in an example provided by Douglas (2018). If a Crazyflie drone was told to reach a certain altitude using only a proportional controller, once the altitude was reached, the drone would begin to fall as the propellers would have stopped spinning. As the Crazyflie's altitude decreases, the error increases causing the propellers to spin in an attempt to once again reach the desired altitude. To maintain altitude requires a controller capable of finding the correct propeller RPM (Rotations per Minute) to allow the Crazyflie to hover.

5.4.4.2 Integral Control

Adding an integral controller allows a control system to overcome steady state error. The integral controller keeps a summation of previous errors over time allowing it to keep a memory of the system's change in error. A main difference in functionality between the proportional and integral controllers is that once the system reaches steady state, as long as the error is not zero, the integral value will continue to change. In the context of a drone reaching the desired altitude, this allows the propellers to continue spinning. After the drone has reached steady state, the proportional controller contributes less to the overall output, whereas the integral value will continue to increase. Like the proportional controller, the integral controller continues to change until the $e = 0$. Below is the pseudo code for calculating the integral output.

$$integral\_output = gain * time\_delta * error$$

One disadvantage is that, due to the integral constantly increasing if an error is present, the integral output may sum to a value greater than necessary resulting in an overshoot (Douglas, 2018). This overshoot creates a negative error. In the case of propeller control in a drone, the integral may have outputted an RPM value larger than necessary, causing the drone to fly past its destination. The amount of overshoot depends on the gain and time delta value (K_i, t).

Integral windup is another potential issue. This is caused when the error does not change for a prolonged period of time, causing the integral to accumulate a larger error sum (Beatica, 2015). This can be visualised by a drone attempting to fly towards a destination but is anchored by a piece of string. The integral continues to sum the error causing the output to increase, increasing the RPM of the propellers. When the string is detected, the drone accelerates towards its destination. However, the high initial speed of the drone causes it to overshoot its target. Furthermore, whilst the overshoot may cause the integral to subtract the negative error, the drone will continue to overshoot until the integral value decreases from the negative error. This means the greater the integral windup, the greater the overshoot (Roberts, 2019).

There are several methods to mitigate integral wind up (Shin & Park, 2012):

- The maximum RPM of the drone can be limited or clamped so that they do not exceed a certain value, for example the propellers cannot exceed 200RPM.

$$if\ RPM\ >\ 150$$
$$RPM = 150$$

- A limit on the maximum value of the integral output can be put in place to limit the acceleration of the drone.

$$integral\_output = gain\ *\ time\_delta\ *\ error$$
$$if\ integral\_output > max\_val$$
$$integral\_output = max\_val$$

- Another method involves multiplying the integral value by a bias to prevent the integral from continually increasing.

$$bias = 0.9$$
$$integral\_output = gain\ *\ time\_delta\ *\ error$$
$$integral\_output\ *= bias$$

5.4.4.3 Derivative Control

To mitigate the any potential overshoot by the integral controller, the derivative controller is used to predict the future and respond to how fast the error is changing (Douglas, 2018). In a drone, the rate of error change is representative of how quickly the drone is moving towards its destination. By looking at the rate of change in the error, if the error is decreasing too fast, the derivative controller can produce a negative error causing the PID output to decrease. Therefore, the value of the derivative output is dependent on the error's rate of change. In other words, if the drone is approaching its target to quickly, the derivative controller recognises this and decelerates the speed of the propellers.

$$derivative\_output = (error\ -\ prior\_error)/time\_delta$$

5.4.4.4 Implementation

Regarding the Crazyflie swarm, Roberts (2019) discusses the appropriate location to implement the PID controller. He states that the Crazyflie Client has the benefit of faster computational speeds due to the Crazyflie's onboard 168Mhz Arm Cortex-M4 processor and 192kb of SRAM. This is in contrast to the more powerful CISC based chips found in most laptops and desktops. In addition to this, if the PID controller were to be implemented in the Crazyflie, it would need to receive the data from the UDP stream containing the positions of itself, the destination and the positions of the other drones in the swarm. This would result in more data needing to be sent to each Crazyflie as opposed to just sending $\theta_c, \varphi_c, \Omega_c$ and ψ_c values; this could also potentially decrease the speed of data transmission.

As for the emergent swarm simulation, the PID controller will be implemented as a C# script for the emergent algorithms to inherit.

5.4.5 Crazyflie GUI

The graphical user interface will be implemented into the Crazyflie Client and created using the QT designer software packaged with the Crazyflie API. The GUI will be saved as a ".ui" file and placed in the folder location "home/bitcraze/projects/Crazyflie-clients-python/src/cfclient/ui/tabs". A python script containing the code for the virtual visual trajectory will be linked to the ".ui" file. A file called "_init_.py" in the same folder location, however, must be edited to enable the new script, allowing the GUI to be viewed in the Crazyflie Client.

The image bellow shows the Vicon Movement tab being enabled from the "_init_.py" file whilst another image shows the tab being selected from the Crazyflie Client.

```
#from .ConsoleTab import ConsoleTab
from .ExampleTab import ExampleTab
from .FlightTab import FlightTab
# from .GpsTab import GpsTab
from .LEDTab import LEDTab
from .LogBlockTab import LogBlockTab
from .LogTab import LogTab
from .ParamTab import ParamTab
from .PlotTab import PlotTab
from .locopositioning_tab import LocoPositioningTab
from .QualisysTab import QualisysTab
from .ViconmovementTab import ViconMovementTab

__author__ = 'Bitcraze AB'
__all__ = []

available = [
    #ConsoleTab,
    ExampleTab,
    FlightTab,
    # GpsTab,
    LEDTab,
    LogBlockTab,
    LogTab,
    ParamTab,
    PlotTab,
    LocoPositioningTab,
    QualisysTab,
    ViconMovementTab,
]
```

Figure 7 _init_.py Script

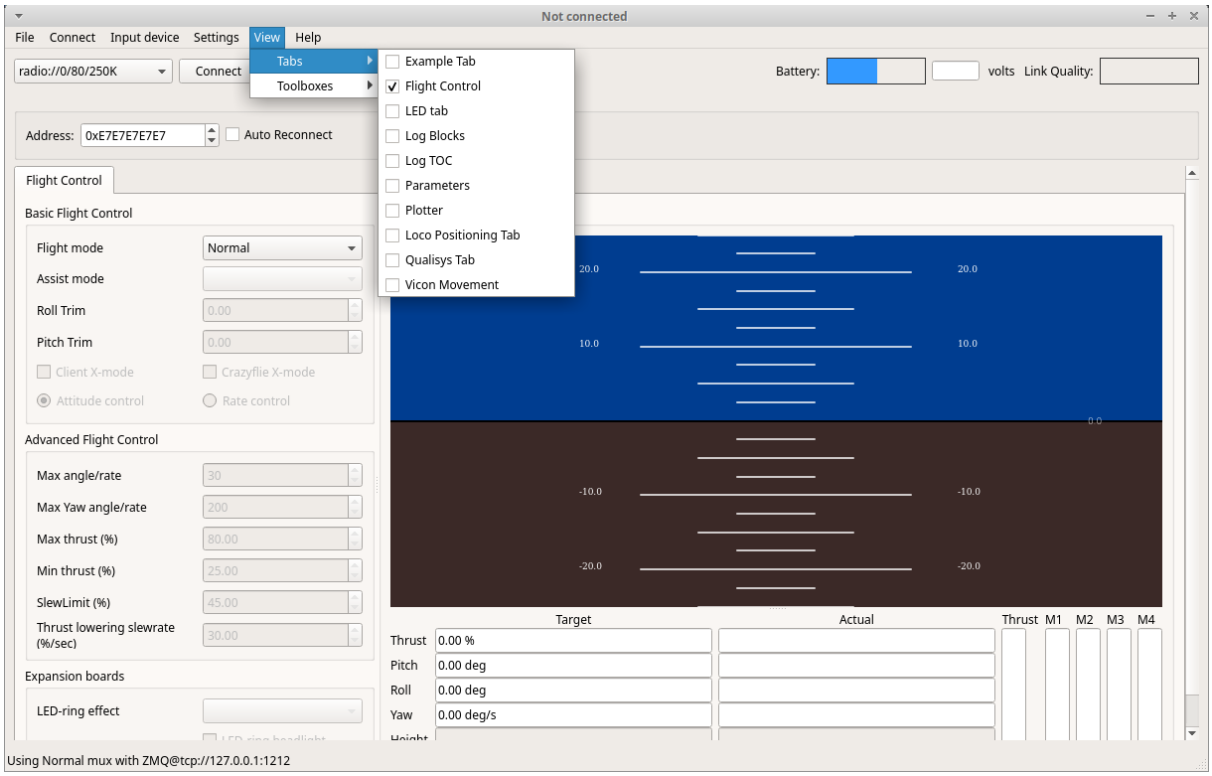

Figure 8 Enable Tabs in Crazyflie Client

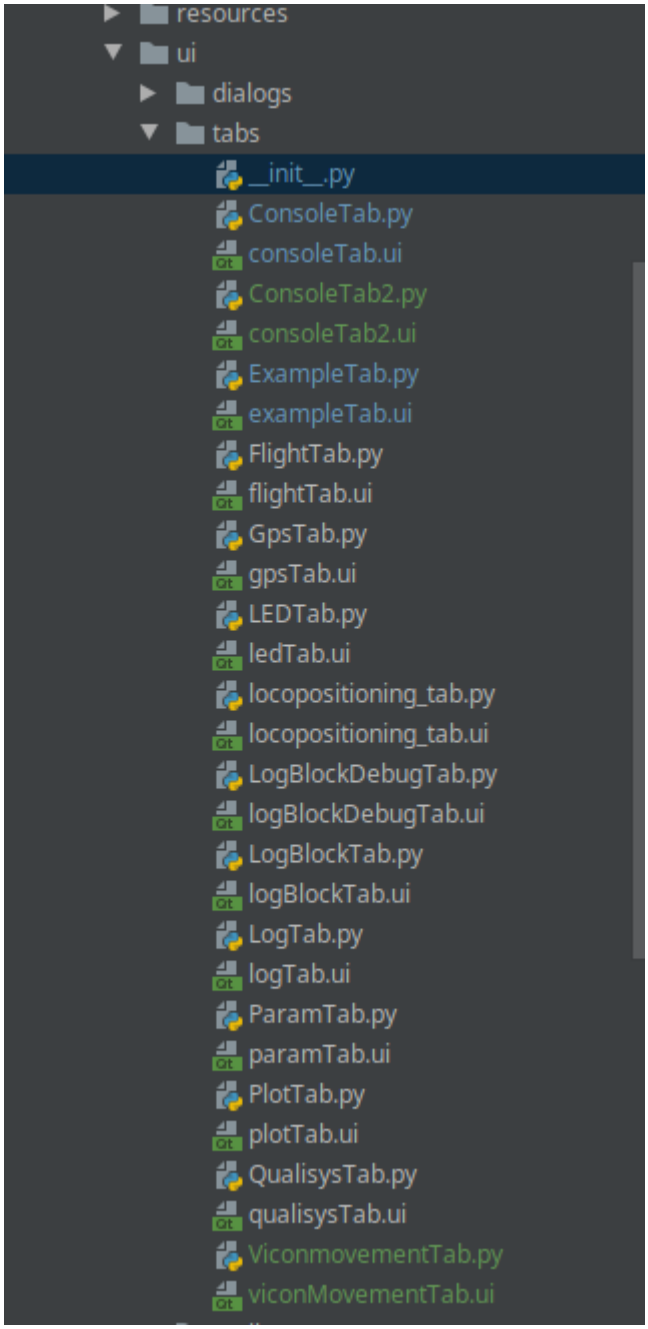

Figure 9 Tab File Structure

5.4.6 ZMQ

The ZMQ messaging library is the most widely adopted method of sending and receiving data packets between the Crazyradio and Crazyflie drones within the Crazyflie API. It makes use of the TCP/IP protocols; this ensures that the data packets being sent are received by the recipient, otherwise if a packet is lost, the system will continue to attempt to deliver the packet.

For the purpose of this project, ZMQ will be used to send the PID controller's outputs in the form of pitch, yaw, roll and thrust commands to the Crazyflie drones. By default, ZMQ client control is disabled in the Crazyflie Client config file located in the folder "//home/bitcraze/.config/cfclient". Simply changing this value to true enables ZMQ to be used as a client.

```
"enable_debug_driver": false,
"input_device_blacklist": "(VirtualBox|VMware)",
"ui_update_period": 100,
"enable_zmq_input": true,
```

Figure 10 Enable ZMQ Example

Doing this allows the drones to be controlled using backend code, after ZMQ is enabled ZMQ@127.0.0.1:1212 must selected from the input device menu (Kimberly, n.d.).

```
{
    "version": 1,
    "client_name": "ZMQ client",
    "ctrl": {
        "roll": 0.0,
        "pitch": 0.0,
        "yaw": 0.0,
        "thrust": 0.0
    }
}
```

Figure 11 ZMQ Example (Bitcraze, n.d.)

5.4.6.1 Implementation

As the method for utilising ZMQ functionality needs to be integrated into the Crazyflie Client, the method is implemented in the ".py" file relating to the Crazyflie GUI's ".ui" file in the aforementioned folder location "home/bitcraze/projects/Crazyflie-clients-python/src/cfclient/ui/tabs".

5.4.7 Emergent Algorithms

To invoke emergence, a set of rules must be established in order to dictate the desired behaviour. The emergent algorithms being developed are inspired by a more physicomimetic approach whereby virtual physical forces will continuously drive the swarm to a desired state, minimising the overall potential energy of the swarm (Spears et al., 2004). The potential energy references the movement of the drone losing energy as it approaches its destination; in a way, this behaviour characterises entropy. At the lowest state of energy, the drones have reached their destination and there is minimal movement throughout the swarm as the PID controllers work to maintain the drone's relative position.

The movement of the drones is physicomimetic in the sense that the drone's directional movement is directly influenced by the vector of interest (Spears et al., 2004). This is opposed to more biological algorithms where simulated modulations in individual behaviour are modelled; such modulations in nature would be caused by disturbances from the environment or the swarm itself (Garnier et al., 2007).

Two main principles that affect the behaviour of a drone when in proximity of another drone are:

- If a drone is too close, vector in the opposite direction of the drone.

$$if\ drone\_distance < minimum\_distance$$

$$movement(-drone\_vector)$$

- If a drone is too far away, vector towards the drone.

$$if\ drone\_distance\ > maximum\_distance$$

$$movement(drone\_vector)$$

- Whenever a drone is in between the minimum and maximum distances, the error is zero, therefore not movement will occur.

There are also characteristics that effect the swarm's topology:

- In a physical system, the sensors on a drone may only be able to recognise a certain number of drones within a radius due to limitations of the hardware.
- Swarm behaviour may be more effective if a drone only considers the distance and vector of the closest neighbouring drone.
- Calculating and only considering the average vector of neighbouring drones within a radius.

5.4.8 Emergent Behaviour Sequence Diagram

The sequence diagram bellow shows a basic outline of the logic used to invoke the physicomimetic emergent behaviour. The actor represents a neghbouring drone winthin the proximity of another drone in the swarm.

