UNIVERSITY OF CALGARY

Flail: A Domain Specific Language for Drone Path Generation

by

Flavia Roma Cavalcanti

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2019

Abstract

The main objective of this thesis was to design a domain specific language that would allow users to easily describe flight trajectories for drones. Conventional drone control schemes rely on handheld controllers and, sometimes, on model specific applications that allow users to pre-plan paths (e.g., FreeFlight Pro). The issue with these is that handheld controllers display a large learning curve for new users and flight plan applications rely on waypoint systems, which limits the complexity of the flight plan. Flail is an alternate control scheme for drones that is capable of programmatically pre-specifying complex flight patterns. Additionally, an HTC Vive was used to simplify Flail code generation by allowing users to use the Vive wand to draw out flight trajectories in 3D space. The viability of Flail was examined through RC drone flight tests and simulations.

Preface

This thesis is an original work by the author. No part of this thesis has been previously published.

To all the time I've wasted...

Table of Contents

List of Figures and Illustrations

List of Tables

Chapter 1

Introduction

This work aims to create a methodology to address the problem of motion planning (also known as the navigation problem) necessary in robotics applications. The goal is to find - and follow a path, in the form of a sequence of valid configurations (positions and orientations) that moves a controllable object, such as a car or a drone, from an initial position to a final destination.

1.1 Motivation and Process

Interest in recreational uses for Unmanned Aerial Vehicles (UAVs, or as they're more commonly known as, drones) has skyrocketed, as many hobbyists have begun to use drones in their day-to-day lives. The most common applications involve aerial photography and filming. Even with the recent appeal of drones for casual usage, there is still a learning curve when using them. Crashes and collisions are common among new drone pilots [1]. The issue appears to lie within the default drone control[1] configuration, which defaults to two joystick controllers that have to be used simultaneously. The joysticks control a number of drone functionalities including ascension, descension, left yaw, and right yaw, as well as forward, backward, left, and right movement (Figure 1.1).

[1]Control in this thesis refers to user defined input that conducts a drone through a specified path.

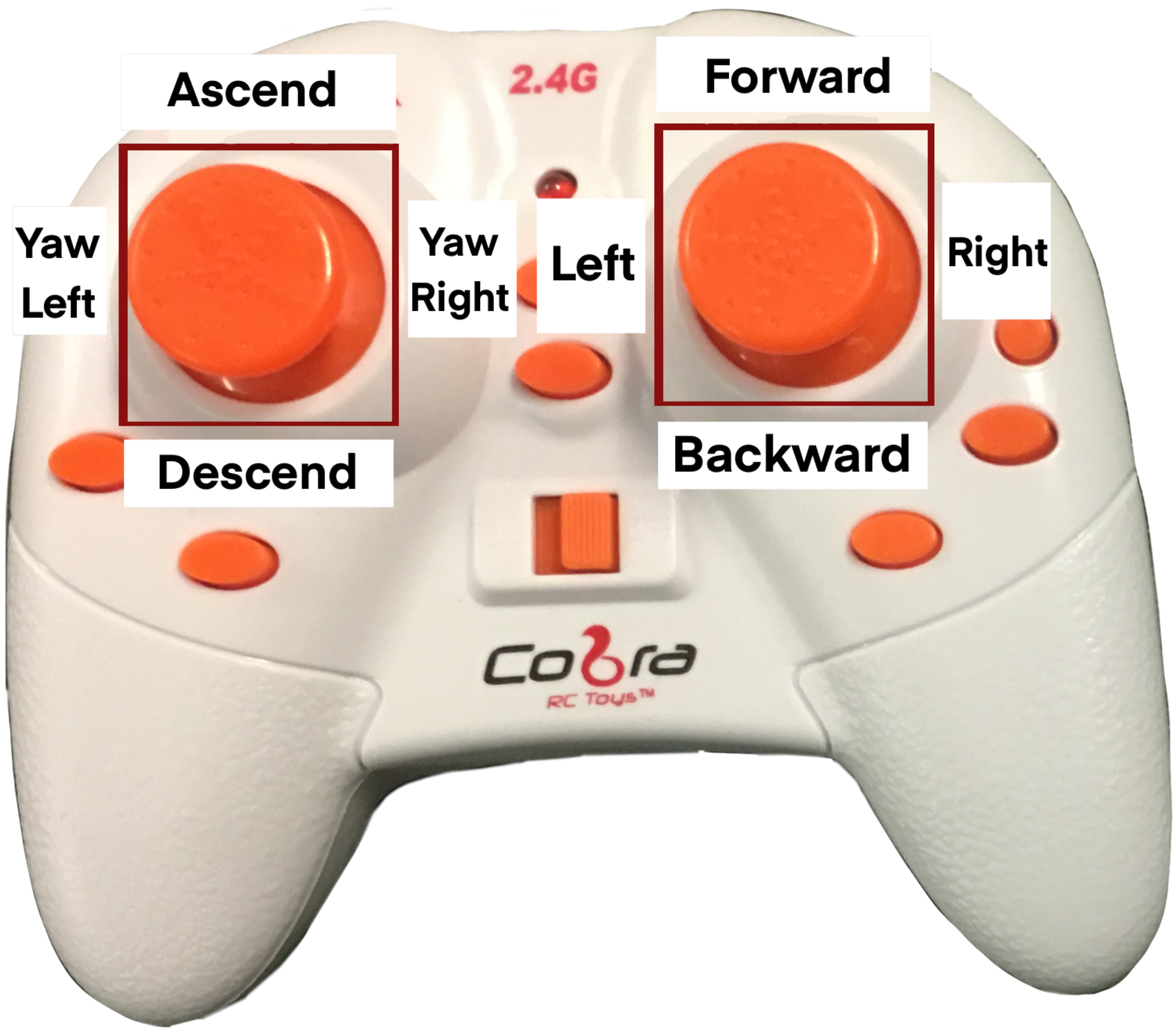


Figure 1.1: Cobra RC microdrone controller with labeled directions

Therefore, The motivation for this work stems from the difficulty of defining complex drone flight trajectories, without having previous experience in drone piloting. Some companies have been successful in coordinating the flight pattern of many drones simultaneously, Intel's Shooting Star drones being a recent example [2]. During the 2018 Olympics, Intel flew over two thousand drones into the sky to form images and animated sequences. (The drones have not yet been released to the public, nor has any information been given out regarding their implementation and development.)

Figure 1.2: Intel Shooting Star Drones light show at California Bear over Travis Air Force Base, Calif., July 5, 2018. (U.S. Air Force photo by Tech. Sgt. James Hodgman) [3]

With this, a question arose. Would it be possible to design a method that would allow users to easily direct their drones to follow specific flight patterns?

To address this challenge, a domain specific language (DSL) was designed that would allow users to do so. The language was dubbed "Flail" (**Fla**via's **I**mplemented **L**anguage) and was developed with the idea of aiding users in pre-planning precise flight trajectories. These routes can be re-used in the future, which allows users to both construct and repeat flights even if they lack experience in piloting drones.

1.2 Research Goals

The overarching research question addressed in this work is: **How can we design a programming language that allows us to clearly describe movement paths for drones?** This question can be decomposed into four more specific questions:

Research Question 1 (RQ1): How should we design a time-based programming language that allows us to describe movement paths for drones? [Chapter 3]

Research Question 2 (RQ2): How should we design a distance-based programming language that allows us to describe movement paths for drones? [Chapter 4]

Research Question 3 (RQ3): How can we use gestures to simplify the generation of Flail code? [Chapter 5]

Research Question 4 (RQ4): What changes, if any, are necessary in order to get Flail to run on a commercial drone? [Chapter 6]

1.3 Contributions

The main contributions of this work are:

1. The creation of a domain specific programming language, which offers an abstraction of the default drone controller functions into a set of commands (Section 3.1 and Section 4.1).

2. The implementation of a simulator, using the Unity graphics engine, for the interpretation of Flail code without requiring an expensive real drone (Section 3.3 and Section 4.4).

3. Technical details on how to modify the hardware of a Cobra RC controller to accept communication from an Arduino, which allows Flail code to control the microdrone (Section 3.2.1).

4. The implementation of a path generator based on parametric curves, by using the paradigm of first person piloting. This generator aids users in creating Flail code based on more complex flight routes (Section 4.3).

5. The interactive creation of paths in a laboratory using the Vive hardware for Flail code creation (Chapter 5).

6. Assessing the portability of Flail code by generating output for a Ground Station (Chapter 6).

1.4 Overview

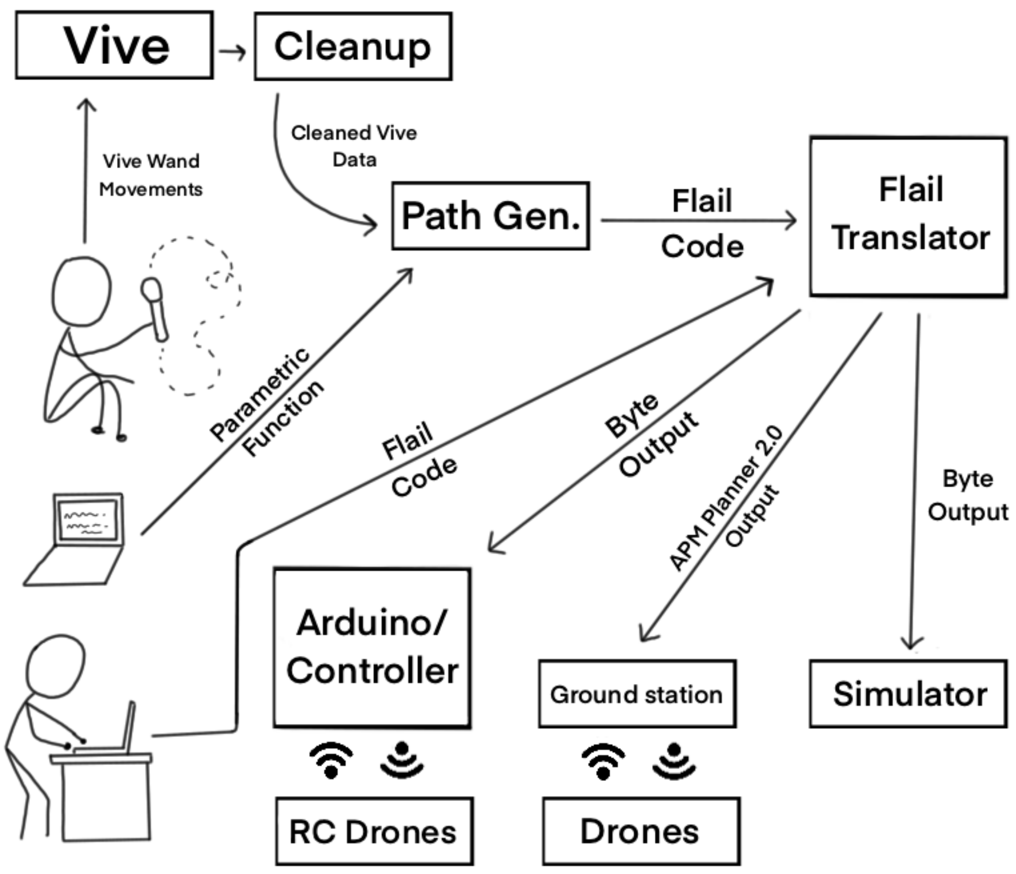


Figure 1.3: Workflow for this research

This work is structured in the following manner; Chapter 2 will provide the related work that motivated this research, Chapter 3 will cover the implementation of the time based version of Flail and its use on RC microdrones, and Chapter 4 will cover the implementation of the distance based version of Flail and the development and use of simulated test flight patterns. Then, Chapter 5 will describe the use of gesture to generate Flail code, and Chapter 6 will cover the portability and generation of flight paths for commercial drones. Finally, Chapter 7 will conclude this work, by describing its overall contribution.

Figure 1.3 displays the workflow of this work. Each chapter will cover a part of the workflow, where specific sections will be highlighted to demonstrate the sections being covered in that chapter.

Chapter 2

Background and Related Work

This chapter provides background information of work that relates to my chosen research topic. This work is distributed amongst five distinct categories:

1. Traditional Methods to Control Drones

   This category covers the most common methods of drone control. These usually involve a variety of hand held controllers, with varying capabilities.

2. Programming languages for drones

   There are a number of programming languages that are drone specific. These might be domain specific languages or language libraries that allow drone connections.

3. Human robot interaction

   This category covers the interaction between humans and robots, with an emphasis on moving robots.

4. Programming by example

   Programming by demonstration (PbD) is a method where users are capable of "teaching" robotic mediums to follow tasks. The path generation aspect of this research can be considered a simplified version of PbD, as users demonstrate their desired paths without manually programming each individual trajectory.

5. Non-traditional methods to control drones

   Non-traditional methods for drone control is an active field of research. These involve different control schemes that are not dependent on default handheld controllers. Emphasis will be given to exocentric/egocentric control and to semi-autonomous drones. Flail can arguably be categorized as a non-traditional, programmatic method for drone control.

2.1 Traditional Methods to Control Drones

Drones are most commonly controlled by a transmitter/remote control, a handheld device that allows users to maneuver and adjust the setting of their UAV. Some drones might also come with a software development kit, or SDK, that allows users to programmatically control their drones.

2.1.1 Controller Based Schemes

The use of controllers is the most direct way to pilot drones; nonetheless, they present a certain learning curve to new users. Drone movement is controlled by a pair of joysticks that have to be used simultaneously to keep the drone aloft in a stable manner. Apart from requiring a substantial learning step, this type of interface limits its user base as mobility impaired people would be unable to manipulate the joysticks effectively.

For simplicity, I'm going to follow the general drone setup controller scheme where:

- The right joystick controls yaw and pitch – moves the drone left/right, forwards/backwards.
- The left joystick controls yaw and throttle – rotates the drone clockwise/counterclockwise and ascends/descends the drone.

2.2 Programming Languages for Drones

Programmatic drone control is an alternate method for users to interact with their drones. This method allows users to control their drones by writing programs.

2.2.1 Domain Specific Languages for Drones

A domain specific language (DSL) is a programming language that is specialized in a specific platform or domain [4]. Certain DSLs allow users to programmatically control the flight trajectories or general capabilities of drones.

In 2015, Pinceroli et al. released Buzz, An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms [5]. Traditionally, programmers are required to focus on individual entities of the swarm and encode their low level behavior to create a swarm. By contrast, the approach taken by Buzz is to expose only the relevant parts of the problem to solve, working with swarm formation and modification in an abstract manner.

Buzz lets programmers subdivide swarms into distinct groupings, treating them like first class objects. Currently, no standardized platform exists that allow researchers to share and reuse swarm behaviors. The design of Buzz is motivated to overcome the traditional methods to interact with swarm behavior.

To the best of our knowledge, Buzz does not emphasize movement design and is mainly suitable for swarm control. Buzz might be a starting point for the addition of swarming capabilities to future versions of Flail.

Visual languages are another option for programmatic control. Tynker's Parrot Drone app [6], which has the end goal to help children learn how to code, is an example. The application uses a block based visual language to direct the actions of the paired drone. These commands vary from simple flight instructions, such as take off and move forward, to basic logical concepts, such as looping and trigger events. Trigger events generally correspond to changes in the drone's state, such as successful landings, take offs, ect...

Tynker's approach to path specification is fairly similar to Flail's, where simple instructions can be combined to move the drone in specific ways. Flail currently does not have trigger events. Flail, on the other hand, is able to generate paths based on user movements and is able to create trajectories based on specific distances. Furthermore, a Flail program can be passed on to a number of different drones, and is not limited to a single model or brand.

2.2.2 Other Languages for Drones

Another possibility for programmatic control is to either add libraries or to extend already existing programming languages. This benefits users who are already familiar with the language, as they would not be required to learn new syntax in order to use the drone capabilities.

A common occurrence is to use Mavlink or Ardupilot systems as intermediaries between the language and the drone. Mavlink [7] is a lightweight messaging protocol system used to communicate with small unmanned vehicles. It is most commonly used between a ground control station (GCS) and unmanned vehicles. Mavlink is notably able to keep track of the orientation of vehicles, their GPS location, and speed. Ardupilot [8] is open source software known as an autopilot suite, capable of controlling a number of different unpiloted vehicles.

A language example that uses Mavlink as a communication medium is Drone-kit Python [9]. This Python API allows developers to create applications that run on onboard companion computers (such as the Raspberry Pi) to communicate with the Ardupilot flight controller using a low latency link. The API communicates with vehicles over MAVlink and as such is compatible with any vehicles that communicate using the MAVlink protocol.

Position, speed, and other attributes can be accessed through the vehicle class. The general position of the drone is represented by latitude and longitude coordinates relative to the World Geodetic System 1984 (WGS84) coordinate system [10], with its altitude relative to mean sea-level (MSL).

There is another well known Python API for drone control, known as the PS-Drone [11]. Differently from Drone-kit Python, this API is only capable of programming the AR.Drone 2.0. It was designed to be easy to learn while still offering the full set of functionalities of the AR.Drone 2.0 including Sensor-Data (NavData), Configuration and Video-support.

Codeminders, on the other hand, released a Java API for AR drones [12] with the goal of using Java to control Parrot's AR.Drone without the use of native code. This project did not use Parrot's SDK, but instead implemented networking protocols directly in Java. Most of the control scheme comes from ControlTower, a Java GUI that uses Playstation 3 controllers and a keyboard to control the drone.

ROS framework for the Bebop quadcopter [13] is an autonomous navigation framework for the Bebop Quadcopter by Parrot Corporation. ROS, or Robot Operating System, is robotics middleware [14], a set of software libraries that help users build robot applications.

Node AR drone [15] is an open source project that allows users to program the AR drone using node.js. This project allows access to drone capabilities, such as its onboard camera and flight control. Animation sequences are given in terms of a duration (in milliseconds) and a command, e.g., `animate('turnaround', 1000)`. Instructions can be set as the drone is in flight by sending over commands along with a speed value, e.g. `forward: 0.5`, which makes the drone fly forwards at half its maximum speed. The control scheme used by Node AR drone is similar to the time-based version of Flail. It is a time based scheme, where instructions can be set and executed for a given duration.

2.3 Human Robot Interaction

Human Robot Interaction (HRI) is an area of study related to people's behavior and attitudes towards a robot's interactive features. There are many different ways to address robotic expression and interaction, but this section will emphasize interactions between humans and robots in motion, with priority given to Human Drone Interaction (HDI) and how it pertains to motion planning.

There is a relatively recent survey paper (2013) by Hoffman et al. [16] regarding HRI that includes a section on social robot navigation that is of interest. Mirri et al. [17] also developed a survey paper regarding the current issues and challenges related to HDI, with 40 papers published between 2012 and 2019.

A common method to conduct robots is robotic teleoperation, or manually operated robots. This method, however, comes with a pitfall as it may lead to increased cognitive burden on the operator. Even so, Hooman et al. [18] state that this form of robotic operation affords a higher level of precision than most. They presented new interfaces that better support perspective-taking for aerial robots. They proposed a framework for structuring the design of AR interfaces that support HRI and used this framework to develop teleoperation models that leverage ARHMD technology to provide visual feedback on camera capabilities. Their work was shown to significantly improve objective measures of teleoperation performance and speed while reducing crashes.

Hoffman et al. [19] developed a design process for interactive robots that bears expressive movement at its center, arguing that robotic movement is essential to developing human-robot comprehension. Their case studies included a wide range of expressive-center robots including Shimon, a robotic mariba player; Travis, a robotic speaker dock listening companion; and a set of telepresence and animated robots. Each robotic medium was tested by how easily people were able to understand their gestures in different scenarios. The authors' conclusion is that a focus on movement design for interactive robots has the potential to bring a breakthrough to robot integration in human society.