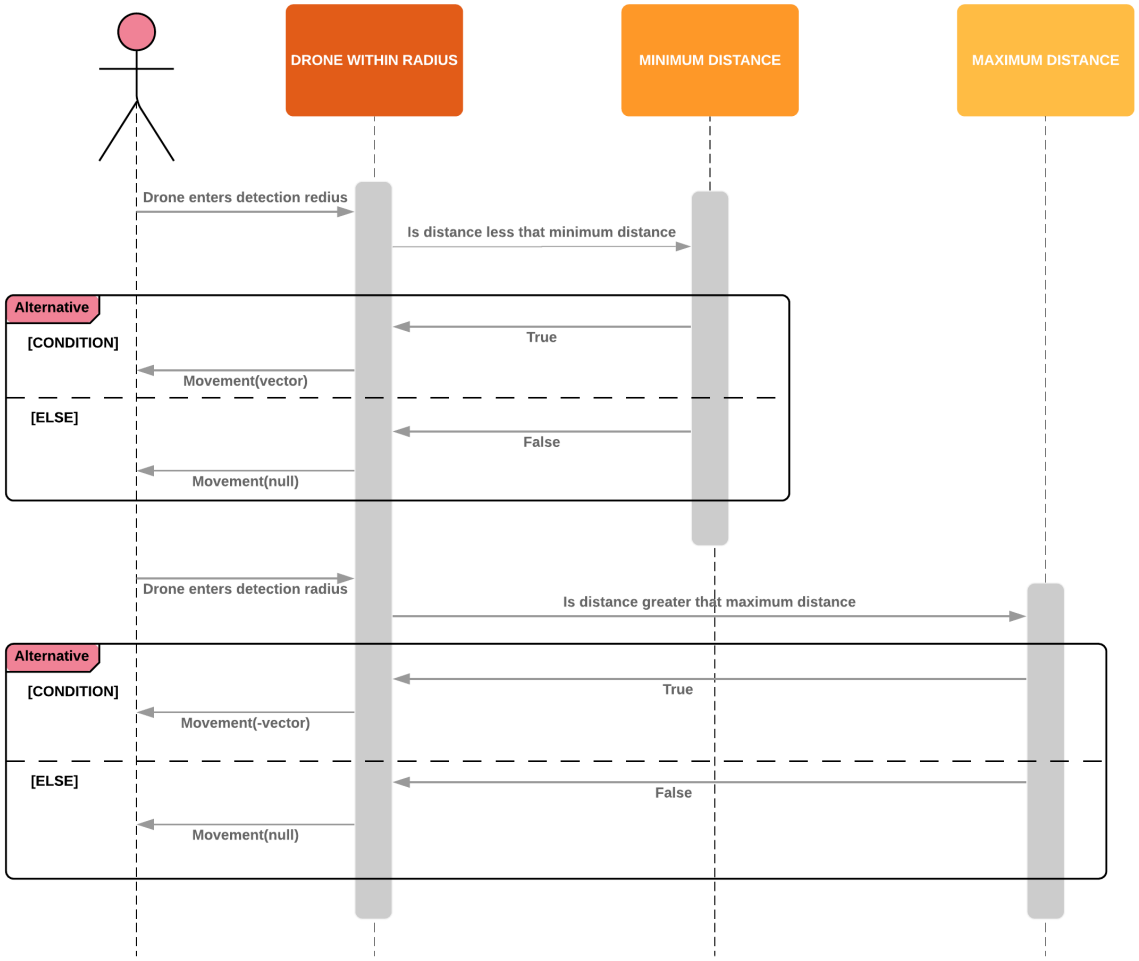

Figure 12 Swarm Logic Sequence Diagram

5.4.9 Box-Muller Gaussian Noise

5.4.9.1 Random Process

For the emergent swarm simulation, random values, also known as Gaussian or White Noise, will be added to the vector (x, y, z) values responsible for identifying the direction of neighbouring drones. Noise will also be added to the neighbouring drone's distance values. This simulates the interference and noise physical sensors and tracking systems would experience when taking readings from their environment. Testing the swarm under these conditions will aid in assessing the proficiency of the PID controller configuration and emergent swarm behaviours. To generate the Gaussian noise the Box-Muller method will be implemented.

The Box-Muller method generates random numbers in a normal distribution. This process uses a mean set to $\mu = 0.01$ and a standard deviation set to $\sigma = 0.1$. The method first generates one random number in a uniform distribution $(z1)$ between 0 and $2(\pi)$. Standard deviation is then calculated by using $B = \sigma(\sqrt{-2(\log(random\ number\ between\ 0\ and\ 1))})$. With the values of B and $z1$, two other random values ($z2$ and $z3$) are generated.

- $z2 = B(\sin(z1)) + \mu$
- $z3 = B(\cos(z1)) + \mu$

With the random numbers $z2$ and $z3$ generated, after each iteration of the Box-Muller algorithm either of the random variables is then added to aforementioned vector and distance values (Weisstein, n.d.).

The advantage of using the Box-Muller method is that, unlike randomly generated numbers, the values generated accurately represent the values being used. This method creates two normally distributed random numbers generated from one random number in a uniform distribution. The Gaussian Noise generated is able to conform around the distribution of the original values. In other words, the noise clusters around the average.

5.4.9.2 Mean Value

Regarding the mean (μ), adding a value would simulate the noise experienced within a physical system. This internal noise may be caused by system static or any naturally occurring vibrations. Therefore, for the sake of this simulation's accuracy, it too will be given a value during testing.

5.5 Experimental Design

To test the theoretical performance of the PID controller's configuration, graphing software, for instance, MATLAB and various python libraries, may be used to simulate the configurations characteristics. Overshoot, integral windup and the performance of their counter measures will be taken into account.

When assessing the performance of the swarm simulation, a way to analyse different configurations is by measuring how often the drones interact or how many of them interact during testing (Vasquez, 2018; Barca, 2018). The simulation is useful in the fact the one can visualise the overall cohesion of the swarm by seeing how well they maintain swarm structure. If drone's fragment from the swarm and other drones move out of detection radius, then the swarm structure has poor cohesion.

The Unity Engine has a feature that can create Debug Rays that allow the vectors and distances between the drones and the destination to be visualised. This offers a more intuitive way to visualise interaction whilst the swarm is completing an objective.

The above-mentioned methods of assessing swarm performance can also be used to indicate how the PID controller's configuration handles maintaining swarm structure whilst moving to a desired set point. How quickly the swarm moves to its objective and settles at the set point can also indicate how optimised the configuration is at managing acceleration and deceleration.

6 Crazyflie GUI Implementation

6.1 Initial Development

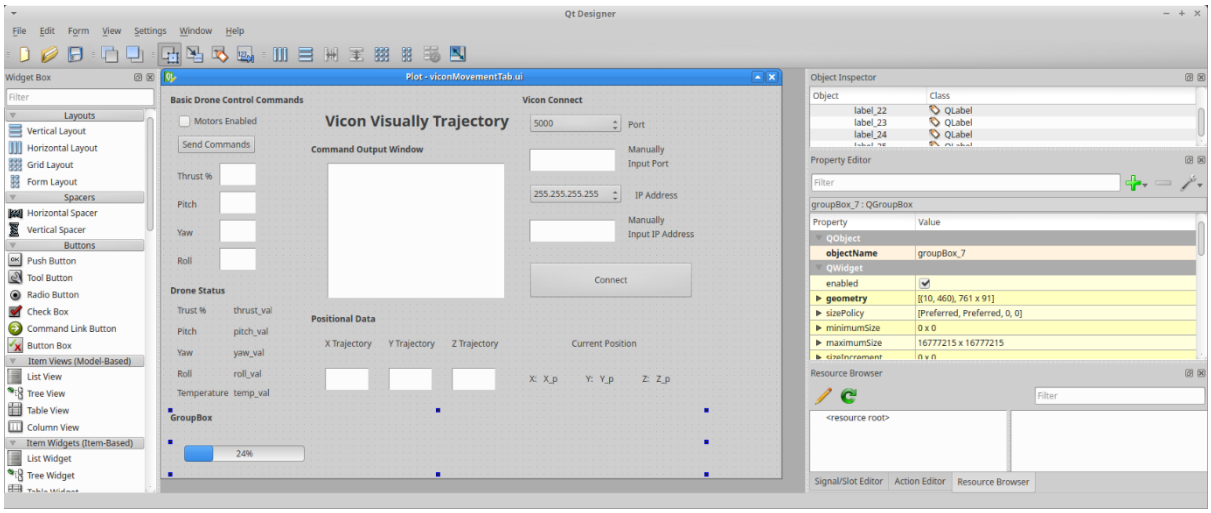

Figure 13 QT Designer Environment

An intuitive interface is imperative, especially when offering a plethora of options and controls. During the initial stages of development, the GUI elements for the base functions required for interfacing with one Crazyflie drone were implemented. Functions that shared a common use case were sectioned close together. The sections were separated as follows:

- Manual ZMQ controls
  - Here, the user will be able to insert their own thrust, pitch, yaw and roll values and send them as a command to a Crazyflie drone. The option to disable or enable the motors is also present. This can be used for debug purposes to test the connection and latency in a controlled manor.
- Displaying drone system status
  - Every time the Crazyflie Client receives a data packet from the Crazyflie, the GUI will display the drone's current thrust, pitch, yaw and roll and system temperature values in real time.
- Virtual visual trajectory data
  - In this section, the user can insert the (x, y, z) coordinates to give a Crazyflie a virtual visual trajectory. These coordinates must be obtained through the Vicon Tracker software to ensure the given location is within the scope of the Vicon Cameras. The Crazyflie's current position an also be monitored and is updated in real time.
- Vicon System connection options
  - This section contains text boxes where the user inserts the IP address and port number to connect to the Vicon System. Drop down boxes also contain a default IP address and port number so the user does not need to constantly retype them in.
- Command window
  - The command window displays system messages conveying the status of the system. This could include showing connected drones to displaying any system error messages. For example, entering a value over 100% for the ZMQ thrust command will display an error in the command window.

Other UI elements, such as the link quality and battery, were implemented into the Crazyflie Client developed by Bitcraze.

6.2 Edit PID Controller Values

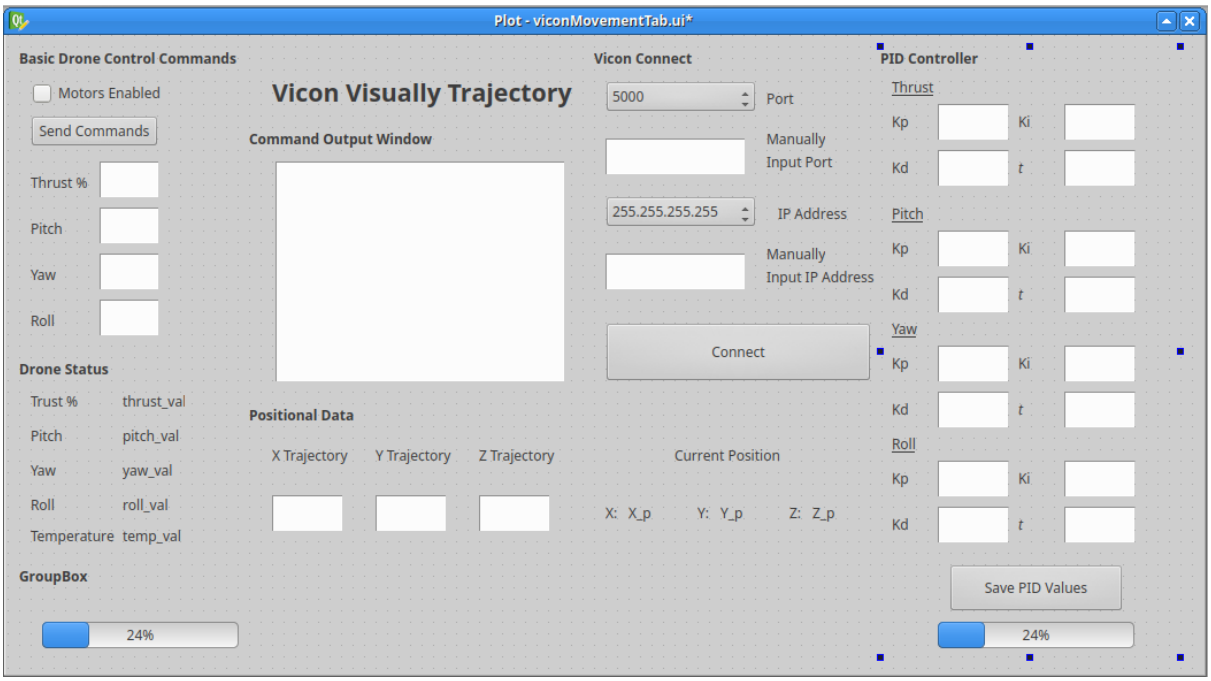

Figure 14 Crazyflie GUI with PID Editor

Fine tuning the PID controller when testing the Crazyflie drone would involve constantly going back to the source code to perform incremental tests. Moreover, after saving the changes to the source code, the Crazyflie Client would need to be restarted to recompile the script, loading the changes into the new instance of the client.

To make the process of editing the PID values simpler, a section in the interface was created. Here, the individual (K_p, K_i, K_d, t) values for thrust, pitch, yaw and roll can be changed within the interface during runtime without restarting the Crazyflie client.

At this stage in development, the edited values are not saved after the Crazyflie client is closed. Theoretically, the PID values could be saved in a text file when edited and read back in once the client is opened mitigating this issue.

After saving the Save PID Values button, the progress bar will indicate when the values have been saved and applied. Saving each value progresses the bar by 6.25%

6.3 Scaling Up GUI Elements

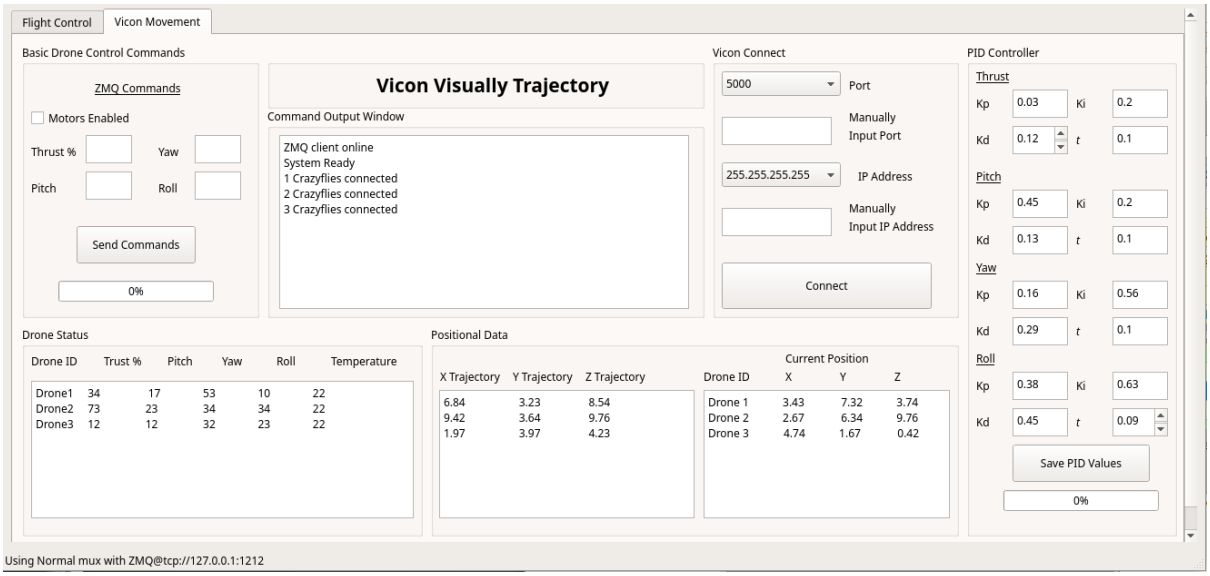

Figure 15 Crazy Swarm GUI

At this stage in development, the Crazyflie GUI only accounted for one drone. However, to the elements in the interface needed to be scaled depending on the number of drones in the swarm. As the contents of the text boxes can be read in by the Crazyflie API, a user's input can be parsed by line and spaces. Therefore, text boxes would serve as an appropriate tool to input coordinates. Each line would represent a set of coordinates with the (x, y, z) values separated by spaces.

Previously, the GUI was only able to show the current position and system status of a single drone. Now, however, with the implementation of text boxes, the current position and system status of each drone can be printed with each drone occupying a single line starting with the drone's ID.

If the size of a text box window cannot fully display all the information it contains at once, the data can be scrolled through. This ensures that the GUI can scale regardless of the number of Crazyflie drones.

6.4 Additional Implementations

The ZMQ section received a progress bar to show the progress of a command being sent. This is mostly for debug purposes in case there are any system errors that hinder the command from being transmitted.

7 Emergent Swarm Simulation Implementation and Testing

7.1 Implementation

7.1.1 Simulink PID Controller

A PID Feedback Controller was created using MATLAB's Simulink software. Simulink is an environment that uses Block diagrams to create simulations of control systems from multiple domains and model-based designs. Features like system level design and automatic code generation allow for continuous testing and verification of a wide range of embedded systems (MATLAB, n.d.). This environment allowed for the optimisation of the PID parameters responsible for controlling the drone's flight.

At first, the PID controller was to be simulated using a custom graphing software made using a MATLAB script. However, Simulink was likely a more reliable tool. With the advantage of useful supporting features, fine tuning the PID parameters was likely to be a more streamlined and accurate experience.

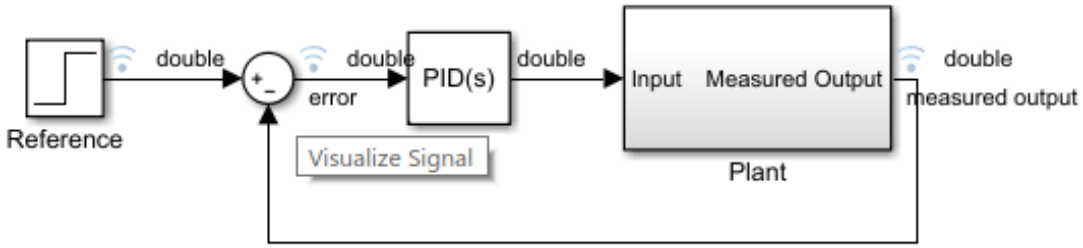

Figure 16 Simulink PID Controller

7.1.1.1 Components Involved

The Reference Block serves the function of giving the PID controller a desired value to get to. It is a simple step function where the reference value goes from zero to one. The PID controller will start from zero and attempt to reach the desired value over the course of 10 simulated seconds.

Figure 17 Reference Block

The Sum Block Parameter can add or subtract inputs depending on which input ports are connected. This is where the error is calculated so it's input can be utilised by the PID controller.

Figure 18 Sum Block

The PID Block is where the control system calculations take place. Here, the (K_p, K_i, K_d, t) parameters can be manually edited to fine tune the PID controller to optimise its output. One thing to note was that MATLAB's PID implementation includes advanced features such as anti-integral windup, external reset and signal tracking (MATLAB, n.d.). As the controller being implemented in the swarm simulation

may not include some of these advanced features, there may be discrepancies between the swarm's performances compared to the graph results.

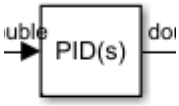

Figure 19 PID Block

The Plant serves to take the input and measure the output for later graphical observation. After taking said measurement, the Plant loops the output value back to the Sum Block parameter where the error is recalculated and fed back into the PID controller, restarting the cycle.

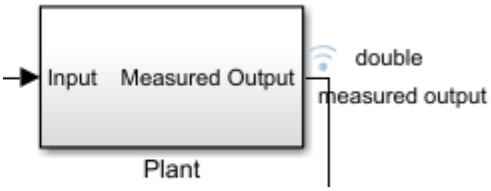

Figure 20 Plant

7.1.2 Create Unity Scene

An environment for the drones to occupy during testing needed to be created. This too had its purpose as rather than simply adding the drones into an empty scene, context was required to add depth to the environment. This worked to better visually represent the swarm's structure in $3D$ space.

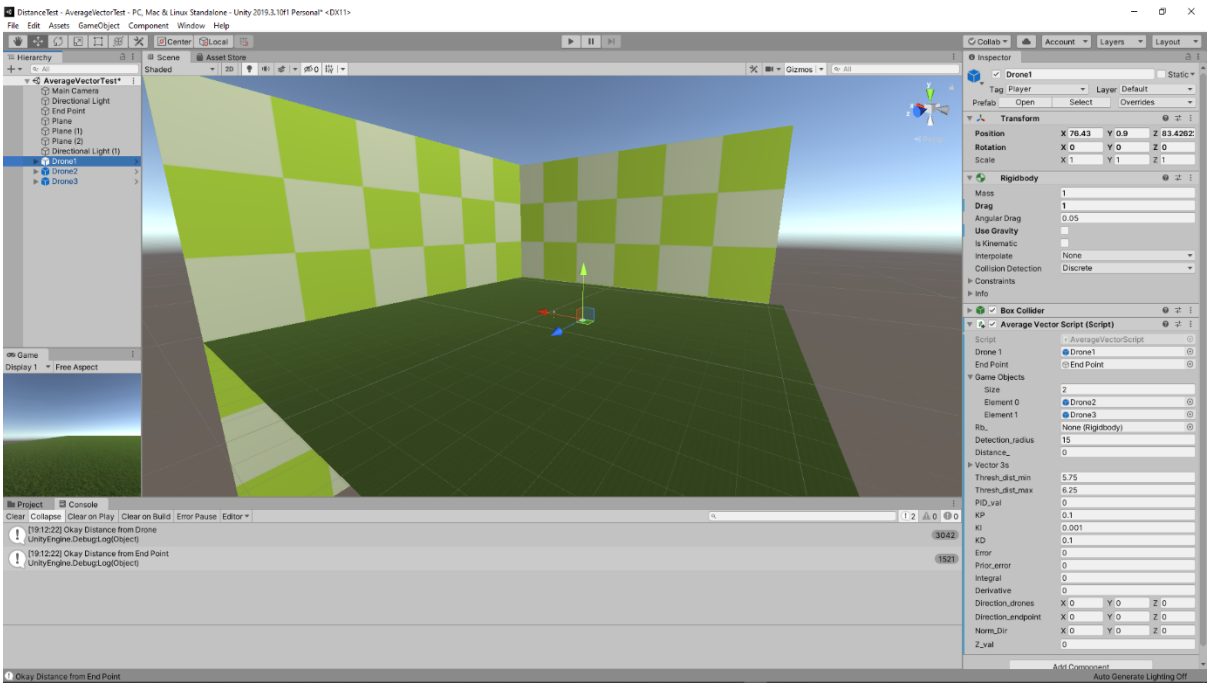

Figure 21 Simulation Environment

7.1.3 Swarm Topologies

Three separate topologies were theorised with the potential of invoking different emergent behaviours.

7.1.3.1 Average Vector

The average vector would be comprised of the vectors from a drone and its respective neighbours within detection radius. However, a separate PID controller handles the vectoring towards the desired destination. During initial simulations, factoring the destination vector into the average vector resulted in the drones also considering the average distance between its neighbours and the destination. This resulted in the drones settling at an average distance, whereas the intention is to move the drone to the desired location. To find the average, a for loop going through a Vector3 List is created whereby the summation of its contents is divided by the size of the list.

```
//Calculate average vector
foreach (var v in vector3s)
{
    Direction_drones += v;
}
Direction_drones = (Direction_drones / vector3s.Count);
```

Figure 22 Average Vector Code

7.1.3.2 Vector All

This topology serves a similar function to the average vector topology as the drones must eventually move in the average direction of its neighbouring drones. The topology uses a GameObject List containing the neighbouring objects in the swarm. This list is then looped through with the distance of each neighbour being fed separately into the PID controller. As only one drone is considered per iteration, the controller's output is used to only add force in the direction of the said drone.

7.1.3.3 Closest Neighbour

Instead of considering the trajectory of multiple neighbours, this topology only considers its closest neighbour within detection radius. This method could potentially reduce the amount of complex movements and interactions within the swarm which may increase the swarm's stability. As emergent behaviour is inherently comprised of simple rules, this topology aims to consider how simple said rules can become whilst still maintaining swarm cohesion.

7.1.4 Required Functions

Basic functions for the drones processing of its environment and movement needed to be created to simulate the topologies.

7.1.4.1 Radius Detection

The purpose of radius detection is to simulate the function of a range finders or proximity sensors. In the simulation, the aforementioned GameObject List contains the objects of all drones currently within the scene. The distance between these drones is then calculated and compared to the value of a detection radius variable. If the distance is less than the radius, then return true.

7.1.4.2 Vector and Distance Calculation

The vector, or trajectory, between two drones is simply calculated by subtracting the position of both objects. The figure bellow demonstrates this.

```
// Grab Direction vector and draw
Direction = Drone2.transform.position - Drone1.transform.position;
```

Figure 23 Vector Code

Distance between two objects be calculated using a “Vector3.Distance()” function by entering the position of said two objects. This is demonstrated in the figure bellow.

```
// Measure distance from End Point
Distance = Vector3.Distance(Drone1.transform.position, EndPoint.transform.position);
```

Figure 24 Distance Code

7.1.4.3 Movement Code

To use the PID controller’s output to invoke force in a given direction, the drone object’s rigid body component is used. The rigid body component puts the objects motion under the influence of Unity’s physics engine giving the object properties such as gravity, drag, centre of mass and rotation. Rigid body can utilise a “.AddForce()” function that takes in the trajectory, to specify direction, and a number, provided by the PID output, to specify the force.

```
void Movement(Vector3 norm_dir)
{
    rb_ = GetComponent<Rigidbody>();
    if (rb_)
    {
        PID_val = PID();
        rb_.AddForce(norm_dir * PID_val);
    }
}
```

Figure 11 Movement Method

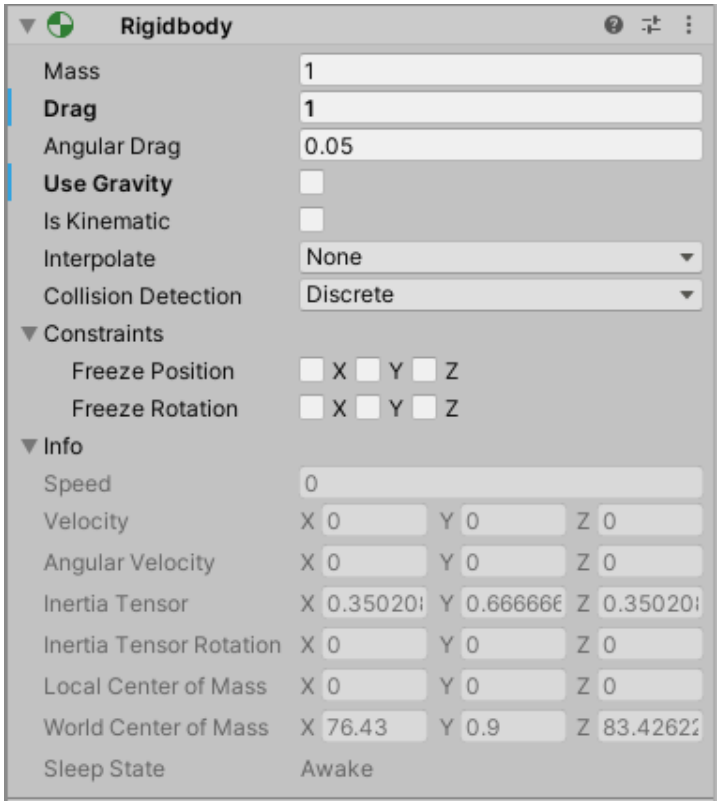

Figure 26 Rigid Body Physics Properties

7.1.4.4 Gaussian Noise Implementation

The implementation of the Box-Muller method mirrors the functionality of the equation. Furthermore, as the two random numbers in a normal distribution $(z2, z3)$ are generated using sin and cos respectively, every time the Box-Muller method is called, it will return either $z2$ or $z3$ if the other was returned previously.

The figures bellow shows how $z2$ and $z3$ are added to the vector (x, y, z) and distance values, adding Gaussian Noise to the simulation.

```
// Add white noise to vector values
closest.x += BoxMuller(); closest.y += BoxMuller(); closest.z += BoxMuller();
        // Add white noise to distance measurement
        Distance += BoxMuller();
float BoxMuller()
{
    var rand = new System.Random();
    float mean = 0.0f;
    float sd = 0.1f;
    float pi = 3.1415926535897931f;

    double z1 = RandFloat(0, (2*pi));
    double B = sd * Math.Sqrt(-2*(Math.Log(RandFloat(0,1))));

    double z2 = B * (Math.Sin(z1)) + mean;
    double z3 = B * (Math.Cos(z1)) + mean;

    if (z_val == 0)
    {
        z_val = 1;
        Debug.Log("z2: " + z2);
        return (float)z2;
    }
    else
    {
        z_val = 0;
        Debug.Log("z3: " + z3);
        return (float)z3;
    }
}

double RandFloat(double min, double max)
{
    var rand = new System.Random();
    return rand.NextDouble() * (max - min) + min;
}
```

Figure 27 Box-Muller Code

7.1.4.5 Unity PID Controller

The implementation of the PID controller utilises various anti-integral windup methods mentioned in the Design section. This includes multiplying the Integral by a certain value as well as clamping the PID's output to prevent excessive acceleration.

```
float PID()
{
    float timedelta = 0.1f;
    Error = Distance - Threshold_dist;
    Integral += timedelta * Error; Integral = Integral * 0.9f;
    Derivative = (Error - Prior_error) / timedelta;
    Prior_error = Error;A

    PID_val = Error * KP + KI * Integral + KD * Derivative;

    if (PID_val > 6) { PID_val = 6; }
    return PID_val;
}
```

Figure 28 PID Controller Code

7.2 Testing

7.2.1 PID Controller Tuning

MATLAB's Simulink makes use of a Data Inspector which plots the output of the PID controller against the actual values from the Reference Block on a graph. This is useful for analysing how the PID controller's parameters affects its characteristics, for example, overshooting and overall responsiveness.

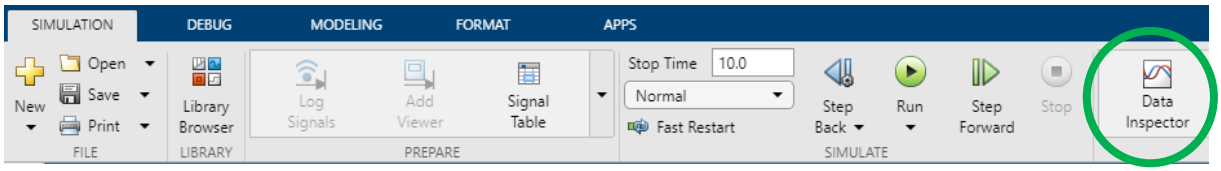

Figure 29 Data Inspector Location

7.2.1.1 Tuning Methodology

To begin tuning the PID parameters, Simulink was first run with the default values $(K_p = 1, K_i = 1, K_d = 0, t = 1)$. Analysing the performance of the default parameters meant that their values could be changed based on what areas were lacking in performance.

After analysing the performance of the default values for potential improvements, each PID parameter will be incrementally fine turned separately. The initial plan was to run the controller with all parameters being utilised. Then, optimisations could be performed to attempt fine tuning the overall controller. Conversely, separating the controller would better allow for the individual performance of each parameter's configuration to be analysed. Therefore, each parameter will be tested separately.

7.2.2 Swarm Simulation Testing

The test scenario involves four simulated drones with the objective of moving to the end point if it is within detection radius. This allows for leading drones to be naturally assigned to influence the swarm's movement in a desired direction. The simulation was run for 25 seconds per test with the detection radius set to 15 units. The end point moved forward for 10 seconds at 2 units per second then stopped, allowing the drones 15 seconds to stabilise the structure of the swarm as they approach the end point.

To fairly test the topologies, one scene, whereby the drones start in the same position and distances from each other and the end point, will be used. In addition to this, the number of drones will also be constant. To load the different topology scripts, Unity's engine allows objects to contain multiple scripts whilst disabling others, streamlining the testing process. As for running the tests, the necessary vector (x, y, z) and distance values will have Gaussian Noise added using the Box-Muller method to simulate the noise experienced in a physical system.

The Unity engine offers debug capabilities that allow the vectors between drones to be visualised. In the context of the swarm, this feature is able to visualise the interactions between the drones. Moreover, depending on the state of the interaction, different coloured rays can be used to signify the said interaction's state:

- Red – Too close
  - If the distance between a drone and its neighbour is less than the minimum threshold $(5.75\ units)$.
- Blue – Too far
  - If the distance between a drone and its neighbour is greater than the maximum threshold $(6.25\ units)$.

- Yellow – Okay distance
    - If the distance between a drone is less than the maximum threshold and greater than the minimum threshold.

Additionally, the stability of the swarm can also be signified by the colour of the debug rays. For example, if the swarm is able to settle once it has reached its destination, then the majority of yellow debug rays should increase signifying a decrease in the potential energy of the swarm. This can also serve to indicate the performance of the PID controller's configuration as it is the controller's responsibility to stabilise the swarm.

The number of interactions an individual drone may have is also represented by the number of debug rays. The more interactions a drone has, the greater the interaction magnitude, and theoretically, the greater the swarm cohesion.

Depending on the state of the drone's interactions, debug log messages will also indicate the what percentage of different interactions the swarm experienced during a test. Theoretically, when testing different topologies, the algorithm that produces the higher percentage of yellow debug lines may signify a high proficiency over other topologies.

7.2.3 Drone Properties

During initial tests of the simulation, the drone's physics properties were left to their default values. At first, the drones angular drag was set to 0.05, their drag was set to 0 and gravity was enabled. Angular drag has the property of slowing down the rotation, or angular velocity, of an object. The higher the value, the greater the rotational deceleration (Unity, n.d.). However, the drag of an object is responsible for slowing down the object's velocity with a greater value increasing the object's deceleration (Unity, n.d.). As the purpose of the simulation only involved the drone's moving to a visual trajectory, gravity was disabled. If it was left enabled, the drones would not be able to generate enough thrust to move towards a trajectory, simply falling to the ground. Theoretically, a separate PID controller could be implemented to manage the downward thrust of the drones, allowing the drones to move freely.

The drone's drag was set to 1 to simulate the air resistance the Crazyflies would experience, adding realistic deceleration to the simulation. Initial tests without drag showed the drones exponentially increasing with speed as force continued to be added by the PID controller.

7.2.4 Testing Constraints

As the swarm simulation served as a contingency plan to the project's original objective, time became a constraint. This affected the level of detail that could be explored regarding the swarm simulation and testing the intricacies of its behaviour. The plan to collect empirical data involved logging the

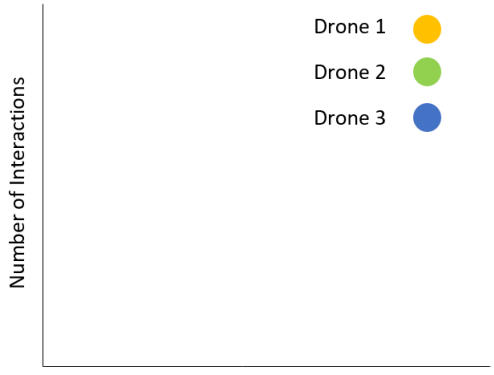

Figure 30 Number of Interactions Over Time

number of neighbours a drone within the swarm interacted with within a certain amount of time. The logged data would then be plotted on a graph with number of interactions on the y axis and time on the x. The graph would have been able to show the interactions of multiple drones at once with the purpose of conveying the overall cohesion within the swarm.