Motion is also useful in the communication of affective information. Sharma et al. [20] explored how a flying quadrotor can modify its locomotion path to communicate affect to people. The authors presented an adaptation of the Laban Effort System as a way to aid the creation of expressive robots. They concluded with preliminary guidelines for designing expressive robotic locomotion paths.

Similarly, Wojciechowska et al. [21] considered how a drone's way of conducting itself affected how it was perceived by its user. Parcel delivery drones, for example, would have to approach people in a different manner compared to a search-and-rescue drone. Their paper presented a taxonomy for HDI user studies which surveys different methodologies from HDI literature; investigating how proximity, speed, direction, and trajectory led to a comfortable drone approach.

Aesthetically pleasing trajectory designs, on the other hand, are still difficult to create when asked to respect the physical limitations of robots, especially when filming in dynamic environments. Nageli et al. [22] proposed a method for the real-time generation of multi-drone aerial cinematography motion plans. The authors propose a method for planning aerial videography in clustered dynamic environments. Their method takes high-level plans alongside image-based framing objectives as inputs. The input paths do not have to be physically feasible as their method only uses the path for guidance.

2.4 Programming by Demonstration

Robot Learning from Demonstration [23] (LfD) or Robot Programming by Demonstration (PbD) is a paradigm that allows robots to learn how to follow certain actions. This technique approaches robot programming as something that can be extracted from simple observations, or more specifically, the observation of human performances of certain movements, rather than from the manual decomposition and programming of such movements and behaviors. As such, the main idea of PbD is to allow users to teach robots new tasks without programming them.

In the early 1980s there was an increased interest in PbD for robotics [24]. In its initial developmental stages, researchers used symbolic reasoning such as guiding or playback methods, which relied on teleoperated or manual control, to perform PbD [25, 26].

The ideals of PbD are used to facilitate the programming of robotic functions. Consider a robot whose job is to prepare orange juice [23]. The task is comprised of a series of sub-tasks such as cutting the orange, squeezing it in a cup, throwing the orange into the trash, and serving the juice into a cup. As an added difficulty, the robot might also have to deal with items being moved from their original locations.

Traditionally, a programmer would have to reason out a method that would allow the robot to respond to changes in its scenario, no matter how unlikely. In contrast, PbD techniques allow users to "program" the robot by demonstrating how it should perform certain actions. If new failures arise, the users could then just perform new demonstrations. LfD-PbD are not a record and replay technique, as it emphasizes learning at its core.

According to the Handbook of Robotics [24], there are three main benefits to using user based interfaces for training robots to perform desired tasks. The first is that PbD or imitation learning is a highly powerful tool in reducing the complexity of the search spaces for learning. In other words, it is possible to reduce the search for ideal solutions when sifting through a series of good or bad examples, by eliminating bad solutions from the search space. Second, PbD minimizes the tedious task of manual programming, making interaction with robots an accessible area to lay people.

Yu Wang Liu et al. [27] presented a novel augmented reality (AR) based interface for motion planning in a 3-axis glue dispenser via programming by demonstration. This interface allows users to efficiently determine and demonstrate the dispenser motion in a demonstration task. The planning process is conducted within a virtual environment while allowing them to determine the spatial relationship between the machine's tip and the work plane of a part. This prevents collisions between the machine and the work part and accelerated positioning the dispenser tip in 3D space.

Andre Gaschler et al. [28] presented a novel augmented reality system for defining virtual obstacles, specifying tool positions, and robot tasks. AR was used as an aid to the user's spatial perception and to help them better understand the robot's capabilities. The user is given a handheld pointing device for specifying waypoints for the robot and to trigger corresponding actions.

Flail follows a simplified version of PbD, where the user “teaches” the drone how to follow a desired path by sketching out trajectories. More complex versions of PbD generalize the user’s demonstrations to form a robust program capable of dealing with new situations [29].

2.5 Non-traditional Methods to Control Drones

Considerable prior work has studied different manners in which users can interact and control their drones. These techniques move away from the traditional drone joystick controller, and often blend in aspects of augmented reality (AR) or virtual reality (VR).

2.5.1 Egocentric vs Exocentric Views

The usual method of piloting drones involves steering with a joystick-based controller, where users can take an egocentric or exocentric approach to piloting. Exocentric mode occurs when the pilot observes the drone from the ground while steering; and egocentric, or first person viewpoint, occurs when the pilot flies based on the drone’s onboard camera stream [30]. It is difficult to pilot in exocentric mode when occlusions are present, and it becomes nearly impossible when the drone is lost from view. Switching over to egocentric mode, however, brings about its own set of limitations due to the drone’s narrow field of view, which reduces the user’s own perception of space. Both of these limitations further increase the risk of crashes, especially for novice pilots.

Erat et al. [30] attempted to reduce flight hindrances that occur in traditional exocentric piloting, by using VR to provide a synthetic exocentric view that allows users to better visualize occlusions. This method dynamically adds a 3D model of the occluded territory to the live video streamed by the drone. This addition redefines a user’s ability to remotely pilot a drone by removing limitations to the user’s view point in occluded space. AR can also be used to make the occluder partially transparent, giving the impression of X-ray vision.

Kwangsu Cho et al. [31] addresses the issue of drone crashing caused by external piloting. Their hypothesis is that perspective misalignment occurs when externally piloting the drone, due to the difficulty of relating the user's egocentric perspective to the drone-centric perspective. To address such an issue, they developed an egocentric interface that allowed pilots to operate the drone from their own perspective regardless of the direction in which the drone was flying. Their research demonstrated that removing the cognitive load of mental rotations, generated by the process of aligning two different perspectives, increases a novice user's piloting success rate.

It is noticeable that the use of egocentric/exocentric views demonstrate interesting applications in drone control systems. Nonetheless, since this work does not involve visual piloting, the use of AR and VR techniques were out of scope. These techniques can be visited in future work.

2.6 Semi-autonomous Aerial Vehicles

An alternate approach to drone control is to use semi-autonomous properties as an aid, which in this context is taken as a drone that has some autonomous properties but still requires a human operator for its effective navigation. Such properties might involve collision detection or environmental recognition, e.g., automatic detection of burning buildings.

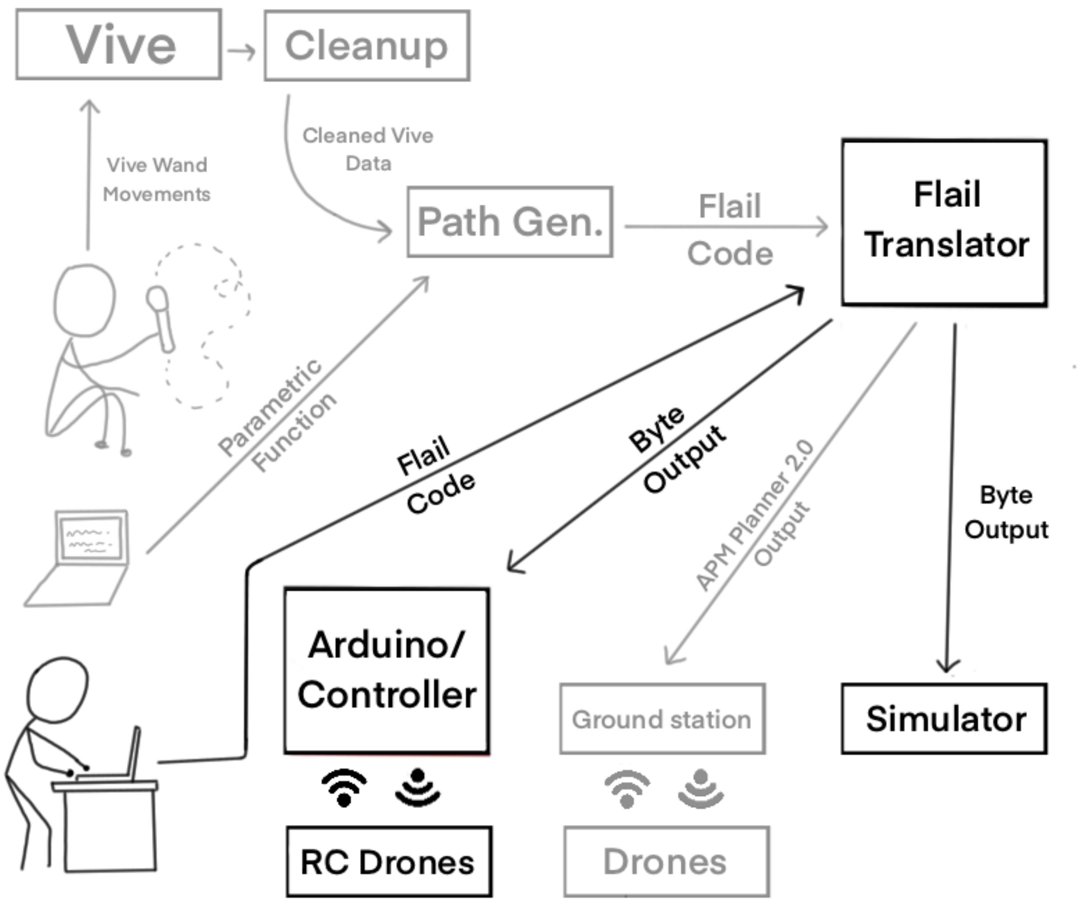
Nico Li et al. [32] designed Flying Frustum, a 3D spatial interface that provides a remote operator an increased level of human-UAV interaction and situational awareness. This interface functions by using pen interaction on a physical model of a terrain, and that spatially situates the information streaming from the UAVs onto the physical model.

Abeer Imdoukh et al. [33] developed an indoor fire proof UAV capable of handling dangerous search and rescue operations within burning buildings. It is capable of carrying a fire extinguisher and of searching for survivors in burning areas. The drone has mounted cameras to aid their pilots in their navigation, allowing firefighters to pilot it at a safe distance.

The use of semi-autonomous properties in drone control is appealing for future developments in Flail, more specifically, the implementation of collision detection during flights. The drone could, for example, detect possible collisions and correct its flight trajectory to prevent crashes while still maintaining its original course.

Chapter 3

Flail: Time-based Motion



Flail is an alternate control scheme for drones and is capable of programatically pre-specifying flight patterns for different vehicles. Flail is founded upon turtle graphics, a key feature in the LOGO programming language [34]. Turtle graphics are vector graphics that use a relative cursor to specify movement. The beauty of turtle graphics stems from its ability to combine a set of simple commands to generate complex paths. Flail's abstraction of drone movements is similar to the basic commands found in turtle graphics. Flail allows drones to move forward, backwards, left, right, and to rotate about their z-axis. I added an ascension and descension command to control the drone's altitude during its flight.

As described in previous chapters, the goal of this research is to develop a language capable of describing flight trajectories to different drones. This chapter will cover the first stage of this implementation, with the goal of answering RQ1: "How should we design a time-based programming language that allows us to describe movement paths for drones?"

There are a number of different ways to describe motion. One way is to focus on how long each displacement took: the vehicle moved forwards for 3 seconds, then it moved to the left for 2 seconds, and then backwards for 1 second. The time-based version of Flail, follows this idea: it is a time-based method of describing movements in 3D space.

The first step to design such a language was to answer the following questions:

RQ1.1 How do we describe drone movements as instructions?

Drones are only capable of limited movements, as they have four degrees of freedom: up and down movements (heaving), left and right (swaying) movements, forward and back movements (surging), and left and right rotations (yawing) [35]. The most direct approach is to describe each of these motions as an instruction: Forward, Backward, Left, Right, Ascend, Descend, Rotate Left, and Rotate Right. These instructions re-create the possible motions executed by a drone controller joystick.

RQ1.2 What time-based functions do we need?

When we describe motions based on time, we need to know how long each motion takes. Should we describe this in terms of minutes? Seconds? Milliseconds? It is common to describe time in short bursts, usually in terms of milliseconds and seconds,

Command	Meaning	Example
PENDOWN	Lower the pen down (to draw a line)	`PENDOWN`
PENUP	Lift the pen up (to not draw a line)	`PENUP`
FORWARD n	Move the turtle forward n steps	`Forward 50`
BACKWARD n	Move the turtle backwards n steps	`Backward 30`
LEFT n	Turn the turtle left by n degrees	`Left 90`
RIGHT n	Turn the turtle right by n degrees	`RIGHT 45`
REPEAT n ENDREPEAT	Commands in between these lines should be repeated n times	`REPEAT 4` `FORWARD 20` `RIGHT 90` `ENDREPEAT`

Table 3.1: Partial list of LOGO commands [37]

which is the approach taken by the time-based version of Flail. Two wait functions were added: wait(), which takes an integer value for seconds, and `waitMili()`, which takes an integer value for milliseconds.

This chapter will focus on developments of Flail. At this stage only manually written Flail programs are accepted as inputs, and the Flail translator is only capable of outputting byte files.

3.1 Flail Structure

The initial approach for Flail was based on the LOGO programming language. LOGO, more widely known for its use of turtle graphics, is an educational programming language developed in 1967 as a means to teach children computer programming [36]. In LOGO, a cursor, also known as a turtle, can be controlled through a set of commands (as shown in Table 3.1), allowing it to move around the screen (2D space) in a number of different ways. The beauty of LOGO is that even though its commands are simple, it is still capable of generating complex patterns.

The turtle used in LOGO was initially conceived as a turtle-shaped robot with a pen attached to its head, that could be lowered and raised as required. The turtle was the ideal medium to teach children programming, as they could easily figure out where the front of

Command	Meaning	Example
Ascend (float intensity)	Ascend the drone at the given speed: (intensity × drone's max speed)	`Ascend(0.75)`
Descend (float intensity)	Descend the drone at the given speed: (intensity × drone's max speed)	`Descend(0.90)`
Forward (float intensity)	Move the drone forward at the given speed: (intensity × drone's max speed)	`Forward(0.35)`
Backward (float intensity)	Move the drone backwards at the given speed: (intensity × drone's max speed)	`Backward(0.20)`
Left (float intensity)	Move the drone to the left at the given speed: (intensity × drone's max speed)	`Left(0.75)`
Right (float intensity)	Move the drone to the right at the given speed: (intensity × drone's max speed)	`Right(0.75)`
YawLeft (float intensity)	Rotate the drone counter-clockwise along its Z-axis at the given speed: (intensity × drone's max speed)	`YawLeft(0.5)`
YawRight (float intensity)	Rotate the drone clockwise along its Z-axis at the given speed: (intensity × drone's max speed)	`YawRight(0.5)`
Wait (int time)	Wait 'time' seconds before moving onto the next command	`Wait(10)`
WaitMili (int time)	Wait 'time' milliseconds before moving onto the next command	`WaitMili(650)`

Table 3.2: List of commands found in the time-based version of Flail. Note that the maximum speed of the drone would vary from drone to drone and could vary depending on which movement the drone was performing.

the robot was and direct it as desired [38]. Even though turtle robots and drones are two different controllable mediums, their expected movement within space is similar. The main difference is that Flail functions in 3D space as opposed to 2D space. This was done by the addition of an "ascend" and "descend" command, that allows the drone to raise and lower itself, respectively, along its z-axis. Flail's initial commands followed the same ideals as LOGO's: they were intended to be simple while allowing the user to easily coordinate the drone during its flight. The commands found in the time-based version of Flail are displayed in table 3.2.

As it can be noted, a great chunk of the instructions rely on the 'intensity' parameter. This parameter is used to control the drone's speed. For example, suppose a drone model had a maximum speed of 80 km/h and it was instructed to ascend at 0.20, or 20%, of its maximum speed. The drone would then ascend at 16 km/h. This parameter was used in order to allow some control over the drone's speed during specific movements. This concept, however, does not account for drone latency or acceleration times.

3.1.1 Designing Flail

Flail is an interpreted language implemented in C, capable of outputting domain specific files with flight instructions for different drones. The translator expects a Flail file containing a trajectory described by the instructions found in Table 3.2.

At this stage, the Flail file was expected to look similar to the following:

```
Ascend(0.5);
Wait(1); # Wait for one second - will ascend for a single second
Ascend(0.0); # Stop ascent

Forward(0.5);
WaitMili(800); # Wait for 800 milliseconds - will move forward for 800 milliseconds
Forward(0.0);
```

Or in other words, instructions would be set at a given intensity (e.g., `Ascend(0.5)`), and after a certain amount of time (e.g., `Wait(1)` – one second), would be unset (e.g., `Ascend (0.0)`). Any number of instructions could be set at any given time, as instructions would only be reset, or stopped, when an intensity value of 0.0 was passed in as a parameter. As such, a program could have the forward, left, and ascend instructions set simultaneously, for example. The only limitation to this is that Flail does not allow conflicting instructions to be set at the same time, throwing a compile-time error when it is attempted (e.g., `Left(0.5); Right(0.7);`).

Below is a more complicated Flail file. Suppose that the input file rectangle.Flail was passed in, which contains the following:

```
# Make drone fly in a rectangular pattern

Ascend(0.6);
Wait(2); # Wait for two seconds - will ascend for two seconds
Ascend(0.0); # Stop ascent

Forward(0.5);
WaitMili(800); # Wait for 800 milliseconds - will move forward for 800 milliseconds
Forward(0.0);

Right(0.5);
Wait(1);
Right(0.0);

Backward(0.5);
WaitMili(800);
Backward(0.0);

Left(0.5);
Wait(1);
Left(0.0);

Descend(0.5);
Wait(2);
```

```
Descend(0.0);
```

When the translator receives the file, it first performs error and syntax checks, including:

1. Are all commands separated by semicolons?
2. Are all commands written correctly?
3. Are there any "open" commands when the program exits? (For instance, a forward command that was never reset.)
4. Are there conflicting instructions? (For instance, a left and right command that were set simultaneously.)

After passing the tests, the translator proceeds to split each instruction at the semicolon, resulting in a token of the following format: (instruction, parameter). At this stage, each parameter is expected to be an integer value, allowing it to be easily clustered into chunks of bytes (where an unsigned byte can represent up to integer value 255). Each instruction is mapped to a specific byte value as described below.

```
// Association of specific bytes to instructions

const Instructions inst = {
    .ascend = 0x1,
    .forward = 0x2,
    .backward = 0x3,
    .left = 0x4,
    .right = 0x5,
    .yawL = 0x6,
    .yawR = 0x7,
    .descend = 0x8,
    .wait = 0x9,
    .waitMili = 0xA
};
```

The output file is composed of the byte values of each (instruction, parameter) pair. The instruction is directly converted from the above mapping, while the parameter is converted into bytes. If the parameter is larger than 255, it is split up into multiple bytes until the full amount can be represented, where each byte is treated as the parameter for a smaller function call. For example, if the command `WaitMili(800)` were to be converted into bytes, the following would occur:

- `WaitMili` corresponds to byte 0x9;
- 800 would be split up into the hex values:

  0xFF, 0xFF, 0xFF, 0x23, since $255 \times 3 + 35 = 800$;
- Result: 0x9, 0xFF, 0x9, 0xFF, 0x9, 0xFF, 0x9, 0x23.

Of course, it is assumed that users would use the time function that more closely met their needs. It would be more space efficient to call the wait function for a 9 second move, than the waitMili function for 9000 ms.