This may have also given evidence to better shed light on the hypothesis that, depending on the swarm topology, the greater the interaction magnitude, the higher the swarm cohesion.

Furthermore, the secondary objectives concerning the swarm's simulation were not able to be met due to time constraints. These intended to further test the obstacle avoidance and stability capabilities of the PID controller by adding random collision objects and wind as an environmental effect.

8 Evaluation

8.1 Simulink PID Controller Results

Based on the performance of the default values, the PID controller seemed to have performed admirably. There was no sign of overshoot (likely due to the anti-integral windup feature) and the controller was able to conform to the actual values. However, it may still be possible to improve the responsiveness of the controller whilst maintaining accuracy.

Contrary to what the results may signify, some initial overshoot can be considered desirable as the controller output is responsible for the regulating the drones speed. If the initial acceleration is high (the controller overshoots), and the drone is able to effectively decelerate, the PID configuration can be considered successful. Moreover, if there is no oscillation in the PID output, the controller is likely to promote stability.

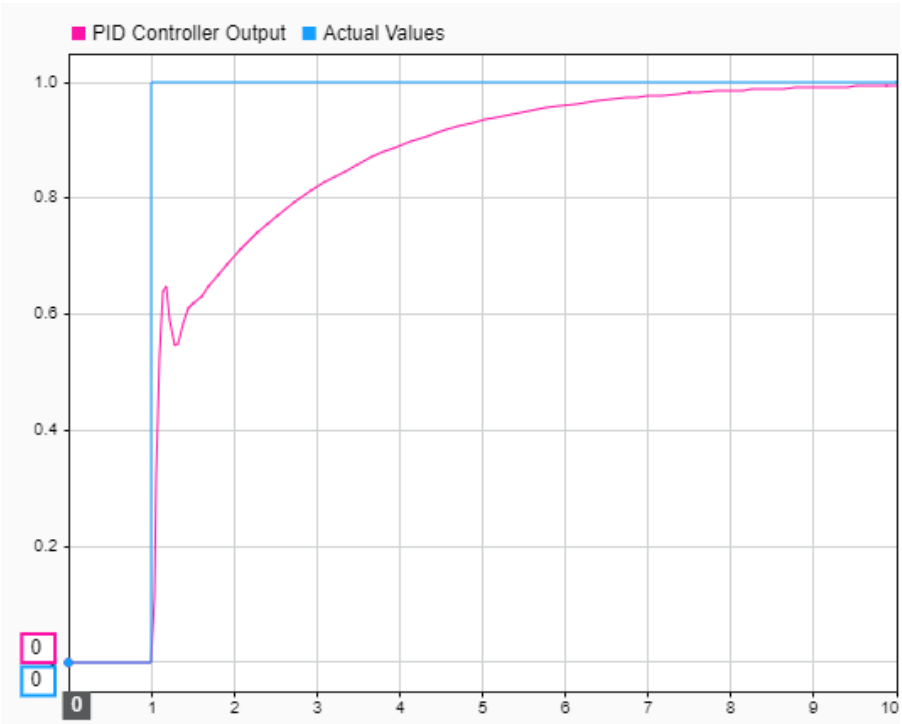

Figure 31 $(K_p, K_i, K_d) = 1$

8.1.1 Proportional Controller

The step being used to generate the actual values forces the PID controller to react abruptly. As the proportional controller greatly effects the initial responsiveness of the output, theoretically, incrementally increasing K_p should eventually provide reasonable results (Vandoren, 2016). On the other hand, raising K_p too high could result in unwanted initial overshooting as the proportional value produces too large an output.

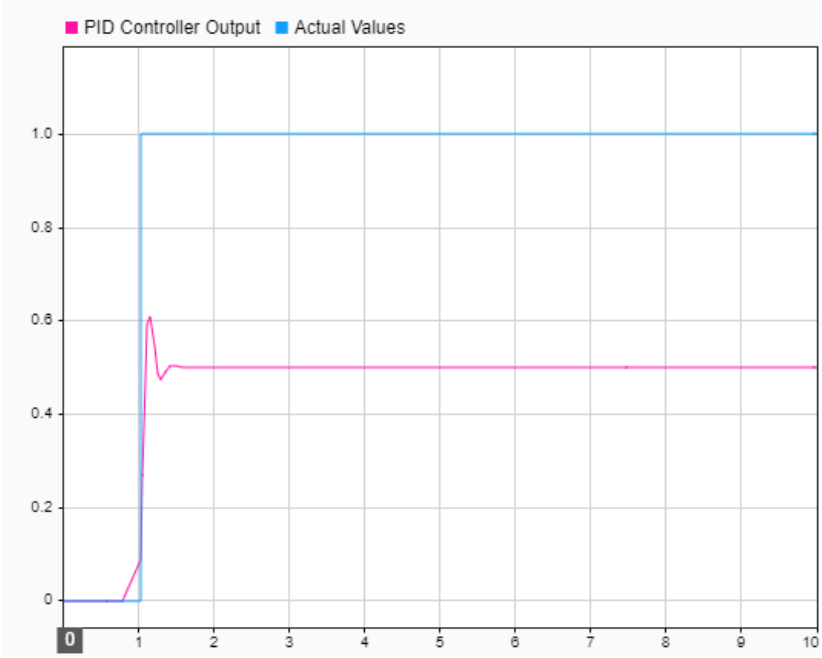

Figure 32 $K_p = 1$

Making $K_p = 2.8$ allowed the PID output to reach a reasonable value without overshooting. However, after the quick initial spike, the output maintains a steady 0.74. Here, the integral controller will begin to contribute more to reaching the actual values.

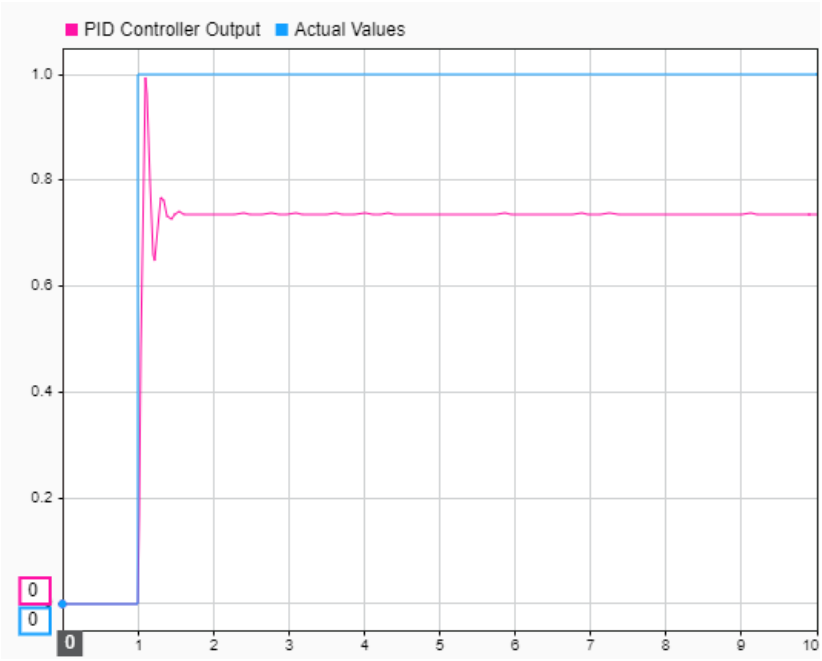

Figure 33 $K_p = 2.8$

8.1.2 Integral Controller

Once the proportional controller hits steady state error, the integral controller must smoothly and surely reach the actual values. This is to offset the initial acceleration from the proportional controller. With the default $K_i = 1$. The output conforms smoothly to the actual values. Reducing the integral to $K_i = 0.5$ produced a much more gradual curve signifying a steadier transition to the actual values. This is an important characteristic for the integral controller to have as reaching steady state error signifies that the drone is approaching its destination. Therefore, the drone's velocity must begin to decelerate.

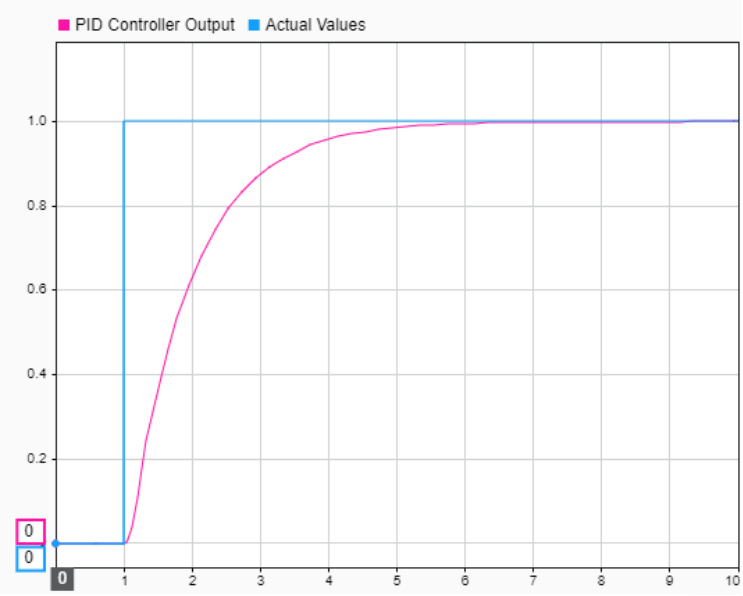

Figure 34 $K_i = 1$

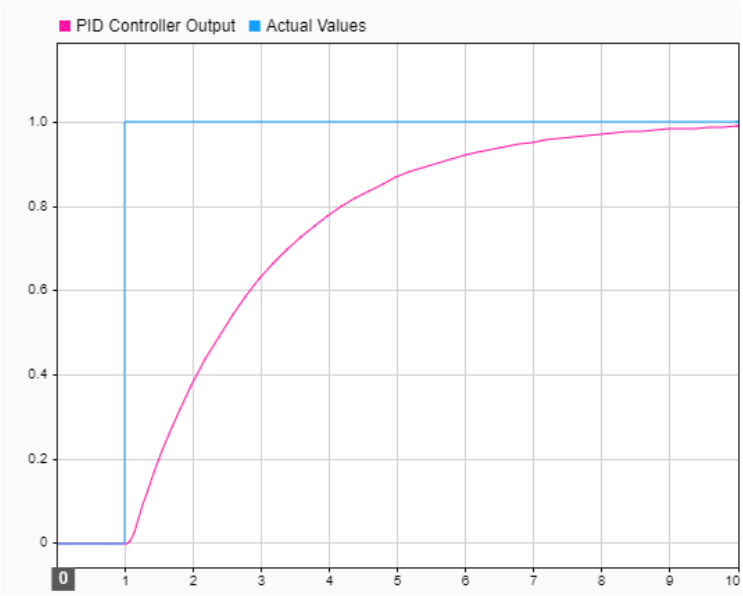

Figure 35 $K_i = 0.5$

8.1.3 Derivative Controller

As the derivative controller is mostly used when producing negative errors in response to overshoot, having too high a value may produce oscillatory results when managing the drones flight. To mitigate this, the value of the derivative cannot be larger than K_p and K_i. $K_d = 0.3$ produced a much lower initial spike with the negative error transitioning to zero much more smoothly in comparison to the default, $K_d = 1$. Nevertheless, if K_d is too low, the derivative controller is not able effectively counter the overshoot caused by K_p and K_i as the negative error produced may be too small.

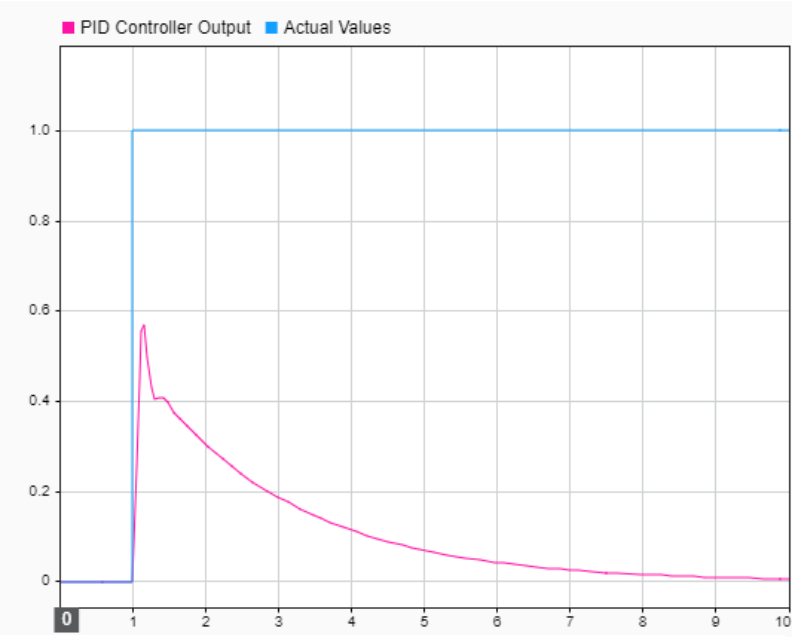

Figure 36 $K_d = 1$

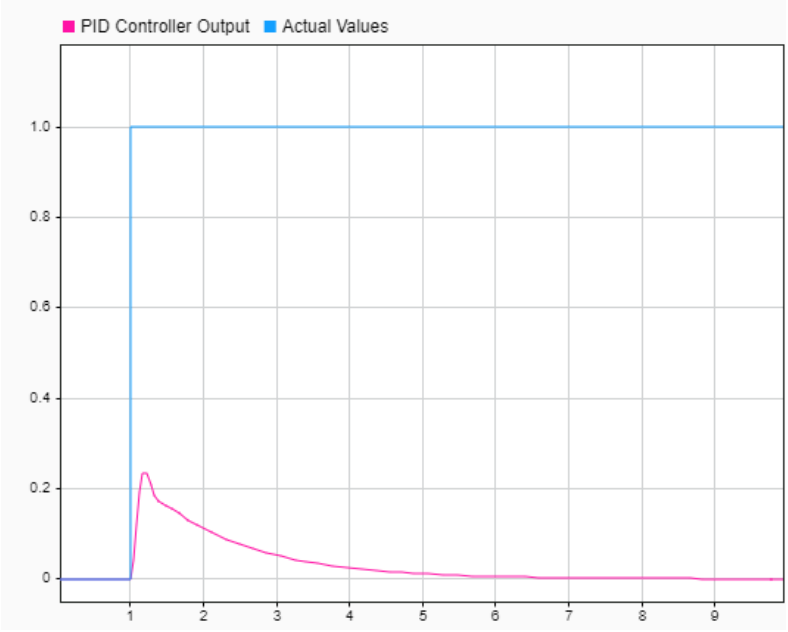

Figure 37 $K_d = 0.3$

8.1.4 Overall Controller

The overall performance of the controller shows the characteristics of having a high initial overshoot before gradually conforming to the actual values. Theoretically, this performance may translate to the drones having a high initial acceleration before gradually decelerating towards its trajectory. This however depends on whether the acceleration causes the drone to oscillate whilst quickly reaching its destination, resulting in a lack of stability. The PID controller's ability to regulate the drone's movement however may depend on the configuration of the Unity physics engine. Consequently, it may be necessary to continue fine tuning the controller within the swarm simulation.

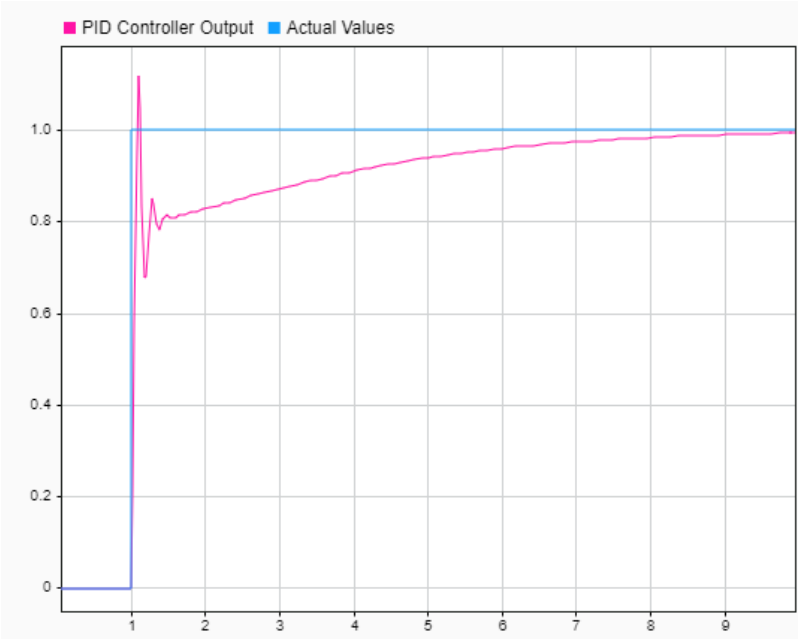

Figure 38 ($K_p = 2.8, K_i = 0.5, K_d = 0.3$)

8.2 Swarm Simulation Evaluation

8.2.1 PID Controller Simulation Adjustments

Whilst running initial tests, having $K_p = 2.8$ resulted in the drones overshooting their objective. Furthermore, the derivative was not able to produce a high enough negative error to mitigate this. The drone's overshoot could be described as erratic and oscillatory in movement, as a result, it appeared appropriate to reduce the drone's initial acceleration by lowering the proportional output. $K_p = 1.5$ produced a slightly slower and gradual acceleration. Nevertheless, the drone's, and subsequently the swarm's movement, saw an increase in precision and stability. $\mathrm{K_d}$ also saw a minor decrease to 0.2 to reduce the chances of oscillation occurring from too large of a negative error following $the\ decrease\ of\ \mathrm{K_p}$.

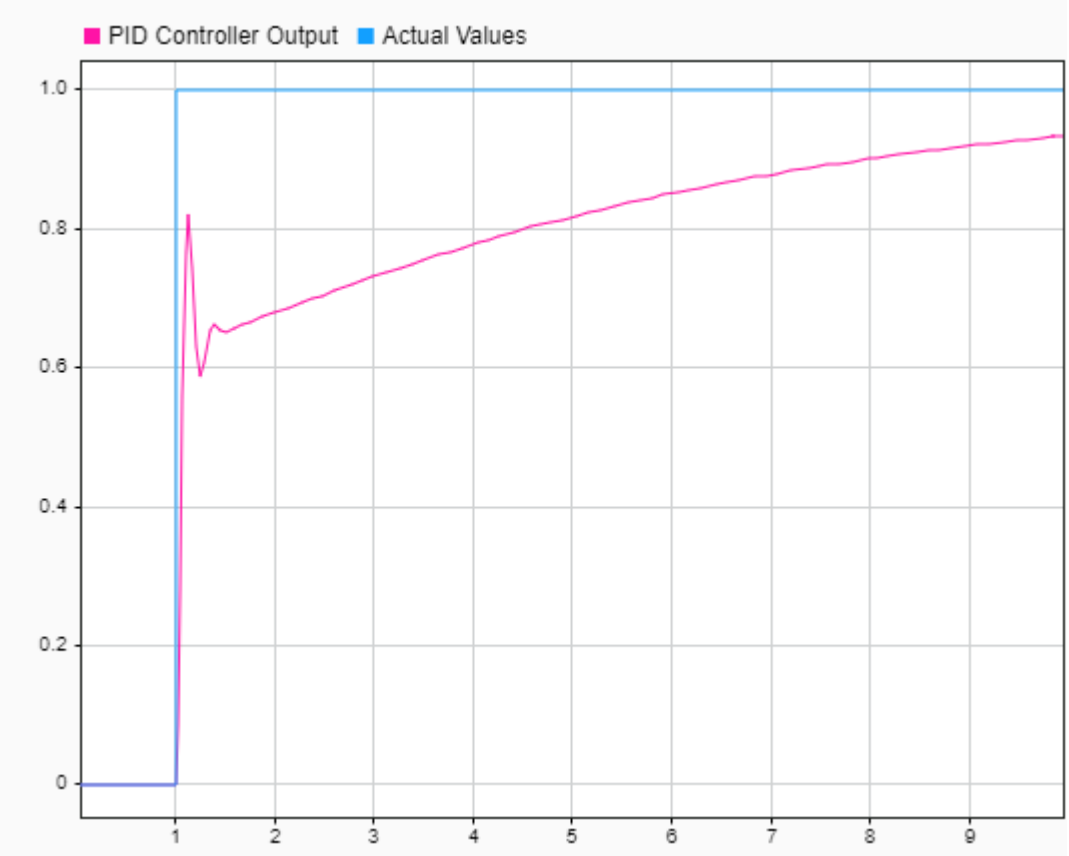

Figure 39 $(K_p = 1.5, K_i = 0.5, K_d = 0.2)$

8.3 Average Vector

As visualised in the figure below, the green rays signify the average vector calculated by each drone. It also shows the green rays converging at a single point in the centre of the swarm conveying the drones successfully maintaining cohesion and stability. On the other hand, the overall lack of yellow rays suggests that this topology for the most part struggles to maintain a healthy distance between neighbouring drones and the end point despite being relatively stable. Nevertheless, the swarm was able to successfully reach and settle at the end point whilst avoiding collisions.

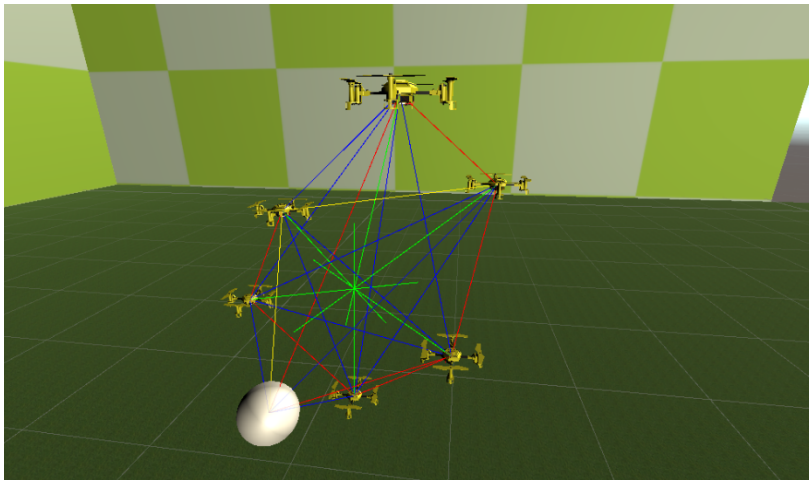

Figure 40 Visualisation of Average Vector Interactions

8.4 Vector All

Unlike the other topologies, the stability shown the figure bellow varied as when the end point stopped moving, the swarm would proceed to slowly orbit the end point as it attempted to stabilise. This could be attributed to the Gaussian Noise added to the vector and distance values potentially skewing the topology's decision-making accuracy. Nonetheless, throughout testing there was no collisions or oscillation despite the lack of yellow rays signifying the drones struggled to maintain a desired distance between neighbours.

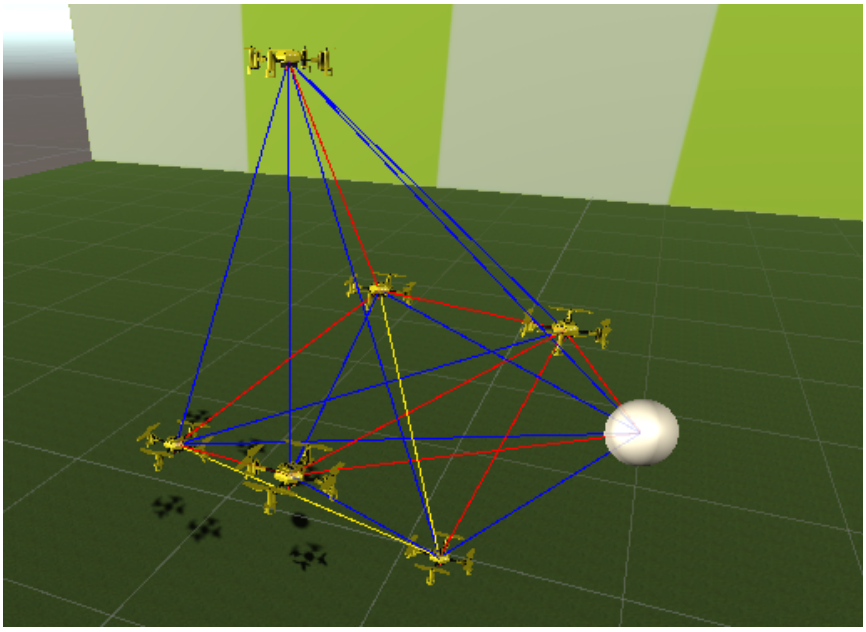

Figure 41 Visualisation of Vector All Interactions

8.5 Closest Neighbour

Despite having less interaction in both numbers and complexity than other topologies, reducing the number of interactions a drone must consider greatly increased the swarm's structural stability. This is signified by the yellow rays indicating that the drones are capable of maintaining a distance between the minimum and maximum thresholds. Theoretically, if an optimal distance is maintained from a drone's closest neighbour, it is likely at a safe distance from neighbours further away whilst still contributing to swarm stability. Although, overall distances between drones, in comparison to the other topologies, appear further due to only one drone being considered at one time, decreasing the swarm's structural cohesion.

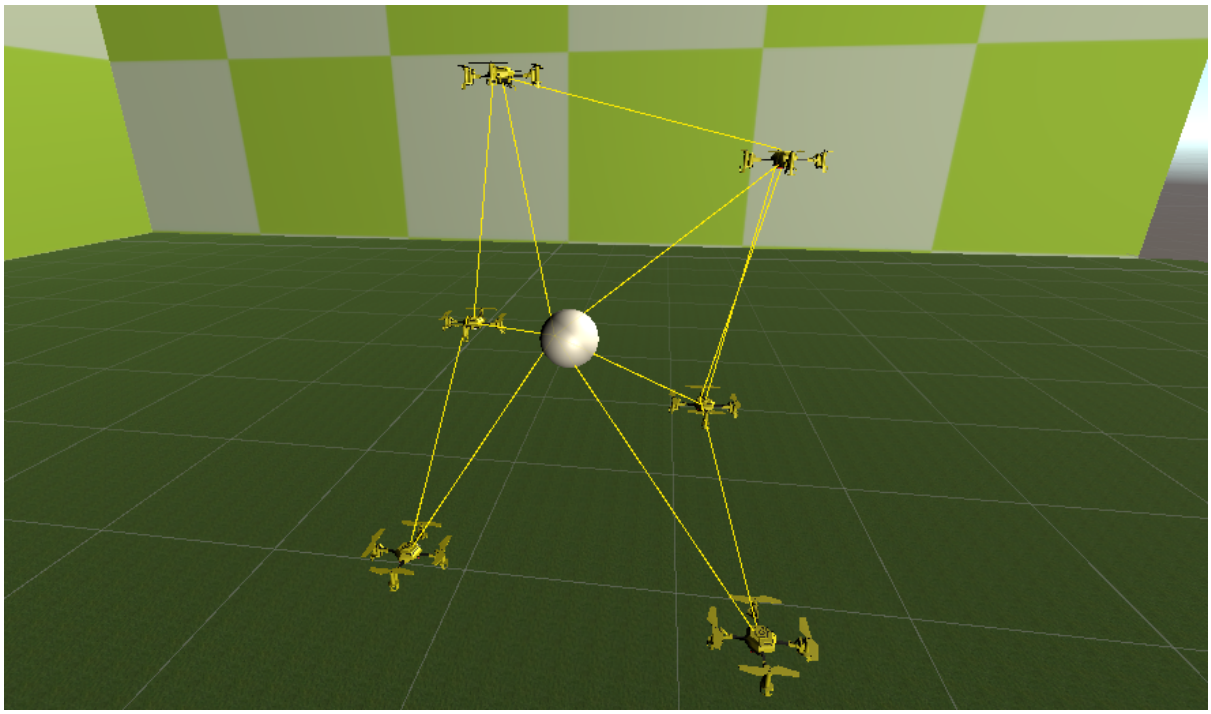

Figure 42 Visualisation of Closest Neighbour Interactions

8.6 Topology Comparison

Based on the interaction data bellow, a clear conclusion shows that topologies that have more data to consider have a lower percentage of interactions where the drones maintain a desired distance between neighbouring drones and the end point. Regarding the swarm interactions, Vector All and Average Vector scored 10% and 8% respectively, whereas Closest Neighbour scored 62%. Additionally, regarding interactions with the end point, Vector All scored 9% and Average Vector scored 17%. Both were dwarfed however by Closest Neighbour's 63%. This data unequivocally proves that the Closest Neighbour topology had a high success rate for maintaining swarm stability whilst moving towards and objective. This satisfies the projects requirements of developing desirable emergent behaviour whilst also having the swarm's movement maintained by a PID controller. Furthermore, an optimal configuration for the PID controller was found, allowing the prevention of collisions between drones whilst testing the final iteration of the emergent algorithms. On the other hand, whilst the swarm cohesion for Closest Neighbour may have been less optimal than the other topologies, its overall structure remained uncompromised with the swam maintaining a majority cluster.

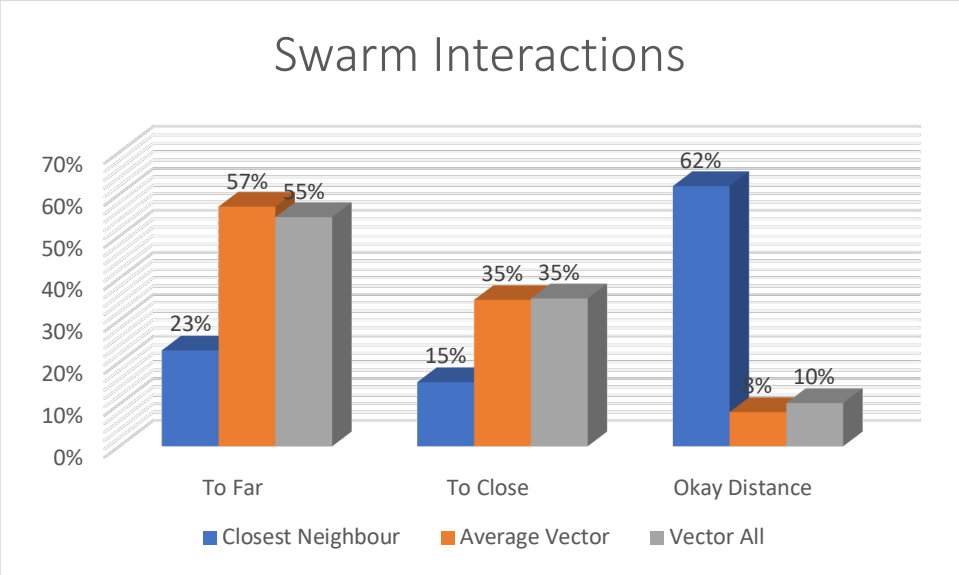

Figure 43 Percentage of Different Swarm Interactions Per Topology

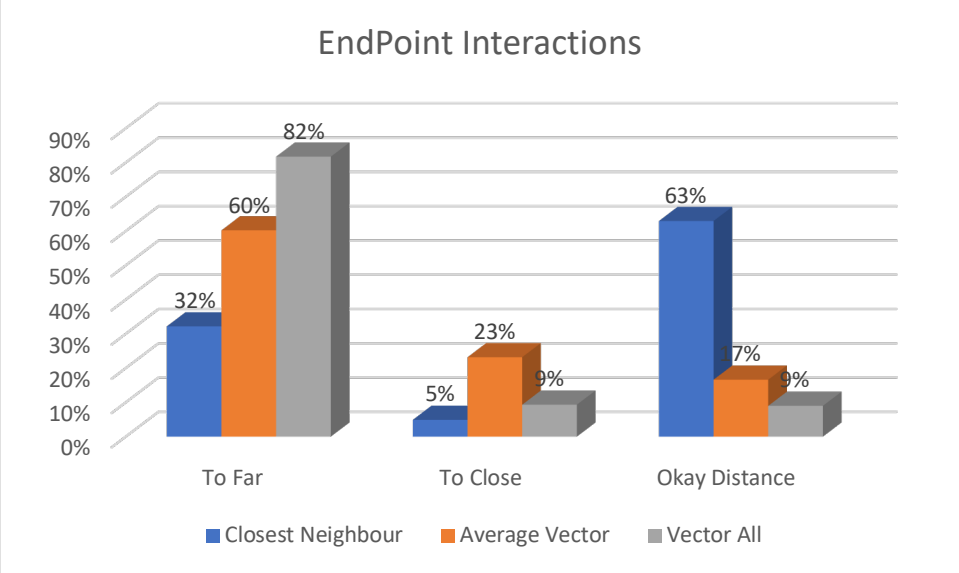

Figure 44 Percentage of Different Swarm Interactions Per Topology

9 Conclusion

Despite the constraints experienced during this project's development, the digital twin aspect was able to provide critical analysis on the consideration of swarm topologies for a physical system, in this case, the Crazyflie swarm. The simulation provided evidence to suggest that physicomimetic emergent behaviour is a potentially viable method of sustaining swarm structure, stability and cohesion whilst moving towards a trajectory. The main objectives surrounding the swarm simulation were met as it was able to sustain proper flight whilst avoiding collisions with other neighbouring drones. Unfortunately, however, time constraints limited the variation in environments that these topologies were tested in. Concerning the secondary objectives, adding environmental disturbances such as wind or obstacles would serve to further highlight the accountability capabilities of the PID controller, showcasing its ability to continue to maintain the drone's, and subsequently the swarm's, trajectory. Nevertheless, the swarm simulation has been able to emulate the original objective which was to implement emergent behaviour in a Crazyflie swarm which would have been tested in an open space. Therefore, theoretically, a physical system may be able to implement similar physicomimetic emergent behaviour based on the results of this project.