3.2 Evaluation of the Time-based version of Flail

3.2.1 Establishing Communication Between a Microdrone and an Arduino

The first practical test used to examine the capabilities of Flail involved the use of a very inexpensive drone. The chosen drone was the R/C Micro Drone from Cobra RC Toys, a C$40 toy drone. The set comes with a small drone and an RC controller. This drone, not surprisingly, did not have an API to allow for its programmatic control, requiring hardware modifications to allow the controller to receive signals from the computer.

An Arduino was used as a mediator between the controller and the computer. The original controller is displayed in Figure 3.1. The Ascend/Descend/Yaw Left/Yaw Right (A/D/YL/RR) joystick, as the name implies, is responsible for ascending and descending the drone, as well as rotating it along its yaw axis. Furthermore, it is used to establish the initial connection between the drone and the controller. After turning on both the controller and the drone, the A/D/YL/RR joystick has to be moved forwards and backwards at least once to connect the drone to the controller.

A few things had to be tweaked within the controller to allow the board to receive specific signals from the computer. To do so, I opened the controller to modify its circuit board as shown in Figures 3.2 and 3.3.

The first thing to do was to disconnect the power source of the controller and to hook up two new wires in its place. This would allow the Arduino to power the controller as opposed to using external battery sources.

After performing some tests with the multimeter, it could be stated that Pin 4 of the A/D/YL/RR joystick is responsible for the drone's throttle, while pin 8 of the same joystick is responsible for its yaw rotations. Pin 4 from the F/B/L/R joystick controls the drone's forward and backwards motion, while pin 8 moves the drone left and right. These four pins had to be disconnected to allow the Arduino to send the signals in its stead. To do so,

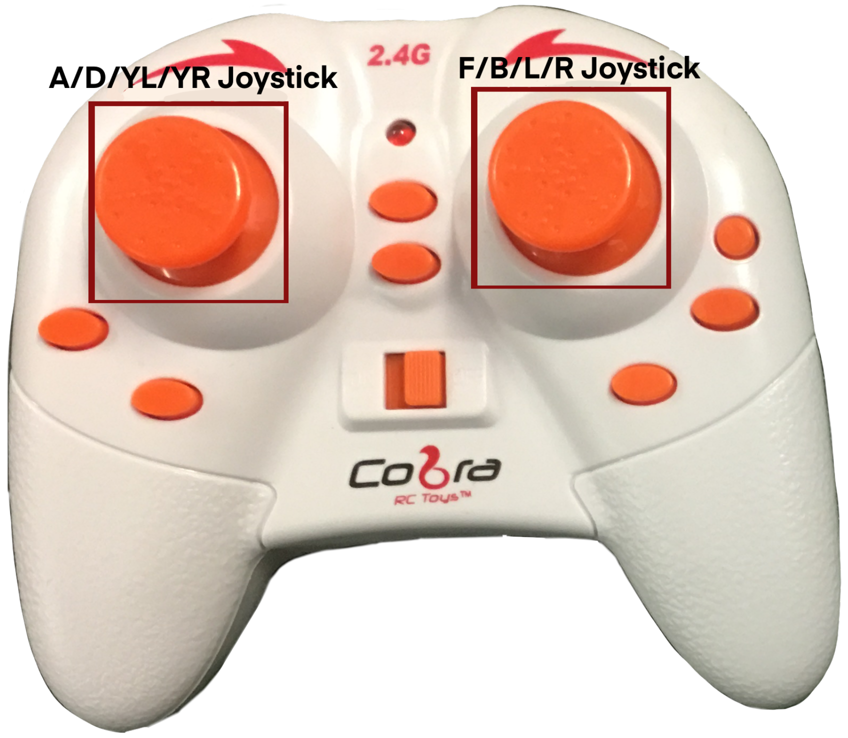


Figure 3.1: RC drone controller with its joysticks labeled. A/D/YL/RR controls the ascension, descension, and left and right yaw. F/B/L/R controls the drone's forward, backward, left, and right movement.

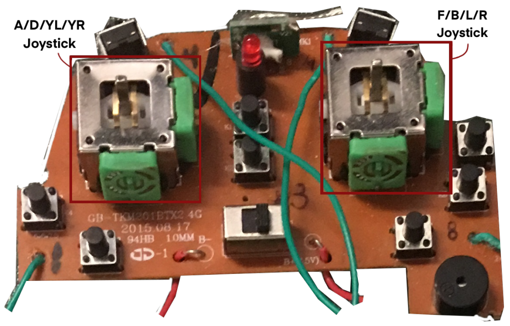


Figure 3.2: Front of the board

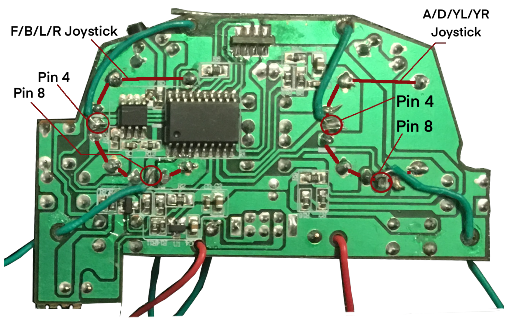


Figure 3.3: Back of the board

the two joysticks had to be unsoldered, have their respective pins peeled back, and then resoldered to their original position. The final stage required soldering four new wires to the circuit board in place of the peeled-back pins.

To send controlled voltage signals from the Arduino to the controller, low pass filters were used as shown in Figure 3.4. The low pass filters were responsible for averaging out the

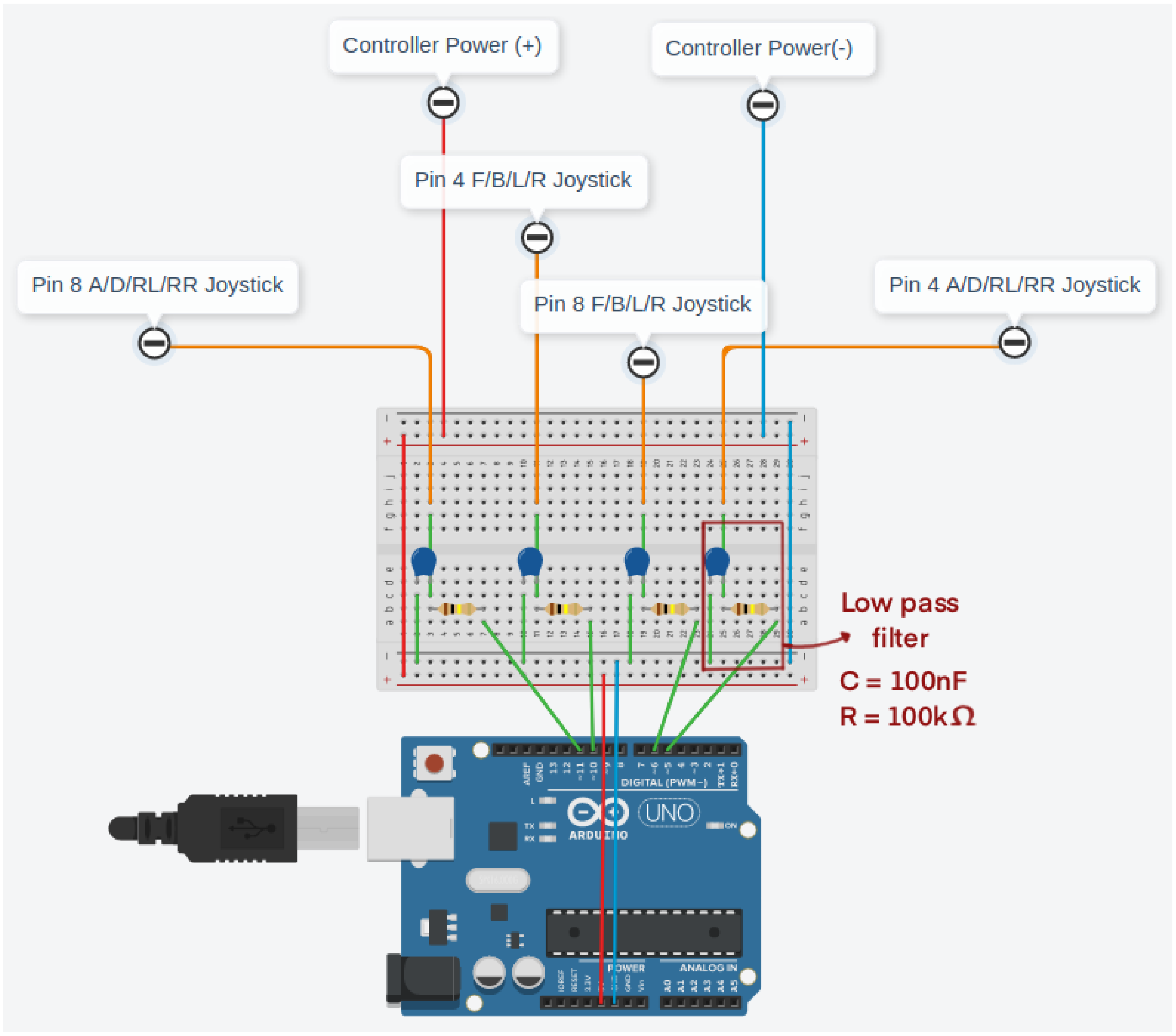


Figure 3.4: Arduino/controller circuit diagram

voltage output of the Arduino.

3.2.2 Evaluating Drone Flight Patterns

A number of flight tests were performed on the RC drone to see whether it was capable of stably following patterns. Considering how poorly they performed in the initial tests this seemed unlikely. Each flight was tracked by the VICON, a motion capture system, to facilitate examination. Four motion tracking beads were used in a square formation as shown

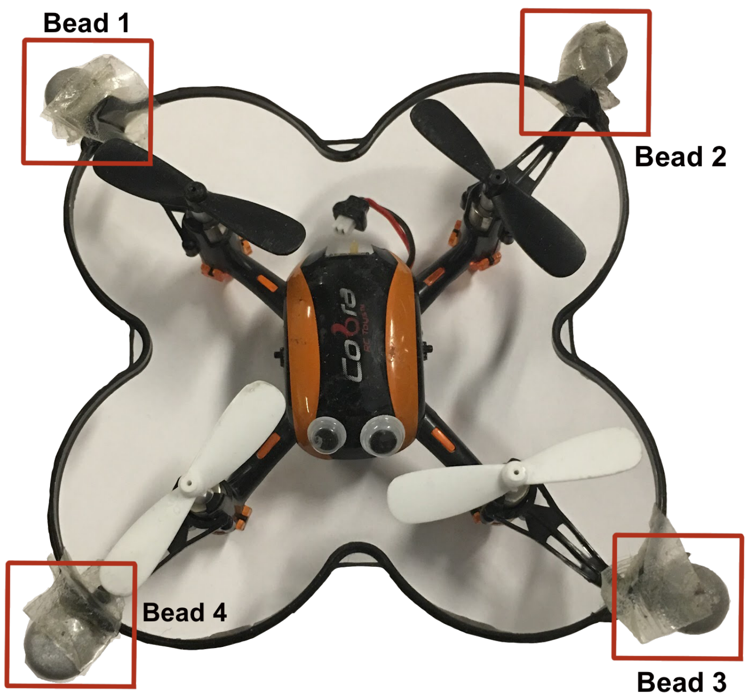


Figure 3.5: Bead formation as shown on the RC drone.

in Figure 3.5.

The symmetrical outline of the beads prevented the capture of the drone's orientation, only allowing the measurement of the drone's position during its flight. Due to the drone's lightweight and weak throttle power, no other bead formation was possible, as asymmetrical patterns tilted the drone and more than four beads prevented lift-off.

Three main tests were performed, each focusing on a different flight pattern. The first test made the drone hop three times in place; the second made the drone fly left and right; and the last, a partial cube pattern. The coordinates of the RC drone's flight, as captured by the VICON, were stored to a text file. To clean up the captured trajectories, each path was decimated[1] and interpolated.[2] These techniques will be further examined in Chapter 5.

For the three hop test, the drone was expected to remain reasonably stable while hopping three times in place. Nonetheless, the actual flight showed unexpected displacements during

[1]Decimated – coordinates that are closer than an epsilon value are removed. For these trajectories, an epsilon = 10 was chosen.

[2] Interpolated – The Catmull Rom Spline technique was used.

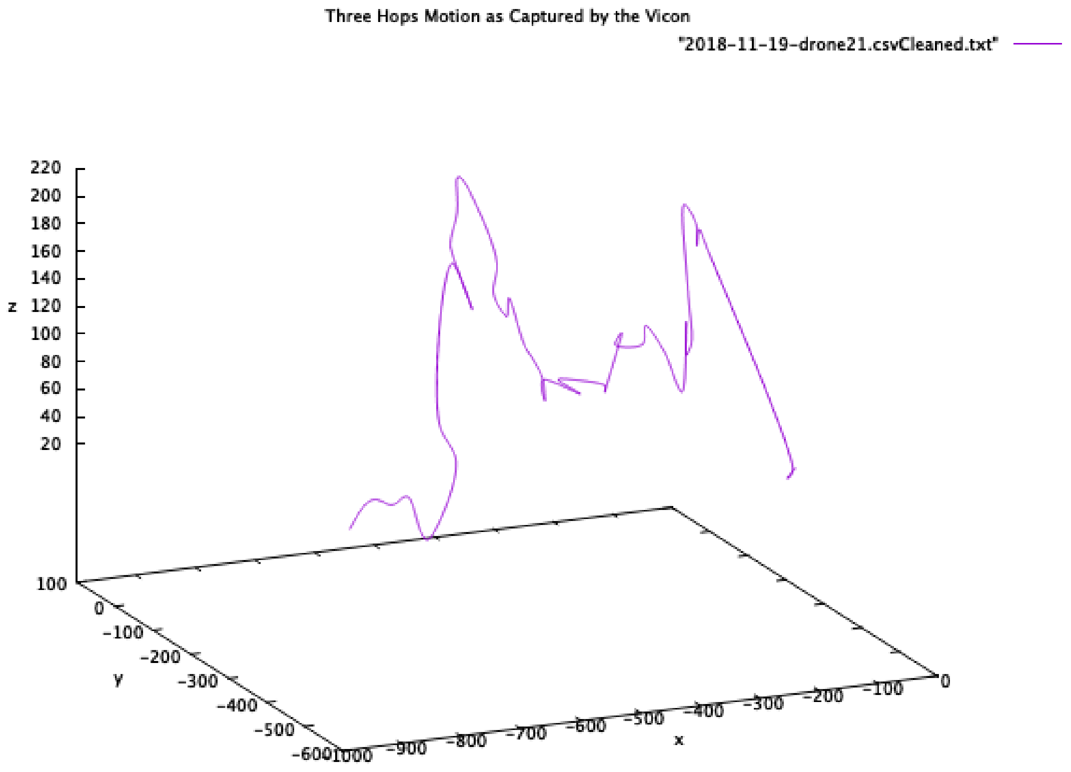


Figure 3.6: Three Hops RC Drone trajectory

its run (Figure 3.6).

The left-right test also demonstrated instability. The drone was expected to perform two motions: first: to rise up, move to the left, and lower itself down; second: to rise up again, move to the right, and lower itself to the ground. As it can be seen from Figure 3.7, substantial fluctuations occurred in the drone movement.

Finally, the partial cube test was expected to draw two parallel faces of a cube. The actual flight was shown to contain a large amount of displacement and instability as shown in Figure 3.8.

The results of these flight tests were unsatisfactory and further demonstrated the unsuitability of the RC drones for continuous flights. Each flown trajectory was noticeably unstable when compared to the expected flight paths shown in the following section.

3.3 Simulator

Since I was unable to fully test Flail with the microdrones, I created a Unity simulator to demonstrate the ideal flight patterns the drones should have followed. As stated, this version of Flail was time-based, and as such, instructions were triggered based on time.

A coroutine was used to parallelize the interpretation of the Flail file along with the physics control of the drone. The coroutine was responsible for determining which instructions were currently active/inactive, and in turn, having the Unity update function trigger/untrigger the respective physics controls.

The state of each instruction was kept track of by a set of global Boolean variables corresponding to each of the main instructions of Flail: ascend, forward, back, left, right, yawLeft, yawRight, and descend. When an instruction was set to true, the simulator interpreted that as an active instruction and triggered the corresponding drone control.

Three Flail programs were given to the simulator to visualize the expected flight paths from the three hops, left-right, and partial cube tests. The coordinates of the virtual drone were stored and graphed with gnuplot. The expected drone flight trajectories are shown in Figures 3.9, 3.10, and 3.11; the measured flights are shown beside each for comparison.

3.4 Discussion

It would appear that the toy RC drone has difficulties maintaining its stability when a number of directional changes are performed consecutively. The flight pattern that displayed the most amount of stability was the three hops test, where the drone was only expected to rise and lower itself consecutively. When the drone was told to fly in a somewhat cube-like shape, it became unstable. The drone had difficulties flying in straight lines, most likely due to its light weight and weak frame. Every crash would subsequently dent its frame and propellers, and after a while, the drone was unable to fly straight.

The selected RC drone displayed a number of problematic characteristics during its usage and was deemed not suitable for future tests. The drone is extremely lightweight and possesses a weak engine, and as such, any additional weight distorted its flight trajectory. Its battery was also fairly weak, being able to maintain flight for only about 8 minutes. Strangely enough, when the batteries were close to being drained, the drone would engage in what appeared to be a drunken stupor. It would become unable to fly in a straight line and would proceed to twirl in circles until its battery completely drained.

Overall, time-based functions, although fairly intuitive descriptors for motion, demonstrated limitations when projecting complex flight patterns. It is hard for users to compute distance based on time. Or more specifically, it is hard to estimate how *far* the drone will move based on how *long* it moved.

Furthermore, it is hard to define the scale of trajectories. There isn't really an intuitive way of using time to constrain the boundaries of a drone's flight. For example, forcing the drone to fly within a $1m^2$ area is difficult when limited to time-based functions. This limitation results in many crashes as the drone is incapable of knowing when it is about to hit an obstacle.

3.5 Summary

In this chapter, the implementation of the time-based version of Flail was looked at in detail. This chapter also included the reasoning behind the implementation of time-based functions to describe movement.

A Cobra RC microdrone was used to test out the language. These toy drones were deemed unsuitable for future flight tests due to their instability. The hardware modification that allowed an Arduino board to communicate with the RC drones was also discussed in detail.

The RC drones were shown to be wobbly during flight. Its delicate frame would quickly bend in crashes, further preventing it from flying in straight lines. It was also unable to accurately replicate changes in directions when receiving commands sent by the Arduino.