Whilst exploring the many aspects of the Crazyflie platform, much has been learnt about the details of its API and the methods surrounding its development most notably from the opensource community. This is especially true in regard to the different technologies required to develop for the Crazyflie API, technologies that naturally overlapped in both hardware and software. Aspects of other fields relating to the understanding and development of both the Crazyflie swarm and swarm simulation have also imparted much knowledge. This was true when understanding the physical implications of the swarm topologies, the mathematical aspects of the PID control system or the biological side of understanding swarm or flocking behaviours in nature.

Concerning both the research question and hypothesis, it is clear that the contrast in previous assumptions compared to the results of this project signify that a greater understanding of the subject matter was obtained. The research question asked how emergent behaviour would be affected based on different decision-making processes influenced by topologies whilst the hypothesis predicted that more complex interactions would benefit swarm cohesion, structure and stability. In reality, the results of the swarm simulation suggested quite the opposite in all aspects aside from structure for the Closest Neighbour topology. Nevertheless, its performance was by far the most desirable, subverting initial expectations. This signifies that the strongest topology is the simplest one, which holds true to the established philosophy of emergence (Reynolds, 1987).

The methodologies this project adopted proved appropriate especially regarding the prototyping method used for the emergent swarm behaviour. Many iterative changes were needed for fine tuning the PID controller and optimising the emergent algorithms, removing the chances of obscure behaviour. On the other hand, the Crazyflie GUI and aspects of the Crazyflie swarm had a set specification and theoretical functionality that was unlikely to change.

Due to the constraints caused by the effects of covid-19, it was imperative to implement a viable contingency plan that would still convey the main objectives of this project. Through this experience, skills, such as using the Unity engine, were learned allowing an important aspect of the original project to be explored. An improvement that could have been made, however, was developing methods to run the project in release mode instead of debug, improving performance and enabling the size of the swarm to be scaled even further.

9.1 Further Work

To further enhance a future version of this project, simulating the pitch, yaw, roll and thrust of a virtual drone and regulating them with PID controllers would increase the simulation's realism as a digital twin. Applying these advancements to complete tasks in virtual hazardous environments would add to the validity of the simulation, allowing more viable swarm topologies to be developed.

9.2 Final Words

Overall, this project was incredibly insightful and enjoyable. Many mistakes were made and learned from, adding to the subject matter's intrigue.

10 References

Arnaud. (2013) Communication with multiple Crazyflies. Bitcraze Forums. 11 November. Available online: https://forum.bitcraze.io/viewtopic.php?f=9&t=624 [Accessed 14/2/2020].

Baetica, A. (2015) *Integrator windup and PID controller design* [Presentation]. Available online: https://www.cds.caltech.edu/~murray/wiki/images/6/6f/Recitation_110_nov_17.pdf [Accessed 15/2/2020].

Bard, J. (2008) Morphogenesis. Scholarpedia, 3(6), 2422. Available online: http://www.scholarpedia.org/article/Morphogenesis [Accessed 5/5/2020].

Bratton, D. & Kennedy, J. (2007) Defining a Standard for Particle Swarm Optimization - IEEE Conference Publication. Ieeexplore.ieee.org. Available online: https://ieeexplore.ieee.org/document/4223164 [Accessed 29/3/2020].

Chien, I. (1988) IMC-PID Controller Design - An Extension. *IFAC Proceedings Volumes*, 21(7), 147-152. Available online: https://doi.org/10.1016/S1474-6670(17)53816-1 [Accessed 16/1/2020].

Chung, D. (n.d.) OFFensive Swarm-Enabled Tactics. Darpa.mil. Available online: https://www.darpa.mil/program/offensive-swarm-enabled-tactics [Accessed 5/2/2020].

Couzin, I. (2009) Collective cognition in animal groups. *Trends in Cognitive Sciences*, 13(1), 36-43. Available online: https://www.sciencedirect.com/science/article/abs/pii/S1364661308002520 [Accessed 31/4/2020].

Crazyflie 2.1 | Bitcraze. (2019) Bitcraze.io. Available online: https://www.bitcraze.io/products/crazyflie-2-1/ [Accessed 17/10/2019].

doc:crazyflie:client:pycfclient:zmq [Bitcraze Wiki]. (n.d.) Wiki.bitcraze.io. Available online: https://wiki.bitcraze.io/doc:crazyflie:client:pycfclient:zmq#input_device [Accessed 2/3/2020].

Douglas, B. (2012) PID Control - A brief introduction [Video]. Available online: https://www.youtube.com/watch?v=UR0hOmjaHp0 [Accessed 29/11/2019].

Engineers create drones based on digital twins. (2019) Techxplore.com. Available online: https://techxplore.com/news/2019-12-drones-based-digital-twins.html [Accessed 13/5/2020].

Flaherty, N. (2019) *Digital twin of UAV provides predictive maintenance*. eeNews Europe. Available online: https://www.eenewseurope.com/news/digital-twin-uav-provides-predictive-maintenance [Accessed 7/5/2020].

Garnier, S., Gautrais, J. & Theraulaz, G. (2007) The biological principles of swarm intelligence. *Swarm Intelligence*, 1(1), 3-31. Available online: https://doi.org/10.1007/s11721-007-0004-y [Accessed 17/4/2020].

Hallinan, J. (2013) Methods in microbiology, 40th edition.Science Direct, 1-37.

Kallenborn, Z. (2018) The Era of the Drone Swarm Is Coming, and We Need to Be Ready for It - Modern War Institute. Modern War Institute. Available online: https://mwi.usma.edu/era-drone-swarm-coming-need-ready/ [Accessed 13/2/2020].

Kennedy, J. & Mendes, R. (2002) Population structure and particle swarm performance - IEEE Conference Publication. Ieeexplore.ieee.org. Available online: https://ieeexplore.ieee.org/document/1004493 [Accessed 31/1/2020].

Kimberly. (2019) Can't control more than 3 CFs with one radio. Bitcraze Forums. 22 July. Available online: https://forum.bitcraze.io/viewtopic.php?t=3574 [Accessed 13/2/2020].

LaValle, S. (2006) Odometry sensors. *Planning Algorithms.* Available online: http://planning.cs.uiuc.edu/node576.html [Accessed 27/3/2020].

Maldini, M. (2018) *PID control explained*. Medium. Available online: https://medium.com/@mattia512maldini/pid-control-explained-45b671f10bc7 [Accessed 1/12/2019].

Marr, B. (2017) *What Is Digital Twin Technology - And Why Is It So Important?*. Forbes. Available online: https://www.forbes.com/sites/bernardmarr/2017/03/06/what-is-digital-twin-technology-and-why-is-it-so-important/ [Accessed 14/5/2020].

Mendivez Vasquez, B. & Barca, J. (2018) Network Topology Inference in Swarm Robotics. *2018 IEEE International Conference on Robotics and Automation (ICRA)*,7660-7666. Available online: https://www.researchgate.net/publication/327805112_Network_Topology_Inference_in_Swarm_Robotics [Accessed 2/5/2019].

Najm, A. & Ibraheem, I. (2019) Nonlinear PID controller design for a 6-DOF UAV quadrotor system. *Engineering Science and Technology, an International Journal*, 22(4), 1087-1097. Available online: https://doi.org/10.1016/j.jestch.2019.02.005 [Accessed 16/1/2020].

Oracle VM VirtualBox. (n.d.) Virtualbox.org. Available online: https://www.virtualbox.org/ [Accessed 8/11/2019].

Potter, D. (n.d*.) Swarm Control using Physicomimetics | Navy Center for Applied Research in Artificial Intelligence*. Nrl.navy.mil. Available online: https://www.nrl.navy.mil/itd/aic/content/swarm-control-using-physicomimetics [Accessed 9/5/2020].

Preiss, J., Honig, W., Sukhatme, G. & Ayanian, N. (2017) Crazyswarm: A large nano-quadcopter swarm. *2017 IEEE International Conference on Robotics and Automation* (ICRA) Available online: https://crazyswarm.readthedocs.io/en/latest/ [Accessed 18/12/2019].

Retroreflective Markers. (2015) Nextstagepro.com. Available online: http://www.nextstagepro.com/retroreflective-markers.html [Accessed 9/10/2019].

Reynolds, C. (1987) Flocks, herds and schools: A distributed behavioral model*. Proceedings of the 14th annual conference on Computer graphics and interactive techniques* - SIGGRAPH '87 Available online: https://dl.acm.org/doi/abs/10.1145/37401.37406 [Accessed 8/4/2020].

Righetti, L., Sharkey, N., Arkin, R., Ansell, D., Sassòli, M., Heyns, C., Asaro, P. & Lee, P. (2014) Autonomous Weapon Systems Technical, Military, Legal and Humanitarian Aspects. Reliefweb.int. Available online: https://reliefweb.int/sites/reliefweb.int/files/resources/4221-002-autonomous-weapons-systems-full-report%20%281%29.pdf [Accessed 9/2/2020].

Roberts, I. (2019) *Turn virtual trajectory into reality by flying drones*. EdD thesis. The University of Hull.

Royce, W. (1970) Managing the development of large software systems: concepts and techniques | Proceedings of the 9th international conference on Software Engineering. Dl.acm.org. Available online: https://dl.acm.org/doi/10.5555/41765.41801 [Accessed 8/5/2020].

Salih, A., Moghavvemi, M., Mohamed, H. & Gaeid, K. (2010) *Flight PID controller design for a UAV quadrotor*. Academicjournals.org. Available online: https://academicjournals.org/journal/SRE/article-abstract/4EFF99440127 [Accessed 7/4/2020].

Sherrell, L. (2013) Evolutionary Prototyping. SpringerLink. Available online: https://link.springer.com/referenceworkentry/10.1007%2F978-1-4020-8265-8_201039 [Accessed 10/5/2020].

Shin, H. & Park, J. (2012) Anti-Windup PID Controller With Integral State Predictor for Variable-Speed Motor Drives. *IEEE Transactions on Industrial Electronics*, 59(3), 1509-1516. Available online: https://ieeexplore.ieee.org/abstract/document/5977027 [Accessed 1/5/2020].

Silano, G. & Iannelli, L. (2018) *CrazyS: A Software-in-the-Loop Simulation Platform for the Crazyflie 2.0 Nano-Quadcopter*. giuseppesilano. Available online: https://giuseppesilano.net/publications/rosChapter19.pdf [Accessed 6/4/2020].

Simulink Documentation - MathWorks United Kingdom. (n.d.) Uk.mathworks.com. Available online: https://uk.mathworks.com/help/simulink/ [Accessed 18/5/2020].

Software Prototyping - Ingsoftware. (n.d.) Software Prototyping - Ingsoftware. Available online: https://www.ingsoftware.com/software-prototyping [Accessed 2/6/2020].

Spears, W., Spears, D., Hamann, J. & Heil, R. (2004) Distributed, Physics-Based Control of Swarms of Vehicles. Autonomous Robots, 17(2/3), 137-162.

Spector, L., Klein, J., Perry, C. & Feinstein, M. (2005) Emergence of Collective Behavior in Evolving Populations of Flying Agents. *Genetic Programming and Evolvable Machines*, 6(1), 111-125. Available online: https://link.springer.com/article/10.1007/s10710-005-7620-3 [Accessed 6/5/2020].

Tahir, A., Böling, J., Haghbayan, M., Toivonen, H. & Plosila, J. (2019) Swarms of Unmanned Aerial Vehicles — A Survey. Journal of Industrial Information Integration, 16, 100106. Available online: https://www.sciencedirect.com/science/article/pii/S2452414X18300086#! [Accessed 18/11/2019].

Top 4 software development methodologies | Synopsys. (2017) Software Integrity Blog. Available online: https://www.synopsys.com/blogs/software-security/top-4-software-development-methodologies/ [Accessed 13/5/2020].

Treherne, J. & Foster, W. (1981) Group transmission of predator avoidance behaviour in a marine insect: The trafalgar effect. *Animal Behaviour*, 29(3), 911-917. Available online: https://www.sciencedirect.com/science/article/abs/pii/S0003347281800280#! [Accessed 27/4/2020].

Unity - Scripting API: Rigidbody.angularDrag. (n.d.) Docs.unity3d.com. Available online: https://docs.unity3d.com/ScriptReference/Rigidbody-angularDrag.html [Accessed 10/5/2020].

Unity - Scripting API: Rigidbody.drag. (n.d.) Docs.unity3d.com. Available online: https://docs.unity3d.com/ScriptReference/Rigidbody-drag.html [Accessed 10/5/2020].

Vandoren, V. (2016) *Understanding PID control and loop tuning fundamentals*. Control Engineering. Available online: https://www.controleng.com/articles/understanding-pid-control-and-loop-tuning-fundamentals/ [Accessed 15/3/2020].

Vidan, C., Mihai, R., Găiceanu, M. & Ilie, G. (2018) *Designing an altitude controller for a mini-UAV using an automated speed device*. AIP Conference Proceedings. Available online: https://aip.scitation.org/doi/abs/10.1063/1.5081626 [Accessed 11/4/2020].

Vásárhelyi, G., Virágh, C., Somorjai, G., Nepusz, T., Eiben, A. & Vicsek, T. (2018) Optimized flocking of autonomous drones in confined environments. Science Robotics, 3(20), eaat3536. Available online: http://hal.elte.hu/drones/ [Accessed 20/4/2020].

Weisstein, E. (n.d.) *Box-Muller Transformation -- from Wolfram MathWorld*. Mathworld.wolfram.com. Available online: https://mathworld.wolfram.com/Box-MullerTransformation.html [Accessed 7/5/2020].

Zhang, Y. (2019) Experimental implementation of distributed time-varying optimization algorithms using crazyflie platform. California Riverside: University of California Riverside, 5-21.

11 Appendix A Notes

Aside from packet loss, utilising UDP with a high enough frequency may prove detrimental to the Crazyflie's onboard systems. Theoretically, using a UPD protocol may be faster than TCP as UDP does not need to wait for delivery a confirmation in order to send the next packet. However, if UDP continued to flood the Crazyflie with data packets, the onboard systems may become encumbered with out of date packets. This assumes that that the Crazyflie is able to receive packets whilst executing previous commands. Theses packets would be out of date in the sense that they were not executed in time. This may cause a decrease in the drone's responsiveness proving detrimental to flight stability.

To mitigate the exponential acceleration of the drones in the simulation when drag is disabled, it may be possible to implement a separate PID controller to manage the adding force in the opposite direction of the drone's flight.

One reason the simulation caused occasional stuttering and drops in frame rates when running on a laptop is because it was run in debug mode as opposed to release mode. This was required in order to be able to use the various debug features needed to visually assess the swarm's stability and cohesion as well as collect data on the different types of interactions. Alternative methods to achieve this whilst running in release mode would have required more time to develop.

Throughout the report, the Vicon Vision System describes the Vicon Vision Cameras, also referred to as Vicon Cameras or Vicon Motion Capture Cameras, as well as the Vicon software required to utilise the cameras. The specific Vicon software referenced throughout this report is simply known as Tracker which is currently in version 3.7.0 (Vicon, 2020).

12 Appendix B Supporting Images of Drone Topologies

12.1 Average Vector Topology Images

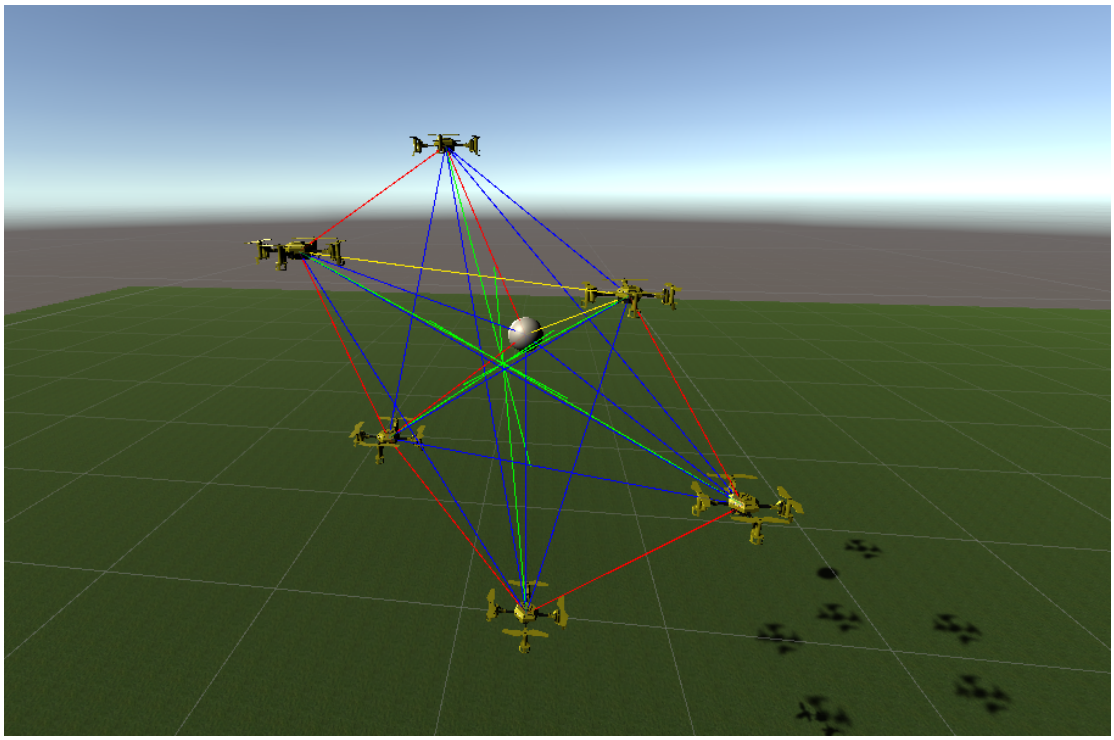

Figure 45

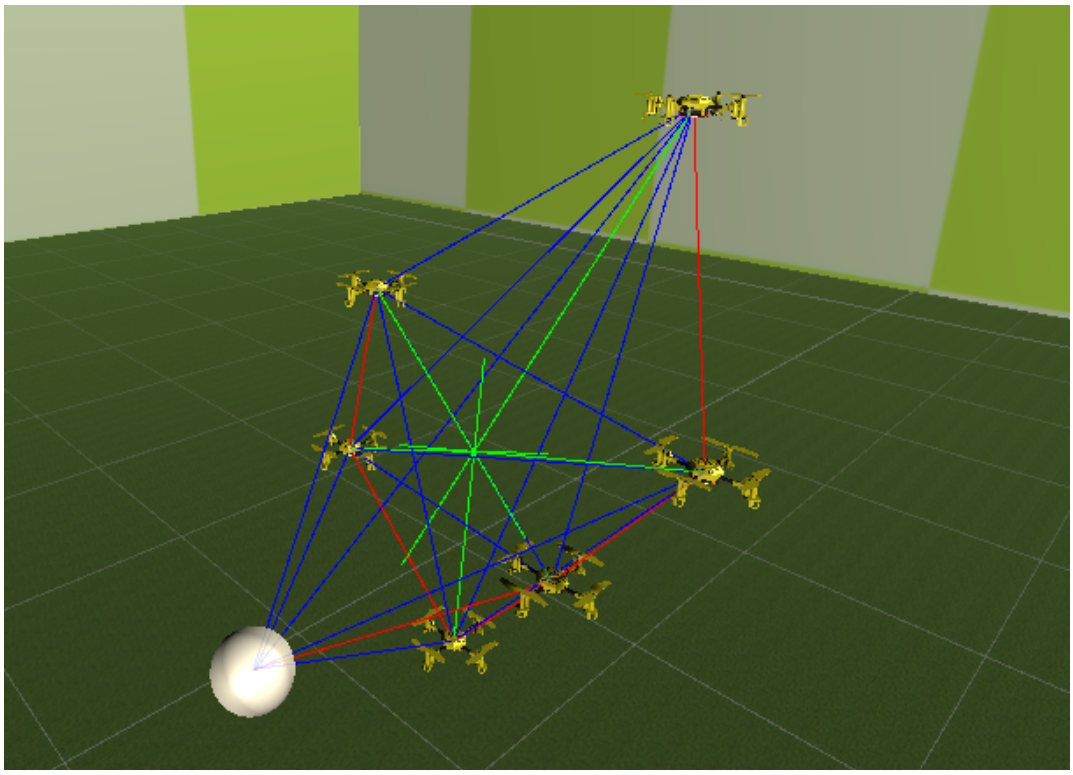

Figure 46

12.2 Closest Neighbour Topology Images

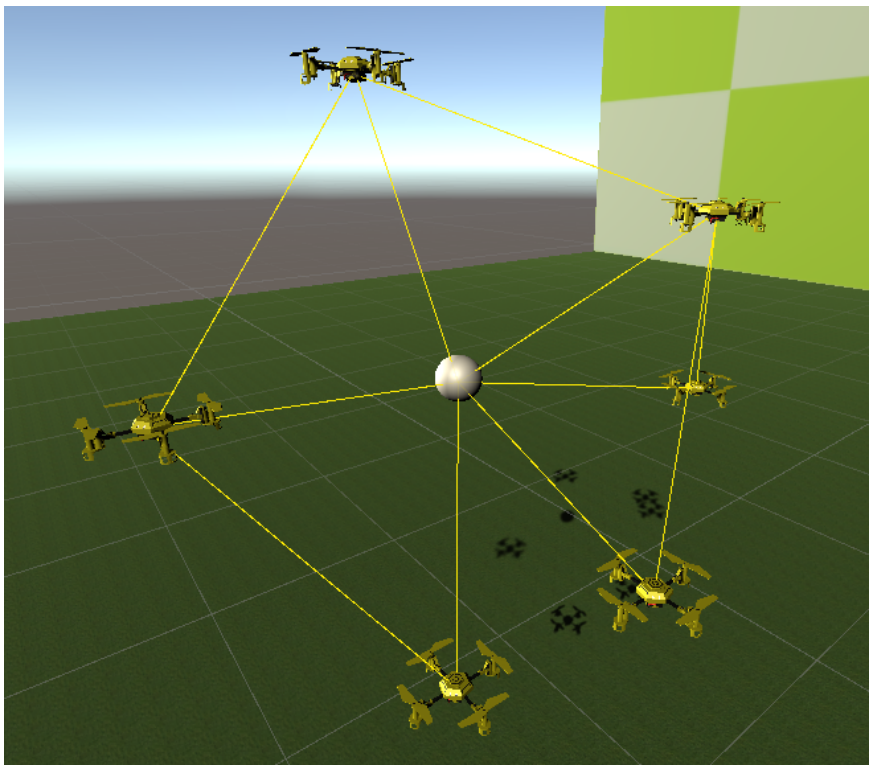

Figure 47

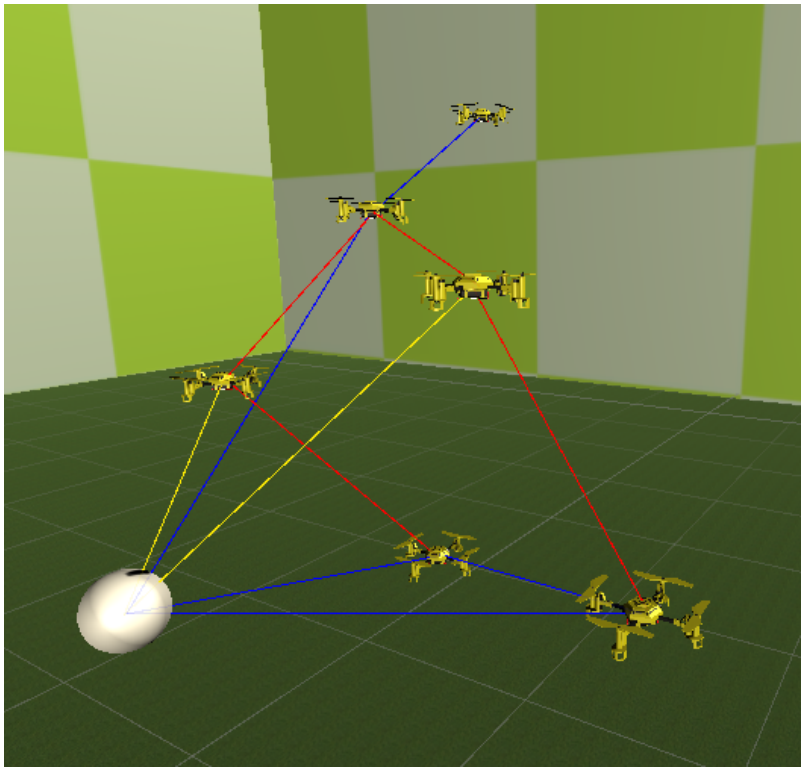

Figure 48

12.3 Vector All Topology Images

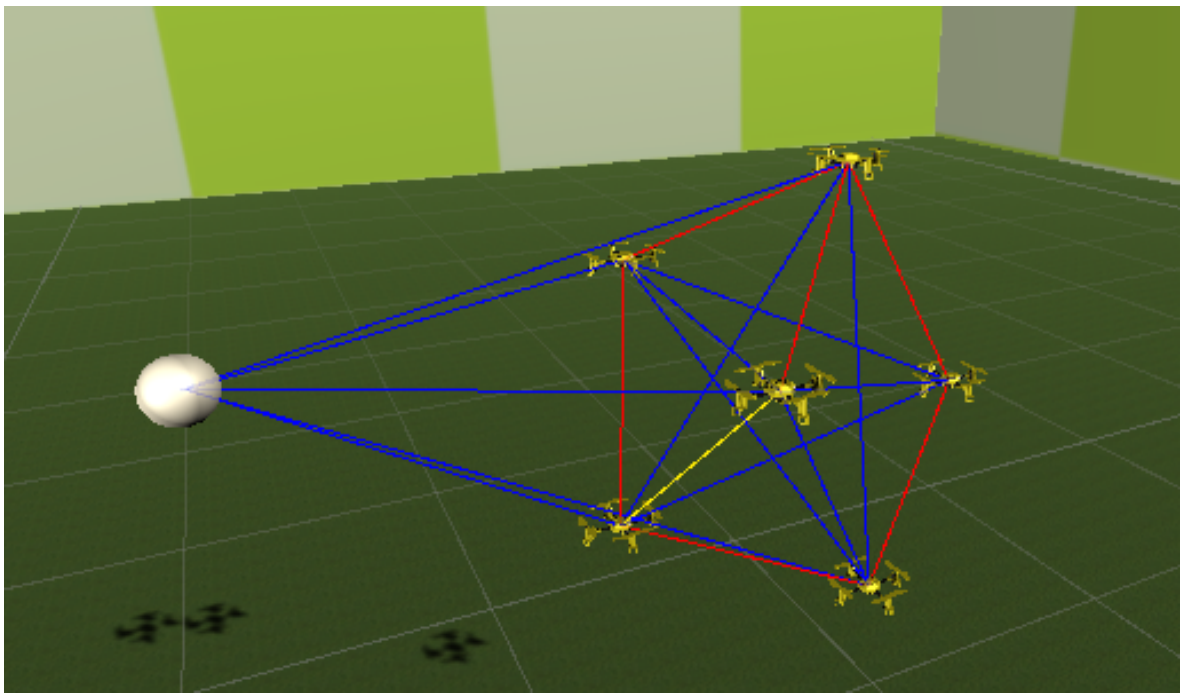

Figure 49

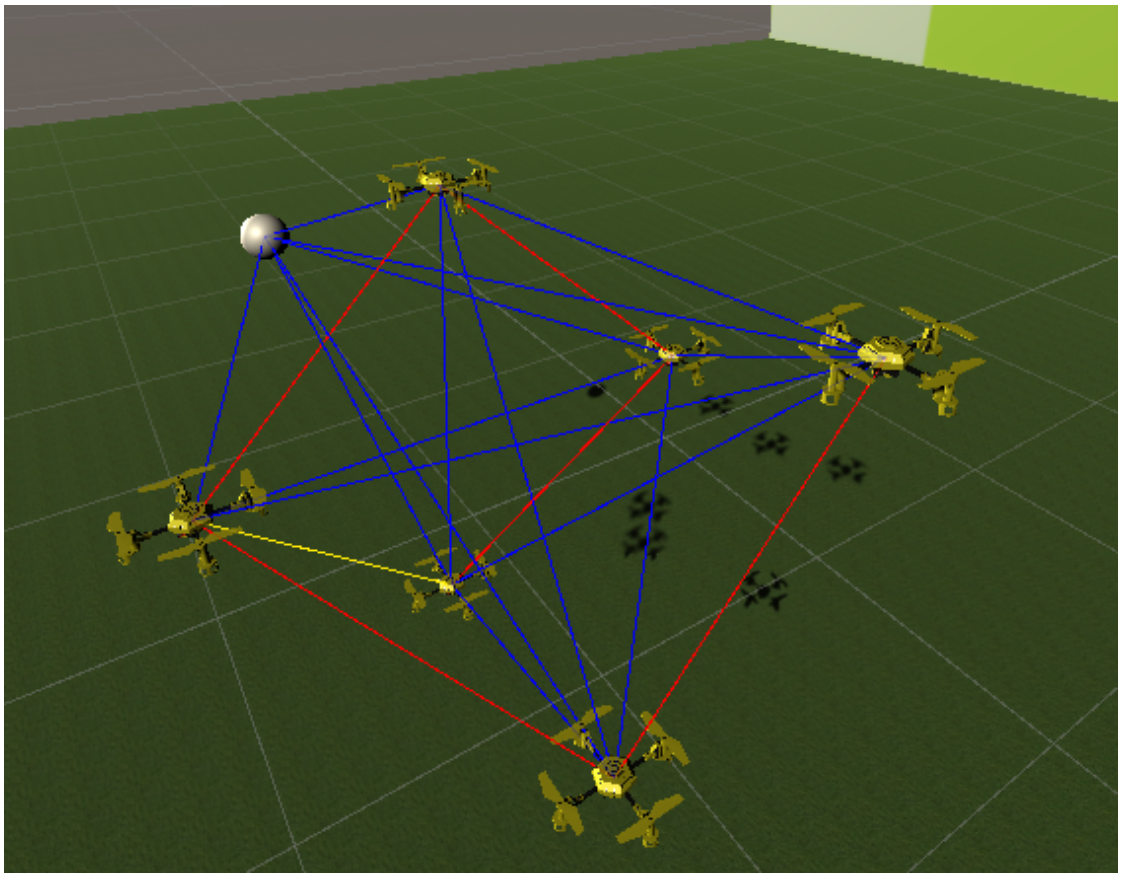

Figure 50

13 Appendix C Topology Unity Parameters

13.1 Average Vector Topology Parameters

Figure 51

13.2 Closest Neighbour Topology Parameters

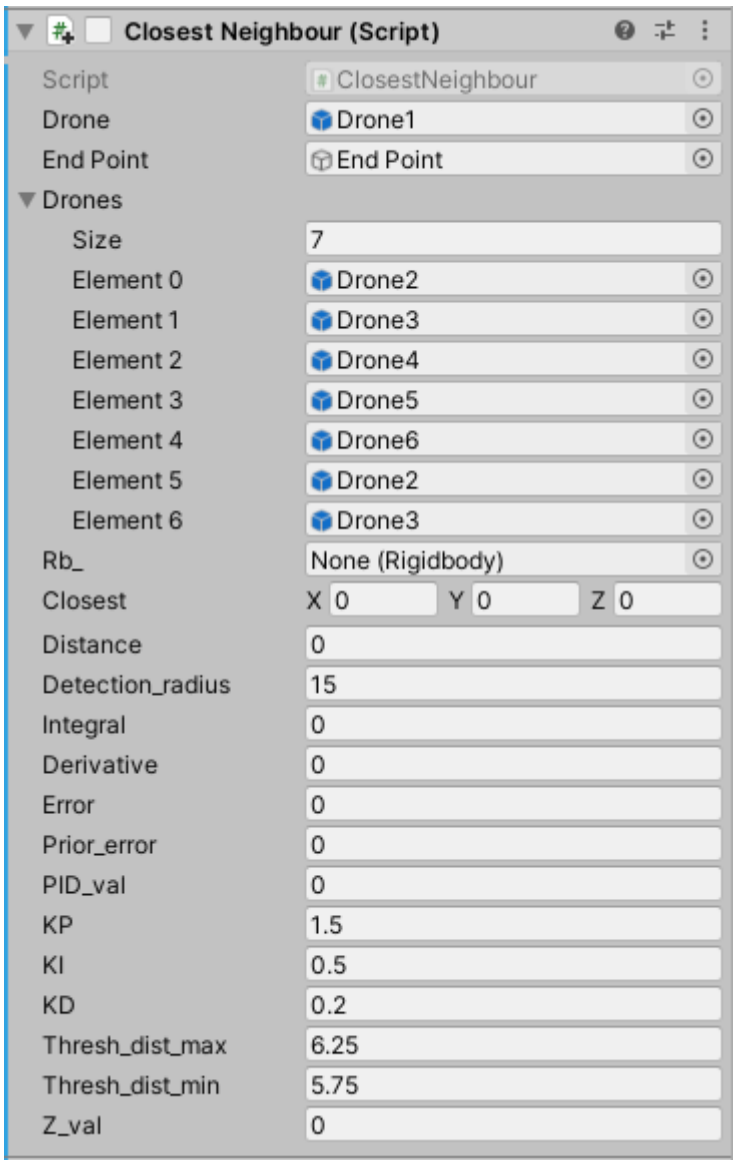

Figure 52

13.3 Vector All Topology Parameters

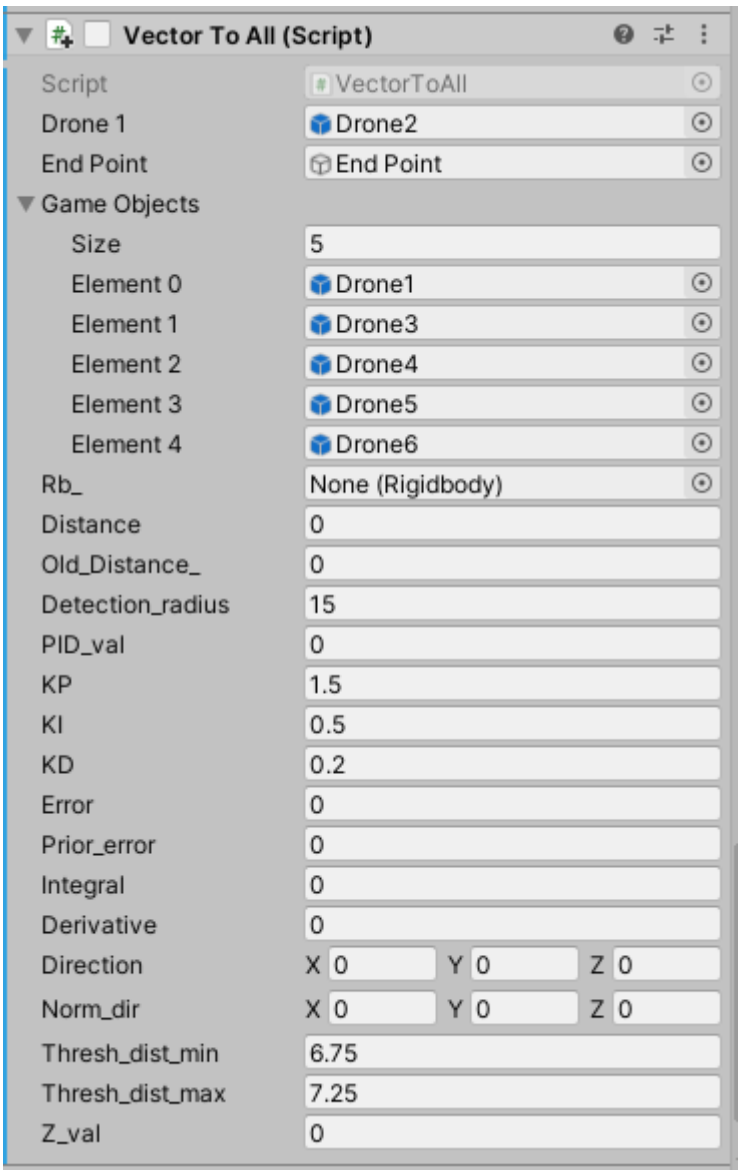

Figure 53

14 Appendix D MATLAB PID Tuning Script

```
goal = 10;
Iterations = 50;
current_val = 0;
op = [current_val];
data(1:1,1:Iterations) = goal;

Kp = 0.8;
Ki = 0.01;
Kd = 0.9;
t = 0.1;

e = 0;
prev_e = 0;
Integral = 0;
Derivative = 0;

for i = 1:Iterations-1

    e = goal - current_val;

    Integral = Integral + (t * e);
    Derivative = (e - prev_e) / t;
    prev_e = 0;
    prev_e = e; e = 0;

    output = (Kp * e) + (Ki * Integral) + (Kd * Derivative);

    current_val = current_val + output * t;
    op = [op current_val];

end

hold on;
plot (op,'DisplayName','PID Output');
plot (data,'DisplayName','Goal');
hold off;
```

Figure 54

15 Appendix E PID Controller and Box-Muller Methods

```
//PID Controller
////////////////////////////////////////////////////////////////////////////////////////////

float PID()
{
    float timedelta = 0.1f;
    Error = Distance - ((Thresh_dist_max + Thresh_dist_min) / 2);
    Integral += timedelta * Error; Integral = Integral * 0.9f;
    Derivative = (Error - Prior_error) / timedelta;
    Prior_error = Error;

    PID_val = Error * KP + KI * Integral + KD * Derivative;

    if (PID_val > 6) { PID_val = 6; }
    return PID_val;
}

float BoxMuller()
{
    var rand = new System.Random();
    float mean = 0.01f;
    float sd = 0.1f;
    float pi = 3.1415926535897931f;

    double z1 = RandFloat(0, (2*pi));
    double B = sd * Math.Sqrt(-2*(Math.Log(RandFloat(0,1))));

    double z2 = B * (Math.Sin(z1)) + mean;
    double z3 = B * (Math.Cos(z1)) + mean;

    if (z_val == 0)
    {
        z_val = 1;
        return (float)z2;
    }
    else
    {
        z_val = 0;
        return (float)z3;
    }
}

double RandFloat(double min, double max)
{
    var rand = new System.Random();
    return rand.NextDouble() * (max - min) + min;
}
```

Figure 55

16 Appendix F Emergent Swam C# Unity Scripts

16.1 Average Vector Script

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AverageVector : MonoBehaviour
{
    // Store Drone Objects in Lists
    public GameObject Drone1;
    public GameObject EndPoint;
    public List<GameObject> gameObjects = new List<GameObject>();
    public Rigidbody rb_;

    // Distance
    public float Detection_radius = 15f;
    public float Distance_;
    public List<Vector3> vector3s = new List<Vector3>();
    public float Thresh_dist_min = 5.75f;
    public float Thresh_dist_max = 6.25f;

    // PID Variables
    public float PID_val = 0;
    public float KP = 1.5f, KI = 0.5f, KD = 0.2f;
    public float Error, Prior_error;
    public float Integral, Derivative;

    // Vectors
    public Vector3 Direction_drones;
    public Vector3 Direction_endpoint;
    public Vector3 Norm_Dir;

    // Box-Muller
    public int z_val = 0;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        // Detect End Point within radius
        if (Vector3.Distance(Drone1.transform.position, EndPoint.transform.position) <= Detection_radius){
            Vector3 temp = EndPoint.transform.position - Drone1.transform.position;
            // Add noise to vector
            temp.x += BoxMuller(); temp.y += BoxMuller(); temp.z += BoxMuller();
            vector3s.Add(temp);
        }
        foreach (var d in gameObjects)
```

Figure 56

```
foreach (var d in gameObjects)
{
    float temp_dist = Vector3.Distance(Drone1.transform.position, d.transform.position);
    // Add noise to distance
    temp_dist += BoxMuller();

    if (temp_dist <= Detection_radius){

        Vector3 temp_dir = d.transform.position - Drone1.transform.position;
        // Add noise to vector
        temp_dir.x += BoxMuller(); temp_dir.y += BoxMuller(); temp_dir.z += BoxMuller();
        vector3s.Add(temp_dir);
        // Average vector now has noise

        // Draw debug rays to neighbouring drones within radius
        if (temp_dist < Thresh_dist_min){
            Debug.DrawRay(Drone1.transform.position, temp_dir, Color.red);
            Debug.Log("Too Close to Drone");
        }
        else if (temp_dist < Thresh_dist_max && temp_dist > Thresh_dist_min){
            Debug.DrawRay(Drone1.transform.position, temp_dir, Color.yellow);
            Debug.Log("Okay Distance from Drone");
        }
        else if (temp_dist > Thresh_dist_max){
            Debug.DrawRay(Drone1.transform.position, temp_dir, Color.blue);
            Debug.Log("Too Far Away from Drone");
        }
    }
}
```

Figure 57

```
    // End Point
    Distance_ = Vector3.Distance(Drone1.transform.position, EndPoint.transform.position);
    // Add noise to distance
    Distance_ += BoxMuller();
    Direction_endpoint = EndPoint.transform.position - Drone1.transform.position;
    // Add noise to vector
    Direction_endpoint.x += BoxMuller(); Direction_endpoint.y += BoxMuller(); Direction_endpoint.z += BoxMuller();
    Norm_Dir = Direction_endpoint;
    Norm_Dir.Normalize();

    if (Distance_ <= Detection_radius){
        if (Distance_ < Thresh_dist_min)
        {
            Debug.Log("Too Close to End Point");
            Debug.DrawRay(Drone1.transform.position, Direction_endpoint, Color.red);
            Movement(rb_, Norm_Dir);
        }
        else if (Distance_ < Thresh_dist_max && Distance_ > Thresh_dist_min)
        {
            Debug.Log("Okay Distance from End Point");
            Debug.DrawRay(Drone1.transform.position, Direction_endpoint, Color.yellow);
            Movement(rb_, Norm_Dir);
        }
        else if (Distance_ > Thresh_dist_max)
        {
            Debug.Log("Too Far Away from End Point");
            Debug.DrawRay(Drone1.transform.position, Direction_endpoint, Color.blue);
            Movement(rb_, Norm_Dir);
        }
    }
}

void Movement(Rigidbody rb_, Vector3 Norm_Dir)
{
    rb_ = GetComponent<Rigidbody>();
    if (rb_)
    {
        PID_val = PID();
        rb_.AddForce(Norm_Dir * PID_val);
    }
}
```

Figure 58

16.2 Closest Neighbour Script

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ClosestNeighbour : MonoBehaviour
{
    public GameObject Drone;
    public GameObject EndPoint;
    public List<GameObject> Drones = new List<GameObject>();
    public Rigidbody rb_;

    public Vector3 closest;
    protected Vector3 Norm_Dir;
    protected Vector3 Direction_endpoint;
    public float Distance;
    public float Detection_radius = 15f;

    public float Integral, Derivative, Error, Prior_error, PID_val;
    public float KP = 1.5f, KI = 0.5f, KD = 0.2f;
    public float Thresh_dist_max = 6.25f;
    public float Thresh_dist_min = 5.75f;
    public int z_val = 0;

    // Start is called before the first frame update