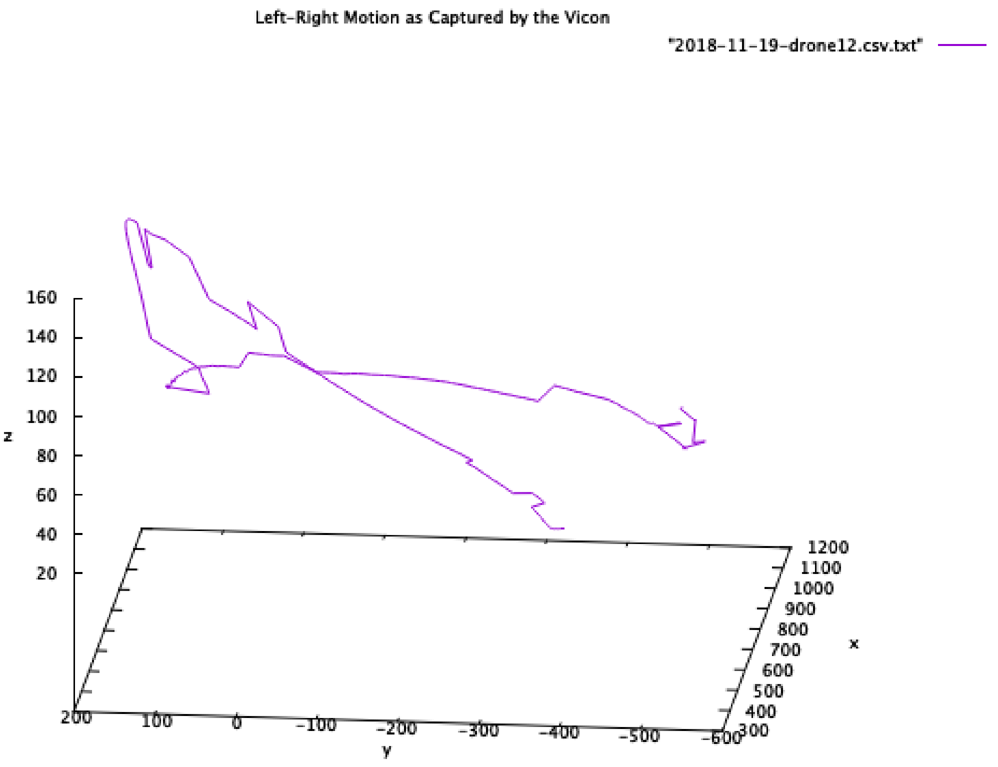


Figure 3.7: Left-Right RC Drone trajectory

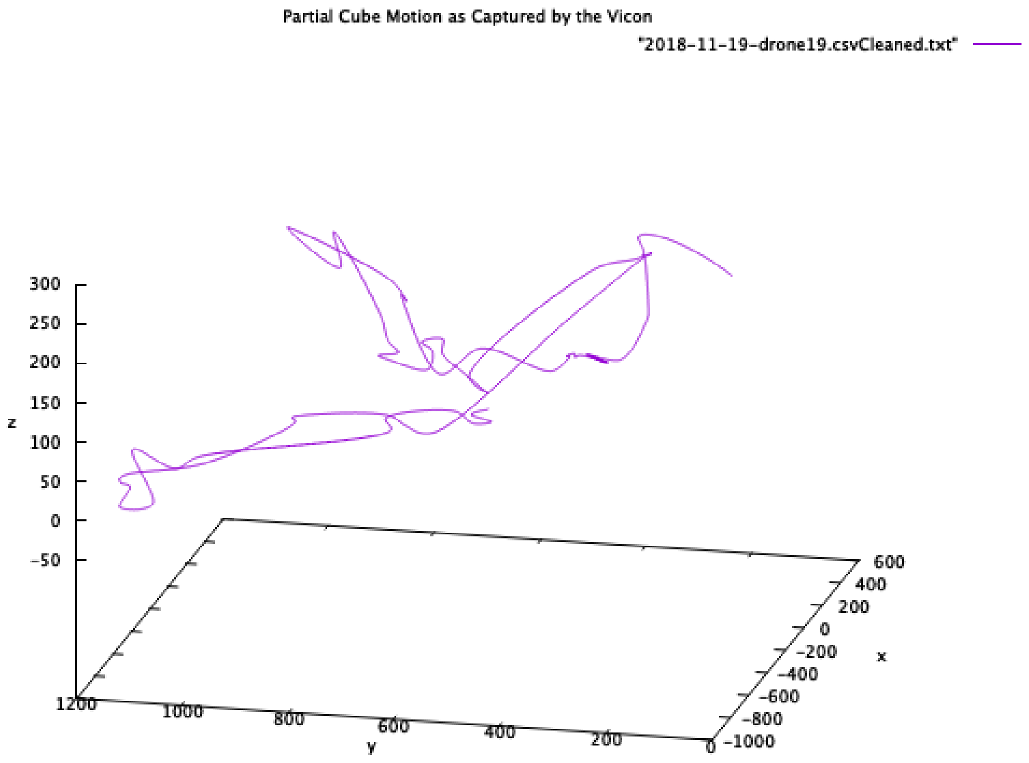
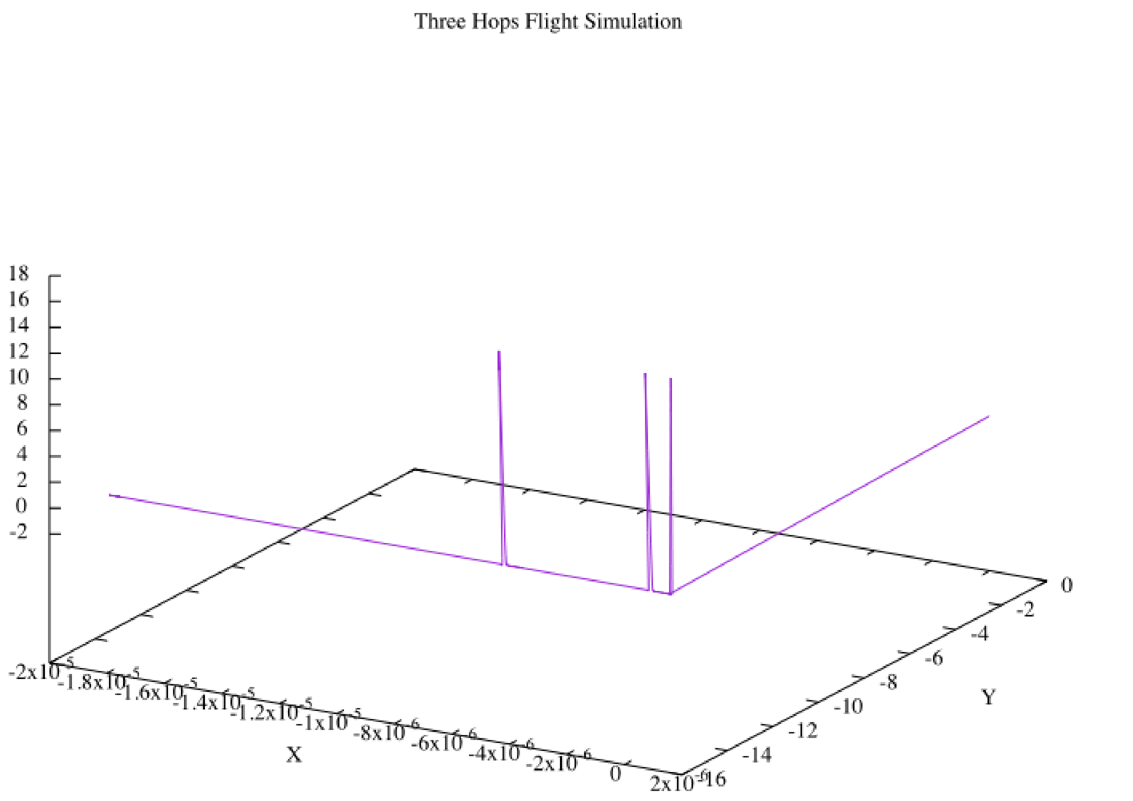


Figure 3.8: Partial Cube RC Drone trajectory



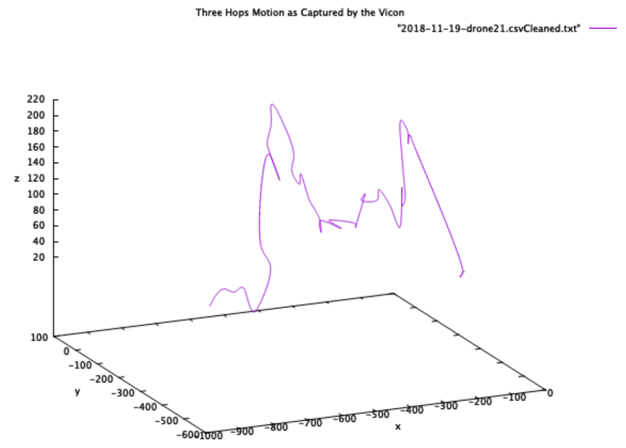


Figure 3.9: Three Hops simulated trajectory

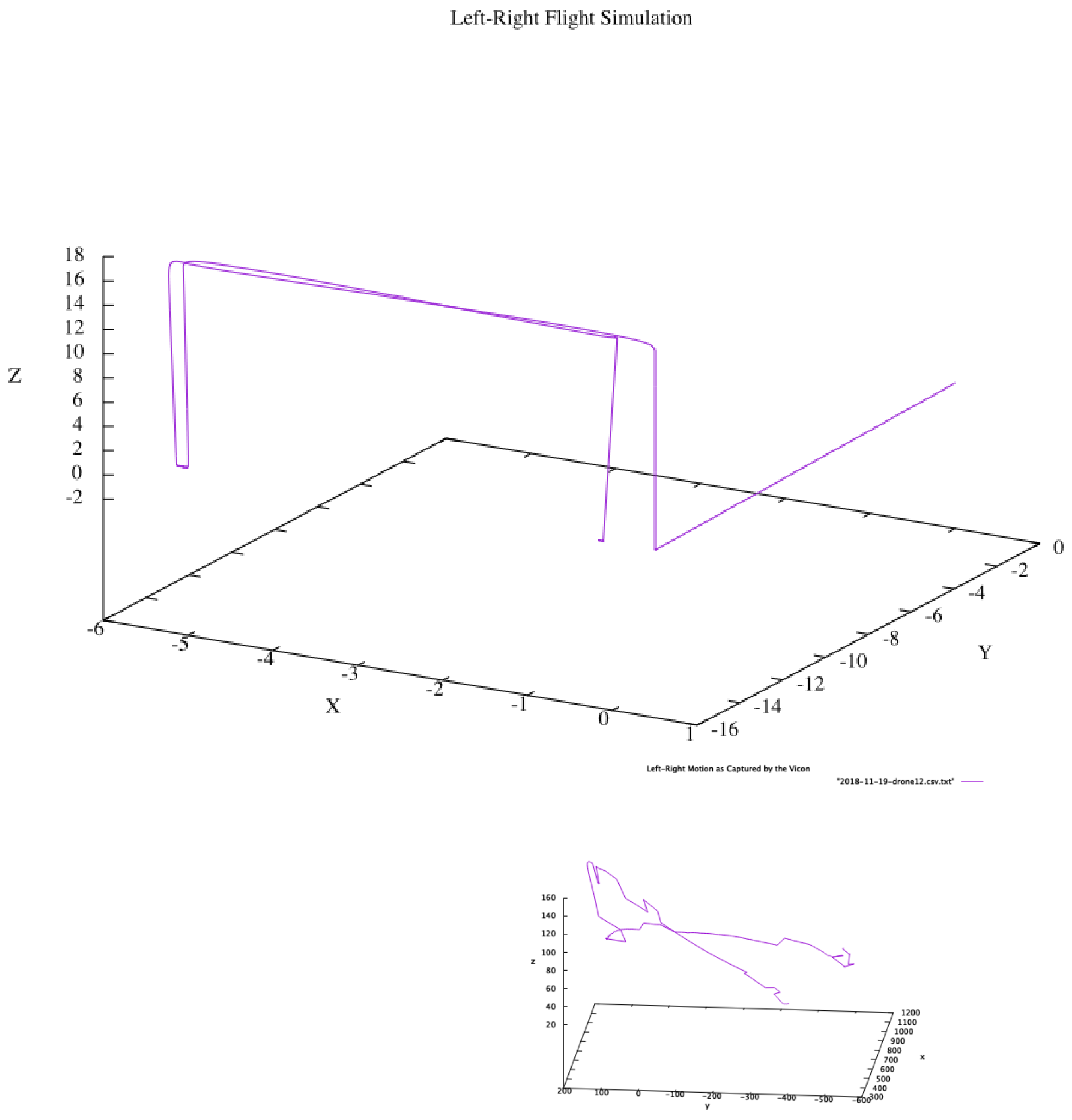


Figure 3.10: Left-Right simulated trajectory

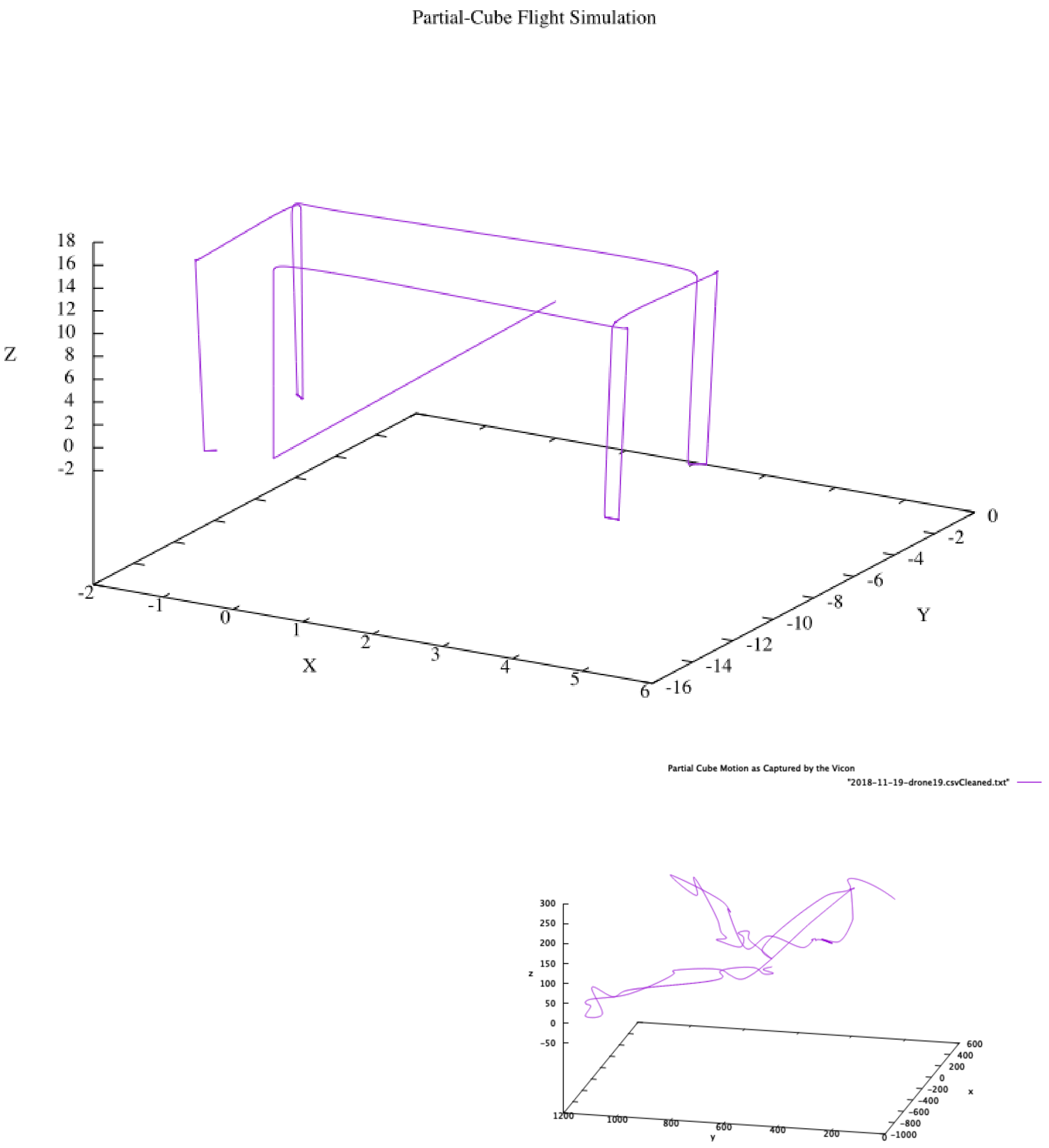
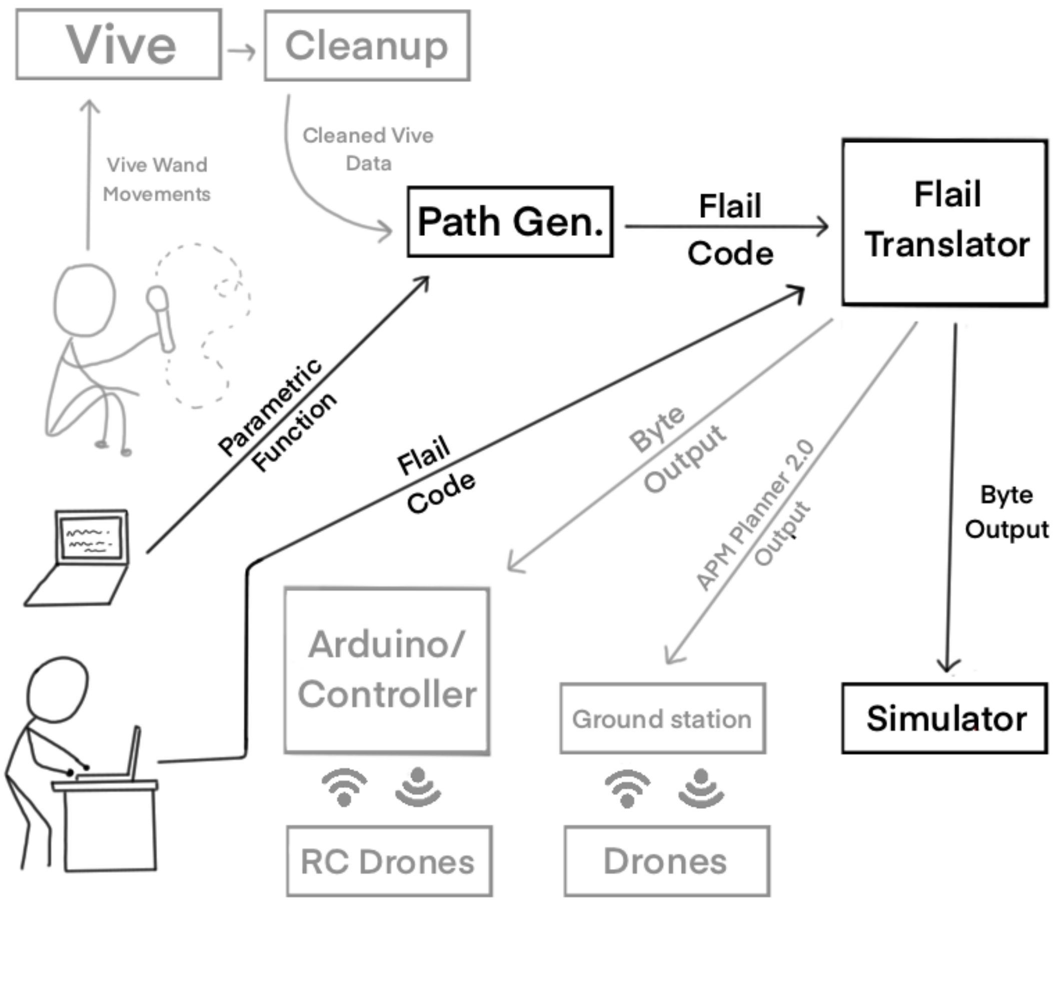


Figure 3.11: Partial Cube simulated trajectory

Chapter 4

Flail: Distance-based Motion



One issue that was presented in Chapter 3 was the difficulty to control how far a drone would fly. As such, this chapter provides an alternative to time-based movement, switching gears to focus on distance-based movement. The goal of this chapter is to answer RQ2: "How do we create a distance-based DSL capable of interacting with drones?"

As opposed to describing motion based on how long each movement takes, we could describe motion according to how far the vehicle traversed. For instance: the vehicle moved forwards for 20 cm, then it moved to the left for 15 cm, and then back for 10 cm. The distance-based version of Flail follows this idea: it is a distance-based method of describing movements in 3D space.

The first step to designing these additions to the language was to answer the following questions:

RQ2.1 How can we specify distance in Flail and what kind of distance functions would we need?

When describing motion in terms of distance, we need to first know how far the drone has to travel. Furthermore, what unit of measurement should we use? Should we describe its motion in centimeters? Meters? It's hard to say, especially since the actual path described by the user might have to follow different scales in different scenarios. The same path might be flown in a room constrained by walls and again in an open space, where the drone could move more freely. In such situations, the interpretation for how much each distance unit should be treated as should be determined by the user.

The distance parameter was implemented as a single floating point number, e.g., `Forward(10.5)`, or move forward by 10.5 units, where the unit is determined by the user. A `setMode` method was implemented that allows users to alternate between the time and distance-based version of Flail.

This chapter will focus on further developments to the Flail translator. A path generation script was developed to generate Flail code based on parametric functions in order to create more complex trajectories. Further improvements were also made to the simulator to make it accept distance-based parameters.

RQ2.2 How can we specify the scale of a trajectory path?

One consideration that has to be taken when working with distances is the scale of the trajectory. Or in other words, the dimensions of the generated path. To allow for the modification of the scale, the path generation script contains a *scale* parameter that allows the user to modify the scale of the generated paths. If a user is manually writing a Flail program, however, they would be responsible for calculating their own scaling system. For example, the simulator relies on Unity world-space coordinates, while the ground station relies on latitude/longitude coordinates. The user would have to determine which scale would better suit their needs for each of these outputs.

RQ2.3 How can we repeat movements?

Patterns are usually comprised of a repeated set of motions. The ability to repeat instructions was lacking in the time-based version of Flail. The distance-based version, introduced the ability to add basic loops through the `Repeat` instruction.

4.1 Introducing Distance Parameters to Flail

The time-based version of Flail was a good start, but it did suffer from some limitations. The main issue was its control scheme, which was time and speed dependent. These parameters aren't ideal when attempting to control how far a drone should fly at any given point in time. Another complication is that drones of different models don't have the same default speed, making it even harder to predict the travel distance. As such, constraining the flight pattern within certain boundaries, such as making the drone fly within the constraints of a room, became difficult.

To get around this issue, a new mode was implemented in the second iteration of Flail, dubbed *precision mode*. In this mode, functions are controlled by distance as opposed to time. It would be the user's responsibility to adjust the scaling system in order to control how far a unit in precision mode would correspond to in their desired output domain.

To alternate between the two different modes, a user can simply use the `SetMode(mode)` instruction. The `mode` parameter is either `intensity` for time based control, or `precision`, for distance-based control. Apart from this extra mode, the distance-based version of Flail also introduces looping. A new `Repeat` instruction was implemented. This function allows chunks of code to be iterated, or repeated, a specified number of times.

```
SetMode(precision);
Repeat 3 {
    Forward(10);
    Left(5);
}
```

This code snippet sets the translator to precision mode, meaning patterns will be defined by distance. Here, the loop will repeat the instructions `Forward(10)` and `Left(5)` three times.

Precision mode expects each function to receive specific distance values. However, there is a chance that higher precision will be required at times, especially for the rotation commands which might require floating point precision. Flail, however, was developed with the idea of treating each parameter as a byte in order to reduce the size of the output program. A single unsigned byte could represent an integer up to 255_{10}. Larger integers could then be split into chunks of 255 as necessary. Floating point values, on the other hand, require more than a single byte to represent a useful range of values (modern floating point representation will typically use 8 bytes).

To get around this limitation when using floating point numbers, binary scaling was used to represent floating point numbers in the Flail system. Binary scaling is a technique that implements pseudo floating point operations by using integer arithmetic [39]. Numbers tend to be more precise when using binary scaling crompared to floating point values with the same amount of bits. Nonetheless, floating point number representations using binary scaling are limited to a certain range, which make them susceptible to overflows and underflows. The scaling technique that was selected for Flail is known as binary angle measurement, a number based on the property that adding 180° to any angle is akin to taking its two's complement [40].

4.1.1 Binary Angle Measurements

Rotations are specified using angles in the range [0°, 360°). Unsigned Binary Angular Measurements, or UBAM, can represent angles between 0° and 360°, while BAM can represent angles between −180° and 180°.

UBAM data words are specifically designed to represent up to 360° of angular displacements in binary form, often in steps or increments of the LSB value (0.0055° for a 16 bit word), as demonstrated by the table below:

```
                        BAM bit table.
                #   0       1       2       3       4      5
bam_bit_table = [ 0.0055, 0.0109, 0.0219, 0.0439, 0.088, 0.1757,
                #   6       7       8       9      10     11
                  0.3515, 0.703,  1.406,  2.8125, 5.625, 11.250,
                #  12     13      14     15
                  22.5,  45.0,   90.0, 180.0 ]


LSB = bam_bit_table[0]
MSB = bam_bit_table[15]
```

This 16-bit word 11001000 00000000 ($b_{15} \times 2^{15}+, \ldots, + b_0 \times 2^0$) can represent a 281.25° angle:

$$281.25 = 180 + 90 + 11.25$$

$$\text{float2UBAM}(281.25) = (1 << 15) + (1 << 14) + (1 << 11) =$$

$$2^{15} + 2^{14} + 2^{11} = 51200$$

When set to one, the LSB (Least Significant Bit) is equal to 0.0055°, while the MSB (Most Significant Bit) is equal to 180°. The MSB value represents half the maximum value that may be transmitted.

When all 16 bits are set:

- an angle greater than 359.9939° is indicated;
- their sum is 359.9939°, and corresponds to the maximum quantity that can be transmitted.

When all bits in the UBAM data word are clear (with zeros), a 0° angle is represented.

UBAM words are also used to transmit non-angular values, such as range, length or height. When non-angular values are being used, the LSB value contains the smallest step, or increment of the quantity being transmitted [41].

```
## Word size for BAM.
WSIZE = len(bam_bit_table)-1


## Convert a float number to an
#  Unsigned Binary Angular Measurement (UBAM).
#  In the C language, integer overflow behavior is
#  different regarding the integer signedness.
#
#   Two situations arise: (Basics of Integer Overflow)
#   - signed integer overflow : undefined behavior
#   - unsigned integer overflow : safely wraps around
#                                 (UINT_MAX + 1 gives 0)
#
#  @param num a float number.
#  @return an Unsigned short integer [0 to 65535]
#  corresponding to an angle [0 to 360).
#
def float2UBAM (num):
    #num %= 360
    res = numpy.uint16(0)
    for i in range(WSIZE, -1, -1):
        if num >= bam_bit_table[i]:
           num -= bam_bit_table[i]
           res += numpy.uint16(1<<i)
    return res


## Wraps an angle to +- 180
wrapPi = lambda x: x - 360 * ((x + 180) // 360)


## Convert a float number to a signed Binary Angular Measurement (BAM).
#
#  @param _num a float number.
#  @return a Short Integer [-32768 to 32767]
#  corresponding to an angle [-180 to 180).
#
def float2BAM (_num):
    _num = wrapPi(_num)
    # bring the sign back.
    res = numpy.int16(float2UBAM(abs(_num)))
    return res if _num > 0 else -res
```

Converting back a BAM number b to a float is very simple:

$$\begin{aligned} sum(v \times ((b >> i)\ \&\ 1) \text{ for i,v in enumerate(bam\_bit\_table)}) &= \\ (180\ b_{15} +\ 90\ b_{14} + 45\ b_{13} + 22.5\ b_{12} + \cdots + 0.0055\ b_0) &= \\ 180\ (b_{15} + b_{14}\ 2^{-1} + b_{13}\ 2^{-2} + b_{12}\ 2^{-3} + \cdots + b0\ 2^{-15}) &= \\ 180\ \times 2^{-15}\ (b_{15}\ 2^{15} + b_{14}\ 2^{14} + b_{13}\ 2^{13} + b_{12}\ 2^{12} + \cdots + b_0) &= \\ 180 \times\ 2^{-15} \times b & \end{aligned} \tag{4.1}$$

Therefore, it is just a question of multiplying the BAM number b by: $180 \times 2^{-15} = MSB \times 2^{-15} = LSB$.

4.2 Constraining Flail Output Sizes

One of the issues presented in the last couple of chapters was the size of the output file. Flail breaks down large integer values into multiple bytes, treating each chunk as a smaller instruction call. This rapidly increases the size of the output file.

This issue is especially visible when binary scaling is used, since a small floating point number can be converted into a rather large integer. For example, 281.25 would be converted into the integer value 51200. Flail would interpret this value as chunks of 201 bytes (51200 = 200×255 + 200). Moreover, Flail associates each parameter chunk with its respective instruction byte, expanding the 201 bytes into 402 bytes. This approach quickly increases the size of the output.

To prevent incredibly large files from being generated, an extra instruction was implemented, called `repeatNextInstructionFor`. This instruction is responsible for repeating the instruction that comes after it n times. This instruction is not accessible to users and is automatically used by the translator when an instruction is to be broken into various byte chunks.

The updated byte association list is now:

```
const Instructions inst = {
        .ascend = 0x1,
        .forward = 0x2,
        .backward = 0x3,
        .left = 0x4,
        .right = 0x5,
        .yawL = 0x6,
        .yawR = 0x7,
        .descend = 0x8,
        .wait = 0x9,
        .waitMili = 0xA,
        .setMode = 0xB,
        .repeatNextInstructionFor = 0xC
};
```

Imagine that the following movement call was made: `Forward(51200)`. By using the `repeatNextInstructionFor` instruction, this call will be interpreted as:

$$0xc, 0x200, 0x8, 0x255, 0x8, 0x200$$

Or, repeat the next instruction (`Forward(255)`) 200 times and then execute `Forward(200)`.

The use of this instruction effectively cuts 402 bytes down to 6 bytes. This procedure limits the maximum size an instruction can take to 6 bytes.

4.3 Path Generation

I wrote a Python script to generate a set of precomputed curves (or trajectories) to test the simulator. This script uses polar equations to create the trajectories, and generates a Flail output file that contains the corresponding instructions needed to pilot the drone to follow the path. The output program executes only `Forward` and `Yaw` instructions. Therefore, one could imagine controlling a plane or a drone as if they were flying it with a remote controller.

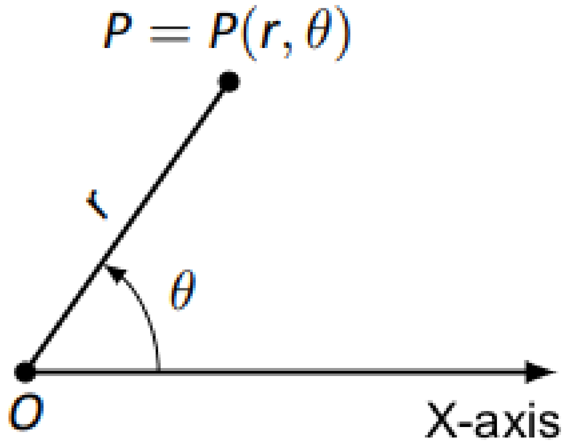


Figure 4.1: Polar coordinate depiction for generic point P

As a brief recap, the polar coordinate system uses an (r, θ) pair to represent points, where `r` is the distance form point `P` to the origin `O`, and θ is the angle formed between the positive X direction and segment $\overrightarrow{OP}$, as shown in Figure 4.1 [42]. Polar coordinates were ideal to use since the path generation script only calculates the amount that the drone has to rotate and move forward by to reach the next point in the sequence.

To draw polar equations by means of a vector system, such as Python's turtle graphics, we should keep track of three points: $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, and $P_2 = (x_2, y_2)$. Vector $\overrightarrow{P_0P_1}$ is the previously calculated movement segment and vector $\overrightarrow{P_1P_2}$ is the current segment.

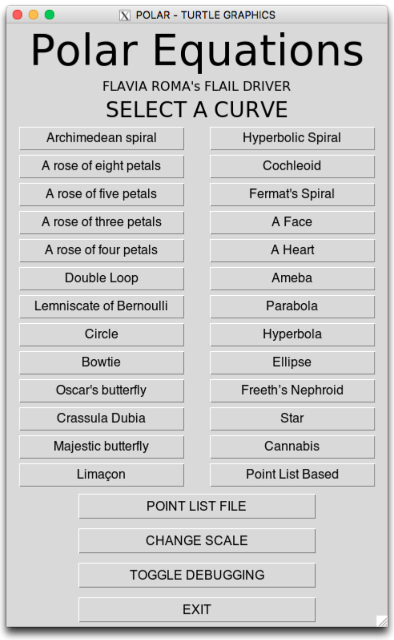


Figure 4.2: Flail's Interface

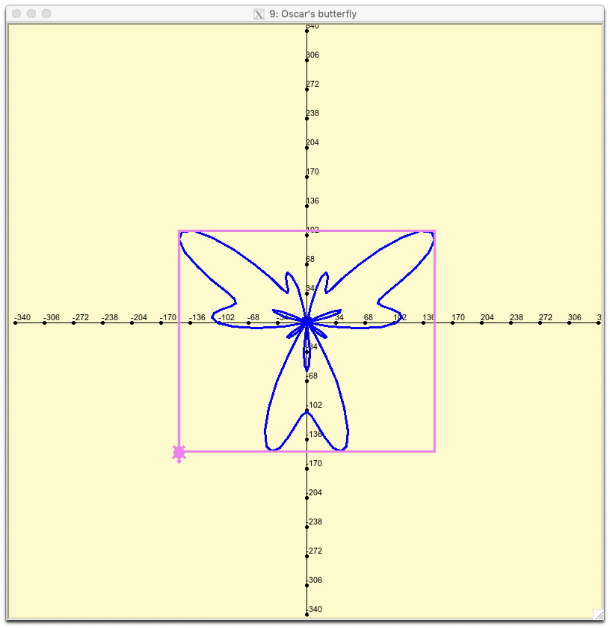


Figure 4.3: Oscar's Butterfly, as drawn by the path generation script

For the first point, we need the angle $\alpha \in [0, 2\pi)$ between the x-axis and the vector $\overrightarrow{P_0P_1} : \alpha = atan2(y_0 - y_1, x_0 - x_1) + \pi$

- $atan2(y, x) \rightarrow \alpha \in$ (-π, π], returns a value between $-\pi$ and π by default and,
- $atan2(-y, -x) + \pi \rightarrow \alpha \in [0, 2\pi)$, returns a value between 0 and 2π (since we want to use the UBAM procedure, as described in Section 4.1).

First, we calculate the angle γ between these two segments, by using the dot product:

$$c = \cos(\gamma) = \frac{\overrightarrow{P_0P_1}}{|P_0P_1|} \cdot \frac{\overrightarrow{P_1P_2}}{|P_1P_2|} = \frac{(x_1 - x_0)(x_2 - x_1) + (y_1 - y_0)(y_2 - y_1)}{((x_1 - x_0)^2 + (y_1 - y_0)^2)((x_2 - x_1)^2 + (y_2 - y_1)^2)^{1/2}} \tag{4.2}$$
$$\gamma = \text{acos}(\text{min}(\text{max}(\text{c,-1}),\ 1))$$

The turtle makes a left or right turn, based on the sign of the cross product of the two segments:

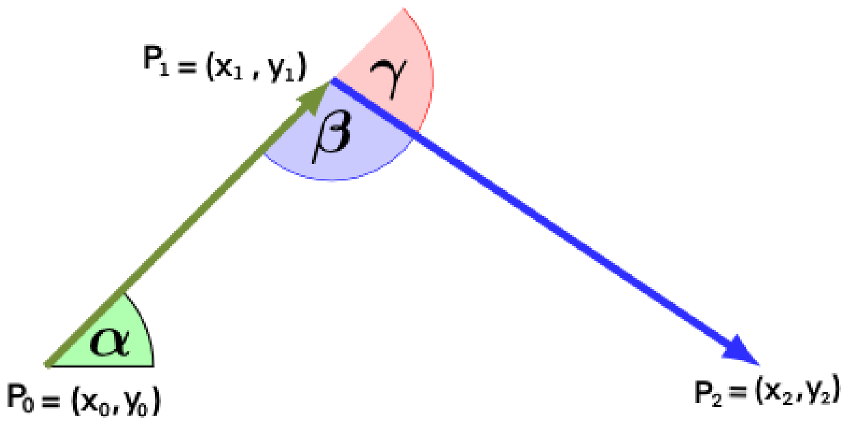


Figure 4.4: An iteration of the vector movement calculation, for the previous vector $\overrightarrow{P_0P_1}$ and current vector $\overrightarrow{P_1P_2}$.

$$\begin{aligned}
&\overrightarrow{P_0P_1} \times \overrightarrow{P_1P_2} = \\
&(x_1 - x_0, y_1 - y_0) \times (x_2 - x_1, y_2 - y_1) = \\
&(x_1 - x_0)(y_2 - y_1) - (y_1 - y_0)(x_2 - x_1), \\
&< 0 \rightarrow \text{right turn} \\
&> 0 \rightarrow \text{left turn}
\end{aligned}$$

Then, the turtle moves forward by a distance equal to the length of the current segment:

$$dist(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.3}$$

All the equations are multiplied by a scale factor, so we can control the coordinate values. As a consequence, we adjust the world coordinates in turtle graphics, to make sure the drawn curves maintain the same size. Angles are represented by UBAM, and distances by BAM [43]. As mentioned, there is a chance for underflows/overflows. Distances can overflow if a length is outside the range [-180, 180), or underflow, if the scale is too small (less than 3.0), since 16 bits is not enough for such a small representation. The default scale is 80.0 for a square window of 680 pixels. Therefore, the world coordinate length and height are set to ($scale \times 8.5$).

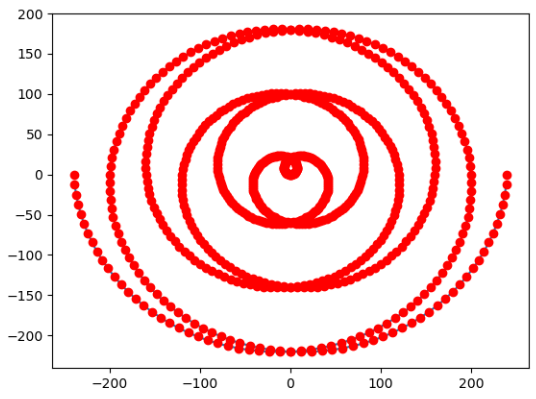


Figure 4.5: Original 2D Archimedean spiral curve

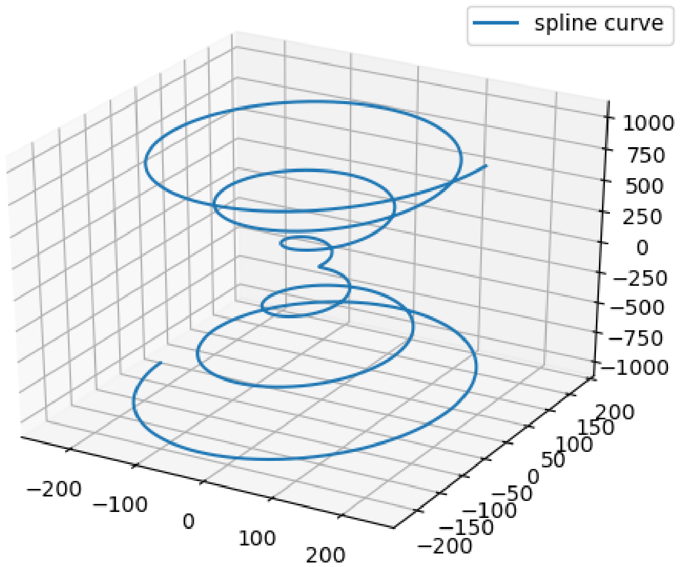


Figure 4.6: Modified Archimedean spiral to be a 3D curve

Even though this path generation script generates 2D paths, it is easy to modify the generated 2D curve to become a 3D path. Consider the double spiral path shown in Figure

4.5, which can be turned into a 3D curve by simply adding a z-coordinate. In this case, the calculated θ angle was used as the z-coordinate. Therefore, it is possible to create 3D paths as well.