    void Start()
    {
        rb_ = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void Update()
    {
        float prev_temp = 0, temp = 0;
        foreach (var d in Drones)
        {
            // Find closest drone
            if (Drones.Count > 1)
            {
                temp = Vector3.Distance(Drone.transform.position, d.transform.position);
                if (temp < prev_temp)
                {
                    closest = (d.transform.position - Drone.transform.position);
                    Distance = temp;
                }
                prev_temp = temp;
            }
            else
            {
                closest = (d.transform.position - Drone.transform.position);
                Distance = Vector3.Distance(Drone.transform.position, d.transform.position);
```

Figure 59

```
        }
    }

    if (Drones.Count != 0)
    {
        // Add white noise to vector values
        closest.x += BoxMuller(); closest.y += BoxMuller(); closest.z += BoxMuller();
        Norm_Dir = closest; Norm_Dir.Normalize();
        // Add white noise to distance measurement
        Distance += BoxMuller();

        // Move away from closest drone
        if (Distance <= Detection_radius)
        {
            if (Distance < Thresh_dist_min)
            {
                Debug.Log("Too close to drone");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, closest, Color.red);
            }
            else if (Distance < Thresh_dist_max && Distance > Thresh_dist_min)
            {
                Debug.Log("Okay Distance From Drone");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, closest, Color.yellow);
            }
            else if (Distance > Thresh_dist_max)
            {
                Debug.Log("Too far from Drone");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, closest, Color.blue);
            }
        }
    }
```

Figure 60

```
    if (EndPoint != null)
    {
        // Calculate distance to end trajectory
        Distance = Vector3.Distance(Drone.transform.position, EndPoint.transform.position);
        // Add white noise to distance measurement
        Distance += BoxMuller();
        Direction_endpoint = (EndPoint.transform.position - Drone.transform.position);
        // Add white noise to vector valuse
        Direction_endpoint.x += BoxMuller(); Direction_endpoint.y += BoxMuller(); Direction_endpoint.z += BoxMuller();
        Norm_Dir = Direction_endpoint; Norm_Dir.Normalize();

        if (Distance <= Detection_radius)
        {
            if (Distance < Thresh_dist_min)
            {
                Debug.Log("Too close to end point");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, Direction_endpoint, Color.red);
            }
            else if (Distance < Thresh_dist_max && Distance > Thresh_dist_min)
            {
                Debug.Log("Okay Distance From End Point");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, Direction_endpoint, Color.yellow);
            }
            else if (Distance > Thresh_dist_max)
            {
                Debug.Log("Too far from end point");
                Movement(Norm_Dir);
                Debug.DrawRay(Drone.transform.position, Direction_endpoint, Color.blue);
            }
        }
    }
}

void Movement(Vector3 dir)
{
    if (rb_)
    {
        PID_val = PID();
        rb_.AddForce(dir * PID_val);
    }
}
```

Figure 61

16.3 Vector All Script

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class VectorToAll : MonoBehaviour
{
    //Store Drone Objects in Lists
    public GameObject Drone1, EndPoint;
    public List<GameObject> gameObjects = new List<GameObject>();
    public Rigidbody rb_;

    //Distance
    public float Distance, old_Distance_, Detection_radius = 15f;

    //PID Variables
    public float PID_val = 0;
    public float KP = 1.5f, KI = 0.5f, KD = 0.2f;

    public float Error, Prior_error;
    public float Integral, Derivative;

    public Vector3 Direction;
    public Vector3 norm_dir;

    // //Threshold Variables
    public float Thresh_dist_min = 6.75f;
    public float Thresh_dist_max = 7.25f;

    public float z_val = 0;

    // Start is called before the first frame update
    void Start()
    {

    }
```

Figure 62

```
// Update is called once per frame
void Update()
{
    foreach (var Drone2 in gameObjects)
    {
        // Measure distance from Drones
        Distance = Vector3.Distance(Drone1.transform.position, Drone2.transform.position);
        // Add noise to distance
        Distance += BoxMuller();

        // Is Drone within sensor radius
        if (Distance <= Detection_radius)
        {
            // Grab Direction vector and draw
            Direction = Drone2.transform.position - Drone1.transform.position;
            // Add noise to vector
            Direction.x += BoxMuller(); Direction.y += BoxMuller(); Direction.z += BoxMuller();
            norm_dir = Direction; norm_dir.Normalize();

            if (Distance < Thresh_dist_min)
            {
                Debug.Log("Too Close to Drone");
                Movement(norm_dir);
                Debug.DrawRay(Drone1.transform.position, Direction, Color.red);
            }
            else if (Distance < Thresh_dist_max && Distance > Thresh_dist_min)
            {
                Debug.Log("Okay Distance From Drone");
                Debug.DrawRay(Drone1.transform.position, Direction, Color.yellow);
                Movement(norm_dir);
            }
            else if (Distance > Thresh_dist_max)
            {
                Debug.Log("Too far from Drone");
                Movement(norm_dir);
                Debug.DrawRay(Drone1.transform.position, Direction, Color.blue);
            }
        }
        else { Debug.Log("No Drone within Radius"); }
    }
```

Figure 63

```
    // Measure distance from End Point
    Distance = Vector3.Distance(Drone1.transform.position, EndPoint.transform.position);
    // Add noise to distance
    Distance += BoxMuller();

    // Is End Point within sensor Radius
    if (Distance <= Detection_radius)
    {
        // Grab Direction vector and draw
        Direction = EndPoint.transform.position - Drone1.transform.position;
        // Add noise to vector
        Direction.x += BoxMuller(); Direction.y += BoxMuller(); Direction.z += BoxMuller();
        norm_dir = Direction; norm_dir.Normalize();

        if (Distance < Thresh_dist_min)
        {
            Debug.Log("Too Close to End Point");
            Movement(norm_dir);
            Debug.DrawRay(Drone1.transform.position, Direction, Color.red);
        }
        else if (Distance < Thresh_dist_max && Distance > Thresh_dist_min)
        {
            Debug.Log("Okay Distance From End Point");
            Debug.DrawRay(Drone1.transform.position, Direction, Color.yellow);
        }
        else if (Distance > Thresh_dist_max)
        {
            Debug.Log("Too Far From End point");
            Movement(norm_dir);
            Debug.DrawRay(Drone1.transform.position, Direction, Color.blue);
        }
    }
    else { Debug.Log("No End Point within Radius"); }
}

void Movement(Vector3 norm_dir)
{
    rb_ = GetComponent<Rigidbody>();
    if (rb_)
    {
        PID_val = PID();
        rb_.AddForce(norm_dir * PID_val);
    }
}
```

Figure 64