4.4 Incorporation of Precision Mode to the Simulator

As demonstrated in Chapter 3, there are a number of issues associated with the selected RC microdrones. Due to their instability, it was decided that they would not be used again for evaluation purposes.

As such, for this chapter, the evaluation of the distance-based version of Flail will be limited to the simulator presented in Chapter 3. The Unity simulator had to undergo a few modifications to accept the newly added precision mode. The simulator described in Chapter 3 only considered time based functions, and as such, only required a single coroutine to set/unset currently active instructions. In precision mode, however, the drone model is controlled by distance, meaning that it can only move onto the next instruction if the given target distance has been reached.

As Unity's update function is executed every frame and was responsible for moving the virtual drone, I had to make sure that the program only moved on to the next instruction when the drone reached the distance set by the current instruction. A queue was used to precompute the Flail file and each (instruction,parameter) was stored in an object named DroneAction.

```
DroneAction {
        // The instruction attributed to this action
        private int instruction;

        // Drone velocity for this given action
        private int intensity;
        // The distance that the drone has to traverse
        private float distance;

        // Has the drone reached the target distance?
        private bool distanceReached;
};
```

The coroutine InterpretActionList is responsible for dequeuing one element at a time from the DroneAction queue and performing the corresponding instruction. The routine will wait until the designated distance has been traversed to unset the instruction and move on to the next DroneAction object.

The instructions found in Flail will either translate or rotate the drone. In Unity, object positions are described by vectors and rotations by quaternions. To check to see whether a translation has been completed, I compared the distance between the drone's current position and its target destination by using the method Vector3.Distance, which returns the distance between two vectors. If the target destination has been reached, the distanceReached flag in the current DroneAction will be set, and the next instruction will be read.

Rotations, on the other hand, are deemed to be completed when the model finishes rotating about a given angle. To check the current rotation status, the Quaternion.Angle method was used, which returns the angle between two quaternions. If the distance between the drone's current rotation and the target rotation is less than the empirically decided value, 0.5, the distanceReached flag is set.

Further modifications included a modified playground for the drone, as depicted in Figure 4.7. Terrain and environment objects were added to give the user reference points for the drone's position in space.

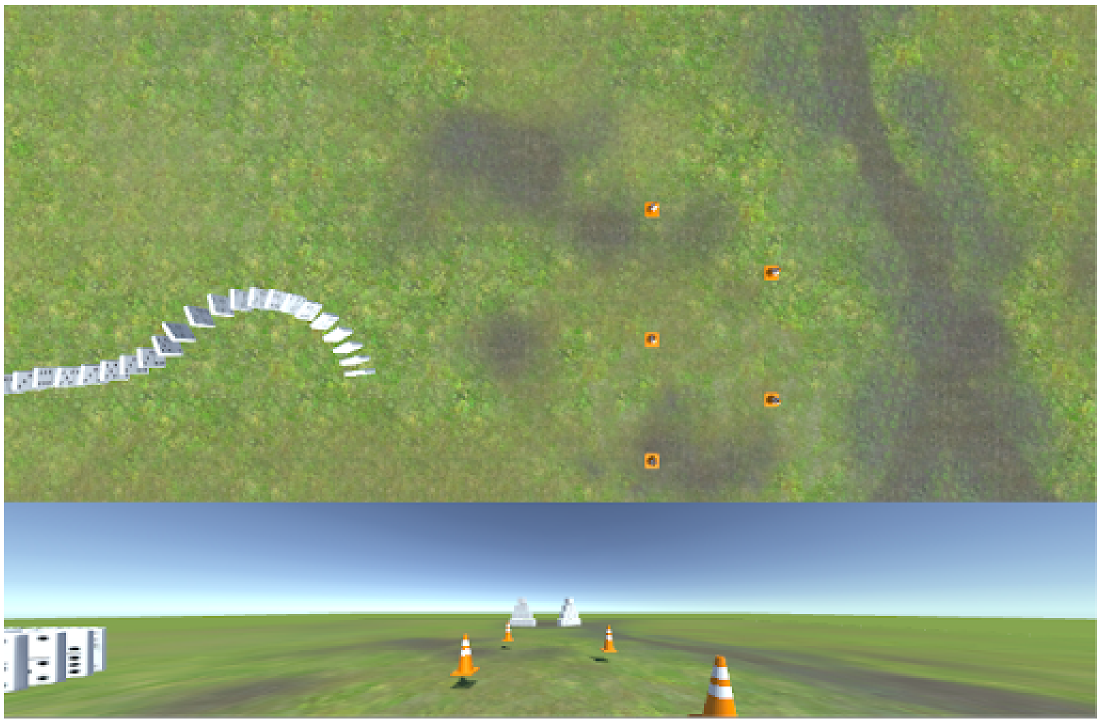


Figure 4.7: Additions to the simulator's play area. The simulator's view was separated into top down and side view to aid the user in examining the drone's flight path.

4.5 Evaluation

To evaluate Flail's precision mode, the script `Polar.py`, presented in Section 4.3, was used. `Polar.py` contains a number of parametric equations that can be used by Flail to describe increasingly complex flight patterns. These equations vary from a straightforward circle to more complex designs such as the Cochleoid shown in Figure 4.8.

The script is able to output a Flail file that describes the drone motion necessary to follow the selected pattern. The output can then be fed into the Unity simulator to examine how a drone would ideally interpret such a path.

The double loop pattern as shown in Figure 4.9 was chosen as a flight test path. The virtual drone was shown to fly the path accurately as demonstrated in Figure 4.10. I considered this test a success.

Nonetheless, precision mode is currently a bit of a hassle for users to use, as it either

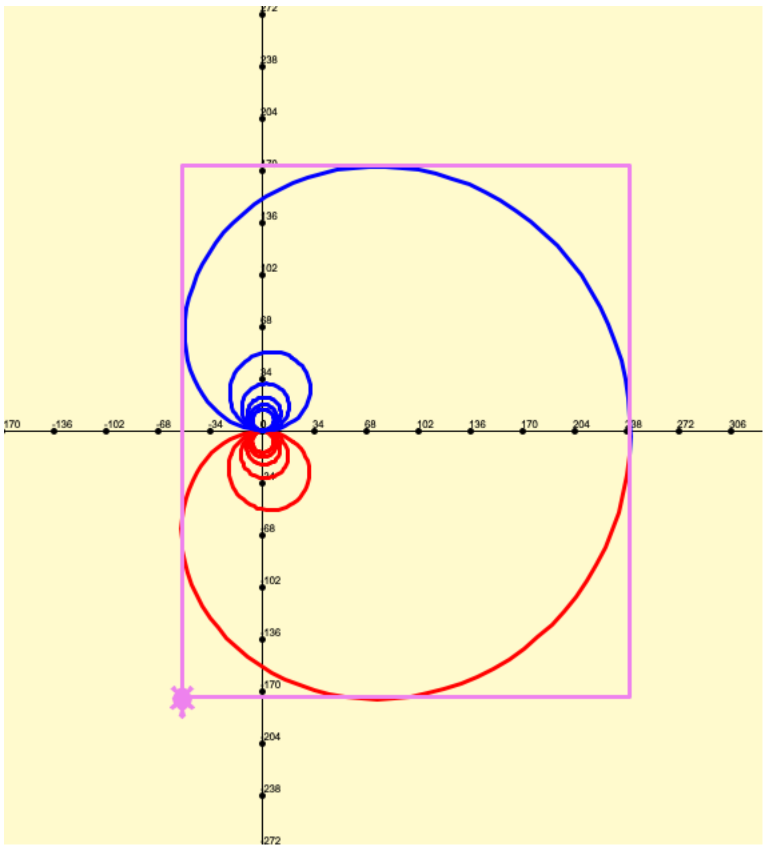


Figure 4.8: Cochleoid curve as generated by `Polar.py`. The red section of the graph represents angles in the negative range, while the blue section represents positive angles. The box surrounding the graph is the generated bounding box which is used to demonstrate the scale of the graph.

requires the manual computation of distances for their desired trajectories, or to mathematically compute equations to describe paths. The first approach limits the complexity of the pattern, as manually calculating distances for large patterns can become taxing for users. The second approach requires the parametric equation of the desired curve, which can be hard to figure out for complex curves.

4.6 Summary

This chapter covered Flail's precision mode, a distance-based method to describe trajectories. This mode usually requires more precision than an integer can provide, oftentimes requiring

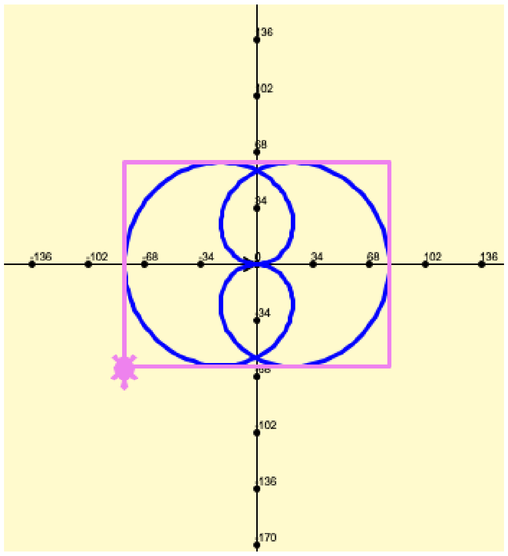


Figure 4.9: Double Loop as shown in `Polar.py`.

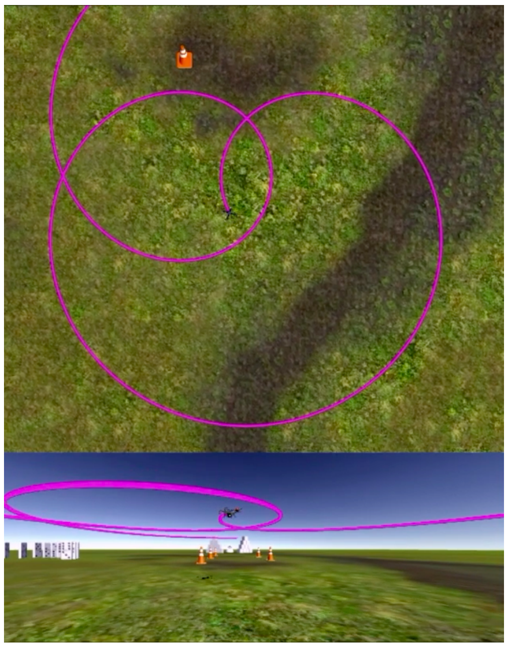


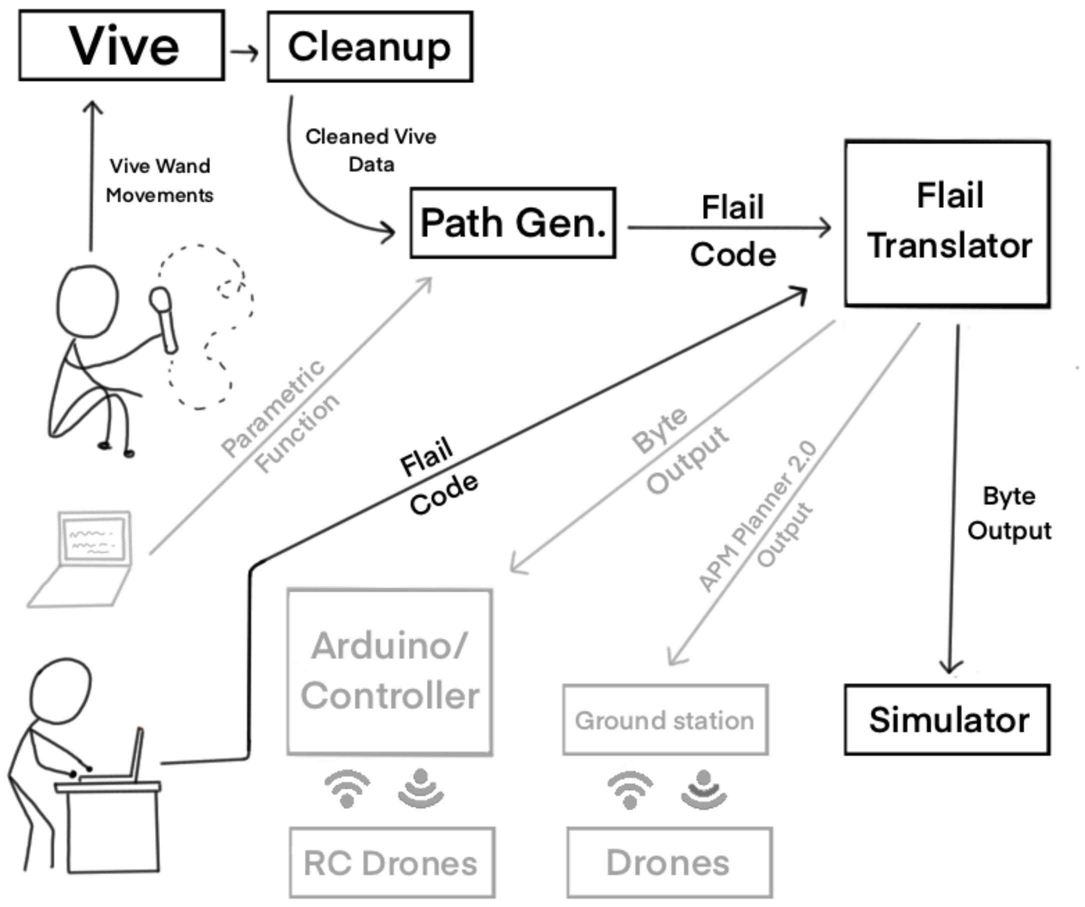
Figure 4.10: Double Loop as flown by the virtual drone

floating point numbers. However, Flail's output is composed of bytes preventing the direct use of floating point numbers. Binary scaling was used to get around such limitations, allowing floating point numbers to be represented as integers.

The distance-based interpretation of movement resulted in a more intuitive approach to design flight patterns, as trajectories could now be modeled as mathematical functions. Nonetheless, this approach is still not intuitive enough to allow users to easily create their own unique trajectories, as more complex patterns still require the step by step computation of distances.

Chapter 5

Towards the Interactive Generation of Flail Code



One of the difficulties of using Flail is that complex trajectories will require more extensive code. Imagine that a user wanted to calculate a trajectory similar to the intricate star pattern shown in Figure 5.1.

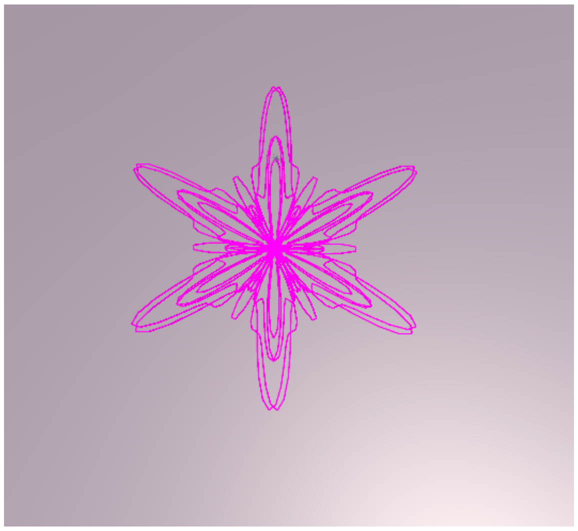


Figure 5.1: Star pattern as generated by `Polar.py`

The trajectory for such a pattern is composed of a large amount of short rotations and forward movements, which would be tedious to calculate by hand.

As such, this chapter will focus on answering RQ3: How can we use gestures to simplify the generation of Flail code? Users may be unable or unwilling to write more extensive code to describe their desired paths, making the use of gestures a more intuitive approach for path description. The idea is to have users use a Vive wand to specify the basis for a trajectory.

Furthermore, this chapter will use the HTC Vive to generate trajectories. The path generation script was updated to generate Flail code based on HTC Vive data. The resultant Flail program will be visualized with the Unity simulator.

5.1 The Vive

As shown in Section 4.3, one way to describe trajectories is to use a parametric function as a basis for the path. Curves of unknown parametrization, however, are difficult to recreate which limits the use of the script.

This chapter will demonstrate the use of the HTC Vive, a virtual reality system, to aid users in drawing paths in 3D space. The Vive can detect motion limited to its “play area”, which has a diagonal area of up to 5m (16 ft 4 in). For a room-scale setup, a minimum play area of 2m x 1.5m (6ft 6in x 5ft) is required [44].

The Vive comes with a controller that allows users to wirelessly interact with the virtual world provided by the Vive. These controllers are trackable objects and can be used to keep track of their position within the play area.

The idea is to allow users to gesture what path the drone should follow by moving the controller around. The Vive will keep track of all points drawn by the user, and this list will then be fed into the path generation script (presented in Section 4.3) to generate Flail motions.

5.1.1 Acquiring Vive Data

The HTC Vive was used to capture curves drawn in 3D space and the Vive wand was used as the gesture medium. The Vive kept track of the wand’s x, y, z coordinates within the play area and Unity stored the coordinates to a text file.

There were three main setups for the trajectories as shown in the Figures 5.2, 5.3, and 5.4. For each setup, paths were generated by placing the Vive wand on the lines and tracing them out.

5.1.2 Cleaning and Smoothing the Data

The data points acquired from the HTC Vive were shown to possess a large amount of jitteriness, as shown by Figures 5.5 and 5.6.

Even though the setups shown in the previous subsection were traced with the wand, it was hard to maintain perfectly straight lines and to prevent tremors while moving. Furthermore, accidental bumps would occur around the chairs leading to extra shakiness in the

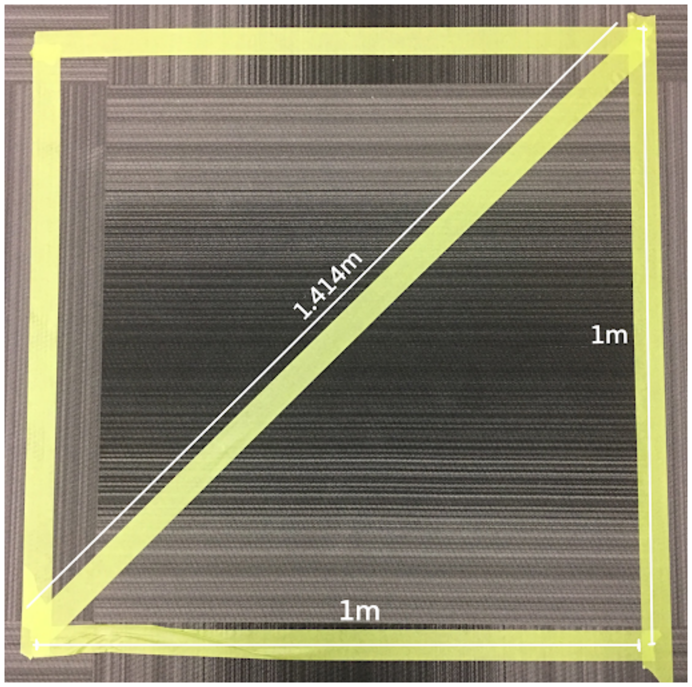


Figure 5.2: Setup 1: A 1m x 1m square was drawn. This setup was used to draw two shapes, a square and a right triangle.

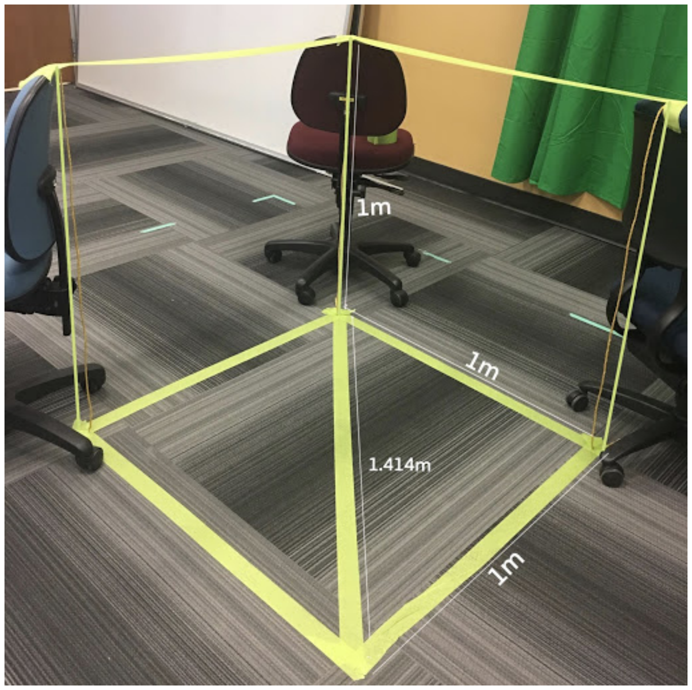


Figure 5.3: Setup 2: A 1m x 1m x 1m cube was drawn. This setup was used to draw out a partial cube with two faces.

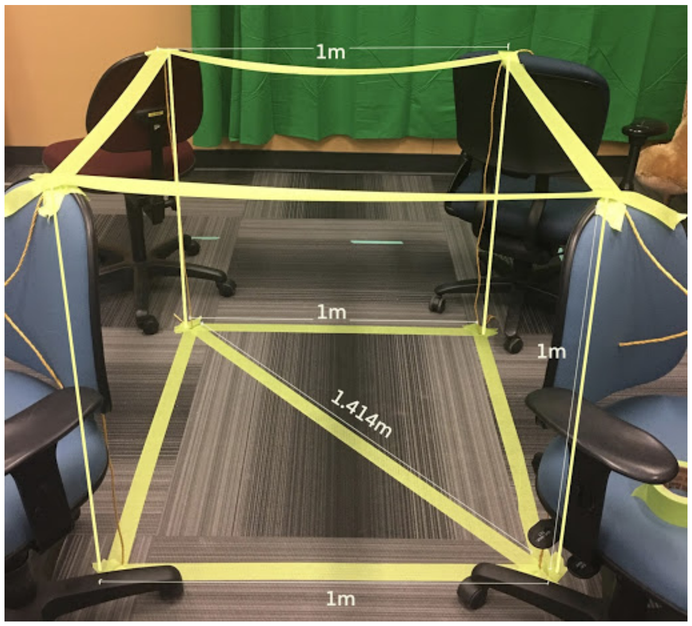


Figure 5.4: Setup 3: A full cube was set up. This setup was used to draw a partial cube with two parallel faces.

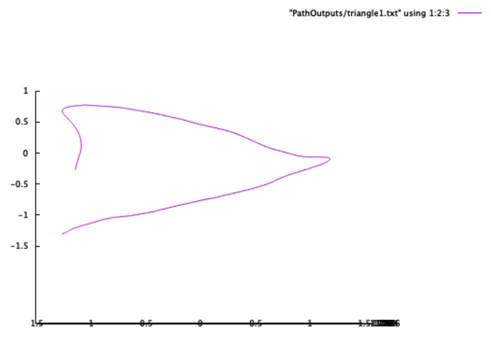


Figure 5.5: A seemingly smooth curve

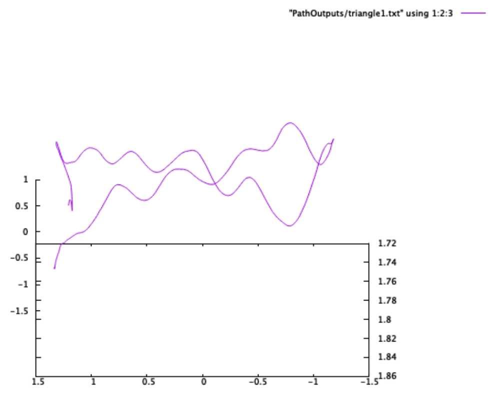


Figure 5.6: Rotating the path displays the jitteriness in the curve

curves.

There were two main issues that had to be dealt with before generating a Flail program for the Vive paths:

1. The closeness of the acquired points. It is very inefficient to calculate displacements that are so small. Apart from leading to a very large Flail file, it is inefficient to make a drone fly in bursts as short as 0.0001 units.

2. The jitteriness of the paths. I did not want to generate a program that would make drones fly shakily.

In regards to the first issue: Unity is capable of querying objects 60 times per second [45], resulting in points that are too close together. This is problematic as the path generation script requires a minimum distance of about 0.0055 units between vectors. Anything smaller than this results in an underflow, or in an output of 0. As such, points that were too close together had to be removed.

The second issue required different smoothing methods depending on whether the given pattern was composed out of any of the following: 3D shapes, straight lines, circular patterns, and 2D shapes. I used three different smoothing techniques to address the aforementioned trajectory types: projection, the Ramer–Douglas–Peucker algorithm, and the Catmull-Rom spline.

Projection involves projecting all curve points onto a given plane. For the setup that I used, I only projected points onto the xz, xy, or yz plane. This is easily performed by “clearing” the respective axis. To project onto the:

- XZ plane, multiply all points by [1, 0, 1], which clears the Y coordinate.
- XY plane, multiple all points by [1, 1, 0], which clears the Z coordinate.
- YZ plane, multiply all points by [0, 1, 1], which clears the X coordinate.

I noticed that the 2D patterns that I traced tended to have fluctuations along the Y axis. Take the square trace from setup 1 as shown in Figure 5.7. There is a noticeable amount of noise along the y-axis. A simple way of removing these fluctuations is to project all points onto the xz plane, effectively “flattening” the shape, as shown in Figure 5.8.

The Ramer–Douglas–Peucker (RDP) algorithm was used to decimate the curves, or to remove points that were too close together. This algorithm, which is also known as the iterative end-point fit algorithm, is a line simplification algorithm that finds a similar curve

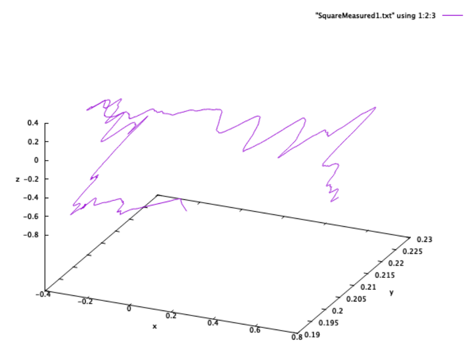


Figure 5.7: Original trace captured from setup 1

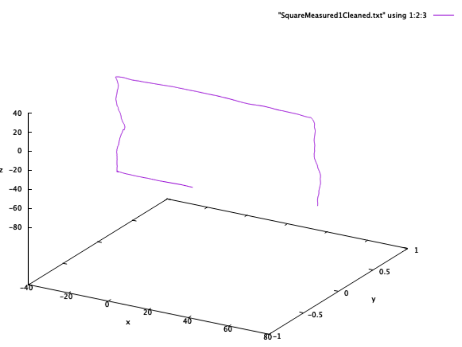


Figure 5.8: All curve points projected onto the xz plane

to the original, but with fewer points. It does so by imagining a line between the first and last point of the curve, and removing any point in between that is closer than a given distance ϵ [46]. A new curve is generated based off of this new list of points.

In order to select a cohesive ϵ, the smallest possible displacement, one characteristic from the BAM table (as shown in Section 4.1.1) has to be maintained. The minimum value to prevent underflows, as shown in the table, is 0.0055, meaning that displacements have to be at least 0.0055 units apart to prevent underflows. The aforementioned staircase method further breaks down the displacements, d, into smaller chunks by dividing it by a value, n. As such, to prevent underflows for this method, ϵ has to be calculated so that its smaller chunks will be at least 0.0055:

$$\epsilon/n >= (0.0055 \times n)$$

One of the partial cubes I traced from setup 2, shown in Figure 5.9, is cleaned up with ϵ = 0.1 (Figure 5.10). The RDP algorithm was effective at stabilizing the cube pattern as it reduced the number of points in the original graph from 2502 down to 13.[1]

A problem arises, however, when dealing with patterns that are curved as opposed to

[1]The reduced number of points in the file would also affect the size of the Flail output file. I did not report the change in file size due to the amount of underflows that would occur in the original HTC Vive trajectory.

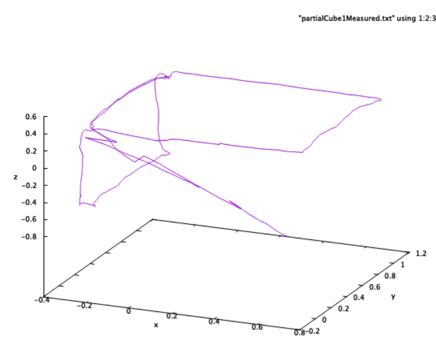


Figure 5.9: Original trace captured from setup 2

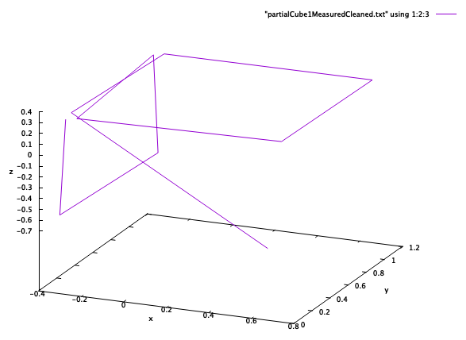


Figure 5.10: Trace after the application of the Ramer–Douglas–Peucker algorithm with $\epsilon = 0.1$

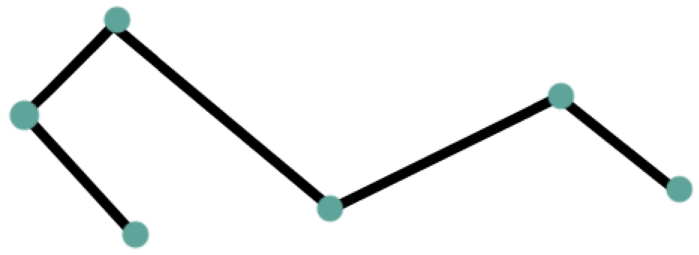


Figure 5.11: Original segmented points

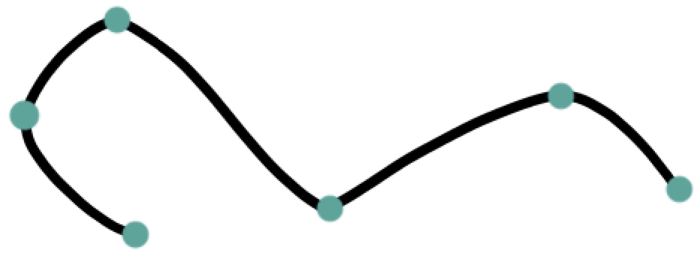


Figure 5.12: Smoothed points after the application of Catmull-Rom

straight. The RDP algorithm, albeit useful in decimating paths, segments patterns that are curvaceous. Such patterns require further smoothing to regain its curves. This is where the Catmull-Rom spline comes in.

Imagine that we have a set of four points connected by straight lines, as shown in Figure 5.11, and we wanted to modify it to create a smoother path similar to Figure 5.12.

The Catmull-Rom spline is a technique that is often used in geometric design as it can easily interpolate control points without solving system of equations [47]. It is a type of interpolating spline (a curve that intersects its control points) defined by a set of four control points: P_0, P_1, P_2, and P_3, with the curve being drawn from P_1 to P_2 [48]. This method approximates its points with a smooth third degree polynomial function that is piecewise defined [49]. Two outer points are required to smooth out a desired portion, or in other words, P_{i-1} and P_{i+2} are needed to calculate the spline between points P_i and P_{i+1}.

5.2 Adding Vive Data Functionality to the Path Generation Script

The path generation script (Section 4.3) allowed 2D curves to be drawn from parametric functions. I modified the script to accept 3D paths and to generate 3D Flail instructions. The script went through two main modifications: an added input parameter that accepts a list of points as the basis for the path, and the acceptance of z coordinates.

The first change was fairly straightforward. When parametric functions are being used, the script continuously queries the functions to fetch the next calculated point in the equation. In order to accept a list as the basis for the path generation, I added a new function that iterates through the list and returns the next available point.

The second change required the calculation of how far a drone would have to ascend or descend to reach the next point in the sequence. Imagine that we are currently at point A and are trying to reach point B. There are two main methods that can be used to calculate the displacement needed to reach point B:

Method 1 (Forward-Ascend): The drone could move forward until it is underneath point B, and then rise up to meet the point. Method 2 (Staircase): The drone could move forward and upwards in short increments, similar to a staircase, until the point is reached.

Both methods are illustrated by Figure 5.13.

As explained in Section 4.3, the script keeps track of three points: $P_0 = (x_0, y_0), P_1 =$

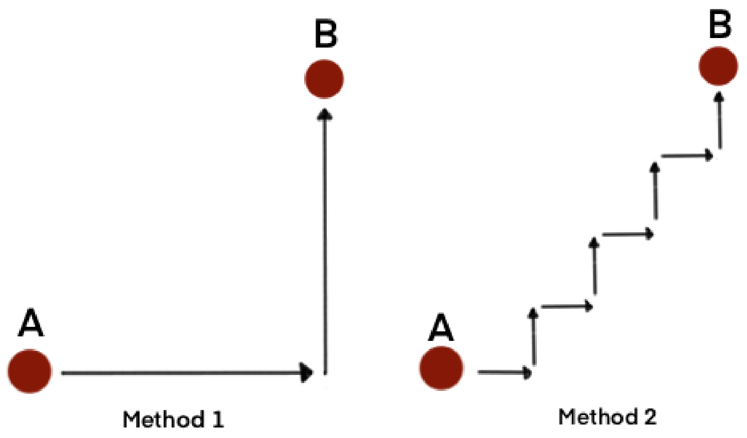


Figure 5.13: Method 1: Move forward and ascend. Method 2: staircase method.

(x_1, y_1), and $P_2 = (x_2, y_2)$, where vector $\overrightarrow{P_0P_1}$ is the previously calculated movement segment and vector $\overrightarrow{P_1P_2}$ is the current segment. To incorporate the 3rd dimension, the points have to include the z coordinate: $(x_0, y_0, z_0), (x_1, y_1, z_1)$, and (x_2, y_2, z_2).

The amount that the drone has to move forwards by is calculated by taking the distance between two vectors with the distance formula:

$$dist(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

This value will be referred to as forward_displacement. The amount that the drone has to ascend by, is calculated by taking the difference between the z coordinates of two vectors. This value will be referred to as z_displacement.

1. Method 1: To reach point B: Move forward by the calculated forwards displacement. Ascend for the calculated z displacement.

2. Method 2: To recreate the "stair" effect shown in Figure 5.13, the forward and z displacements have to be broken down into chunks. The size of these chunks may vary according to how precise, or how close to a diagonal line the user might want the flight to take. To break down the displacements, a value n has to be chosen and used to

divide up the displacements.

Suppose the drone has to fly forwards by 10 units and upwards by 3 units. If the chosen n is 5, then the script will output a loop composed of the following instructions:

```
Repeat 5 {
        Forward(2);
        Ascend(0.6);
}
```

As such, the Flail output will use the following format to reach point B:

```
Repeat n {
        Forward(forward_displacement / n);
        Ascend(z_displacement / n);
}
```

5.3 Evaluation

5.3.1 Evaluating Different Methods of Ascension

Two methods to ascend the drone were presented, dubbed the forward-ascend method and the staircase method. Both present distinct deviations from the expected flight path. The root mean square deviation (RMSD) was used to calculate the amount of error that each method presented. The RMSD is commonly used to measure the difference between the values predicted by a model and the values observed [50]. RMSD is always a positive value, and a result of zero would imply a perfect data fit. Generally, a lower RMSD is better than a high one. RMSD can be calculated with the following formula, where T is the number of data points:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{T}(y_{predicted} - y_{actual})^2}{T}}$$

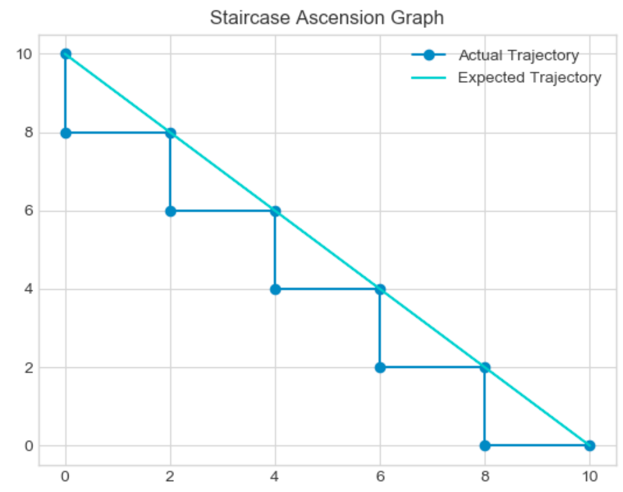


Figure 5.14: Staircase method of ascension for n = 5

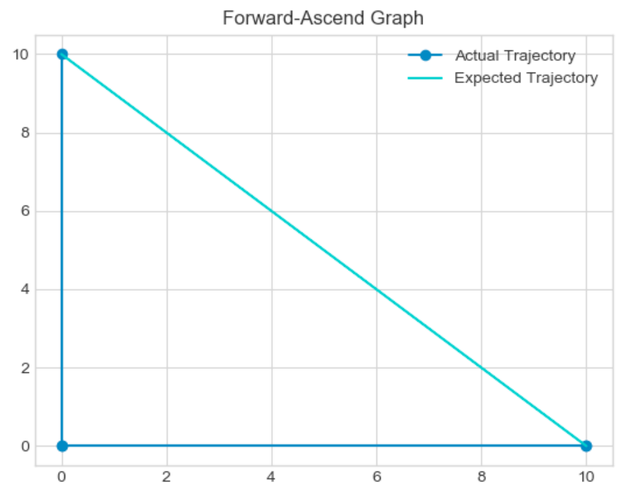


Figure 5.15: Forward-Ascend method

Let's take the following path as an example. Suppose a drone is flying from position (10, 0) to (10, 10). The drone can either ascend by flying forwards for 2 units and then ascending for 2 units in incremental bursts, or by flying 10 units forwards and ascending for 10 units, as shown in Figures 5.14 and 5.15.

The RMSD for both methods can be calculated by taking the difference in y values between the predicted and actual values:

$$RMSD_{staircase} = \sqrt{\frac{\sum_{i=1}^{5} 2^2}{5}} = 2$$

$$RMSD_{Forward-Ascend} = \sqrt{\frac{\sum_{i=1}^{1} 10^2}{1}} = 10$$

As the staircase method always bears smaller displacements compared to the forward-ascend method, it can be stated that the following will always be true:

$$RMSD_{staircase} \leq RMSD_{Forward-Ascend}$$

A question is posed, however. Is there a range of values where either method could be used interchangeably? Assume that we were to fly from point A to point B, where both points were separated by a width, w, and height, h. Is there a range of w and h values, where either the staircase method and the forward-ascend method could be used and generate the same RMSD?

To check, I plotted three distinct graphs that depict the RMSD for points separated by varying widths and heights (Figures 5.16, 5.17, 5.18). Given a maximum value (max), a surface plot is generated that includes the RMSD of every (width,height) coordinate pair between points (0, 0) and (max, max). Different values for the staircase iteration, n, were chosen.

As each staircase increment is consistent, meaning each individual "stair" has the same width and height, the RMSD formula can be simplified to:

$$\sqrt{\frac{\sum_{i=1}^{T}(y_{predicted} - y_{actual})^2}{T}}$$

$$= \sqrt{\frac{n \times (y_{predicted} - y_{actual})^2}{T}}$$

$$= \sqrt{(y_{predicted} - y_{actual})^2}$$

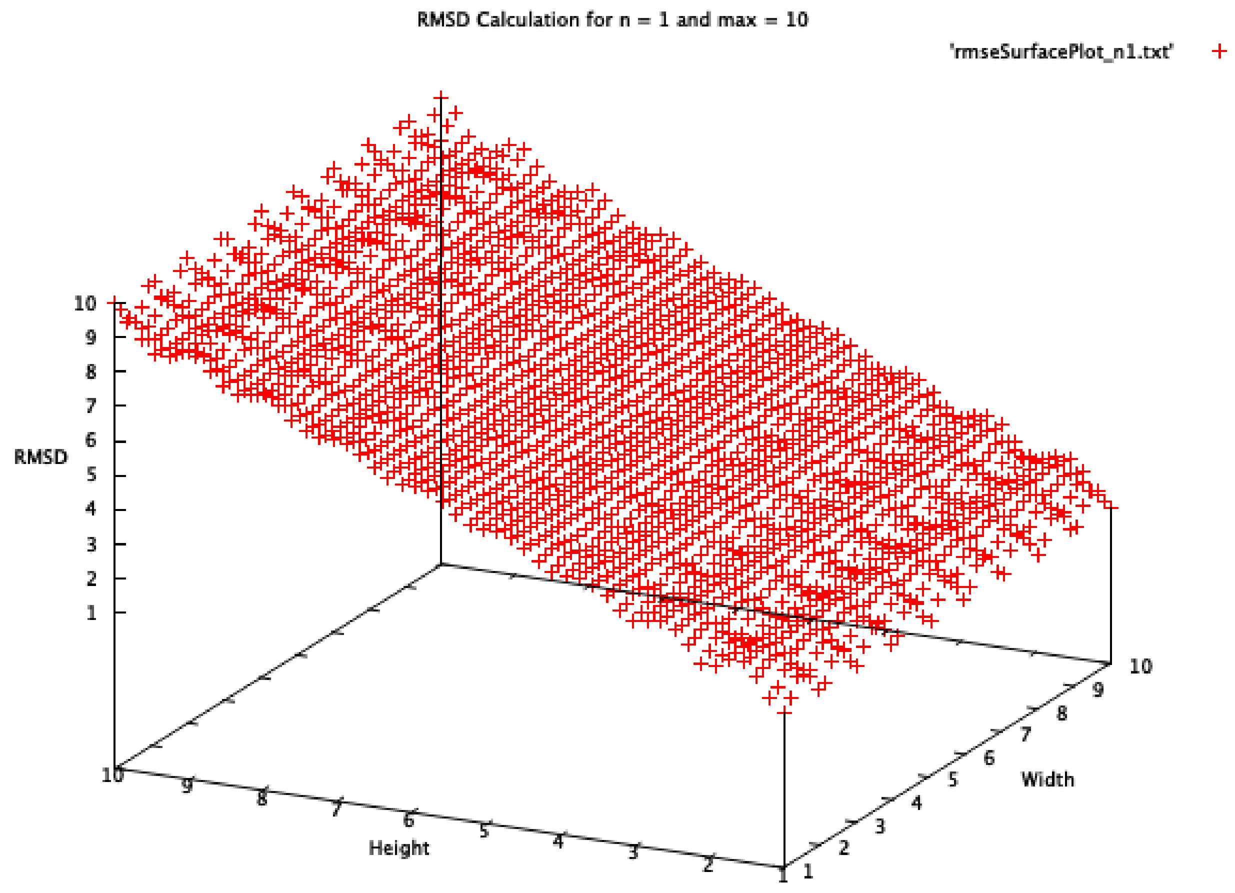


Figure 5.16: RMSD plot for n = 1. This corresponds to the forward-ascend method.

$$= (y_{predicted} - y_{actual})$$

Where $(y_{predicted} - y_{actual})$ is the height of the individual "stairs".

The height and width of each "stair" is calculated by dividing the original height and width of the ascension points by 'n'. As the above graphs vary from 0 to 10, the height of each stair will vary from 0, 1/n, 2/n, . . . , 10/n. As such, it makes sense to end up with a series of linearly increasing points.

The fourth graph (figure 5.19) displays all three previous graphs together. There isn't any noticeable range where both methods could be used interchangeably. However, it can be stated that there are height values for which the RMSD is the same. This happens in the

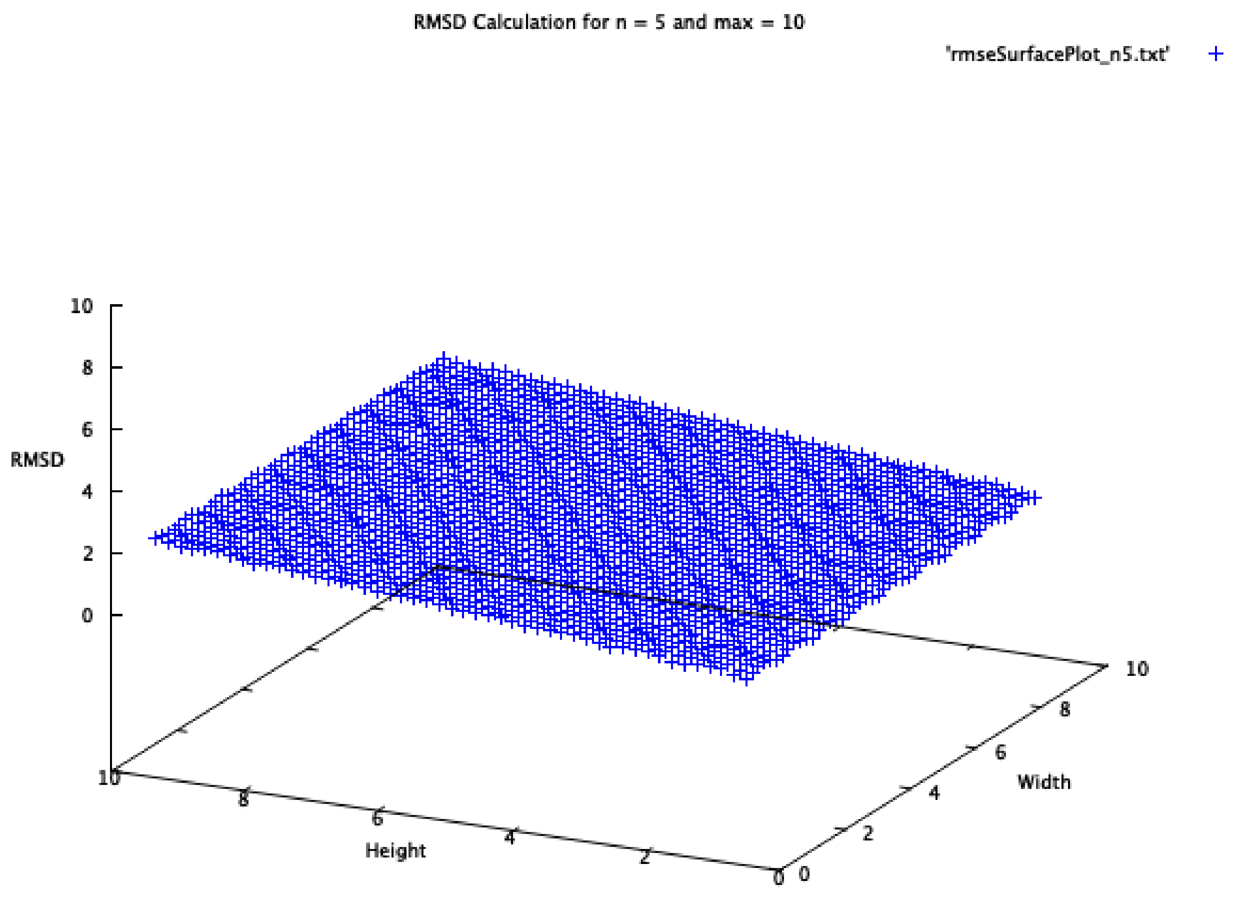


Figure 5.17: RMSD plot for n = 5

above scenario when the RMSD = 1.0. Which occurs when: n = 1, for height = 1; n = 5, for height = 5, n = 10, for height = 10.

Conclusively, the methods are indistinguishable from one another when they both have the same “stair” height.

5.4 Comparison of Curve Data

To evaluate the Flail programs generated from the Vive paths, we will examine the original path taken from the Vive, the cleaned data points, and the simulated flight path.

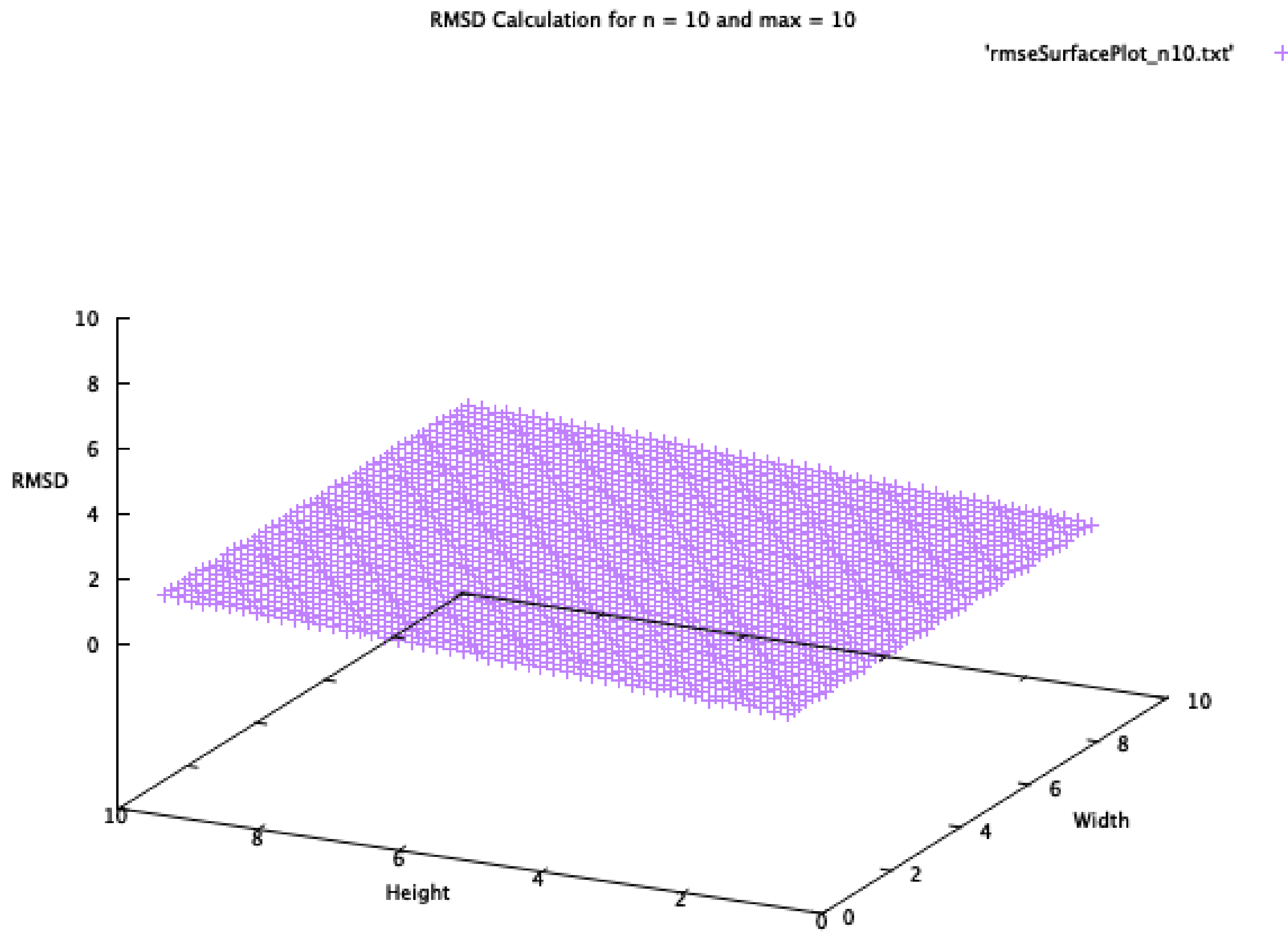


Figure 5.18: RMSD plot for n = 10

Path 1: Right Triangle Taken From Setup 1

The original path (Figure 5.20) was shown to be unstable and to not fully resemble a right triangle. To clean up these points, the path was projected onto the XZ plane and decimated with a chosen $\epsilon = 0.1$ (Figure 5.21). These two techniques were shown to be effective, as the cleaned up data points are noticeably closer to a triangle. As shown in Figure 5.22, the simulated flight was capable of following the expected path.

Path 2: Partial Cube 1 Taken From Setup 2

The original Vive path was expected to be composed out of straight lines, but was shown

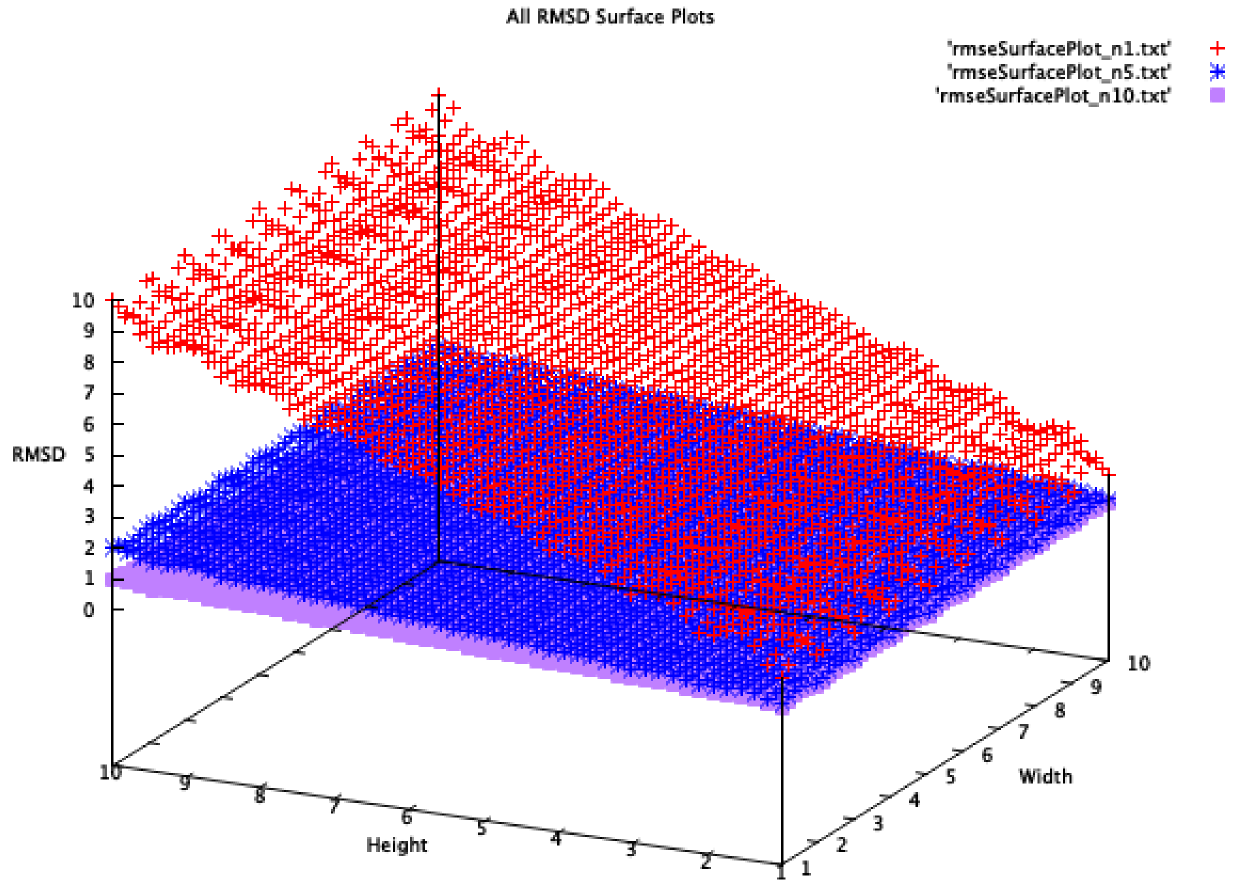


Figure 5.19: Combination of all 3 surface plots

to be more curved with some outlier points (Figure 5.23). A similar procedure was used to clean the path where RDP was applied with $\epsilon = 0.1$ (Figure 5.24). The resultant path was shown to be straighter and appeared to more closely resemble a cube (Figure 5.25).

Path 3: Partial Cube 2 Taken From Setup 3

Although the overall shape of the partial cube was understandable, the generated shapes were still jittery (Figure 5.26). The cleaned up points were decimated with an $\epsilon = 0.1$. It is noticeable that the bottom chunks of the path were removed after the Ramer-Douglas-Peuker algorithm was applied (Figure 5.27). This happens every so often when RDP is applied due to the oversimplification of the curve. Nonetheless, the cleaned points presented a more stable path for the virtual drone to follow (Figure 5.28).

5.5 Discussion

The use of gestures is a more intuitive way of expressing complicated Flail code. Nonetheless, positional points have to be somewhat far apart to prevent underflows. This reduces the precision of the generated curve and might lead to some amount of distortion.

It was shown that the virtual drone was capable of following the cleaned up trajectories suitably well, even though some deformation was present. This is most likely due to estimation errors that occured in the `Polar.py` script.

It can be argued that the use of different filtering algorithms reduces the intuitivity level of this method for flight generation. Nonetheless, filtering is unavoidable. Real world data always bears noise that has to be removed. In an ideal scenario, users would be given an application that would allow them to manually fix the data to better suit their needs.

5.6 Summary

This chapter covered the use of gestures to control drone flight paths. The HTC Vive's wand controller was used to capture the performed motion. The `Polar.py` script (presented in Section 4.3) was modified to accept a list of points as input and to take the z-axis in consideration when planning paths.

After data points were acquired from the HTC Vive, they were cleaned and filtered. Points that were too close together were removed and a low pass filter was applied to the curve. Points had to be at least 0.0055 units apart to prevent underflows, but this resulted in less precise curves and in some distortion. The virtual drone was shown to follow the generated paths well but with some fluctuations. These were most likely caused due to the estimation made when generating the curves.

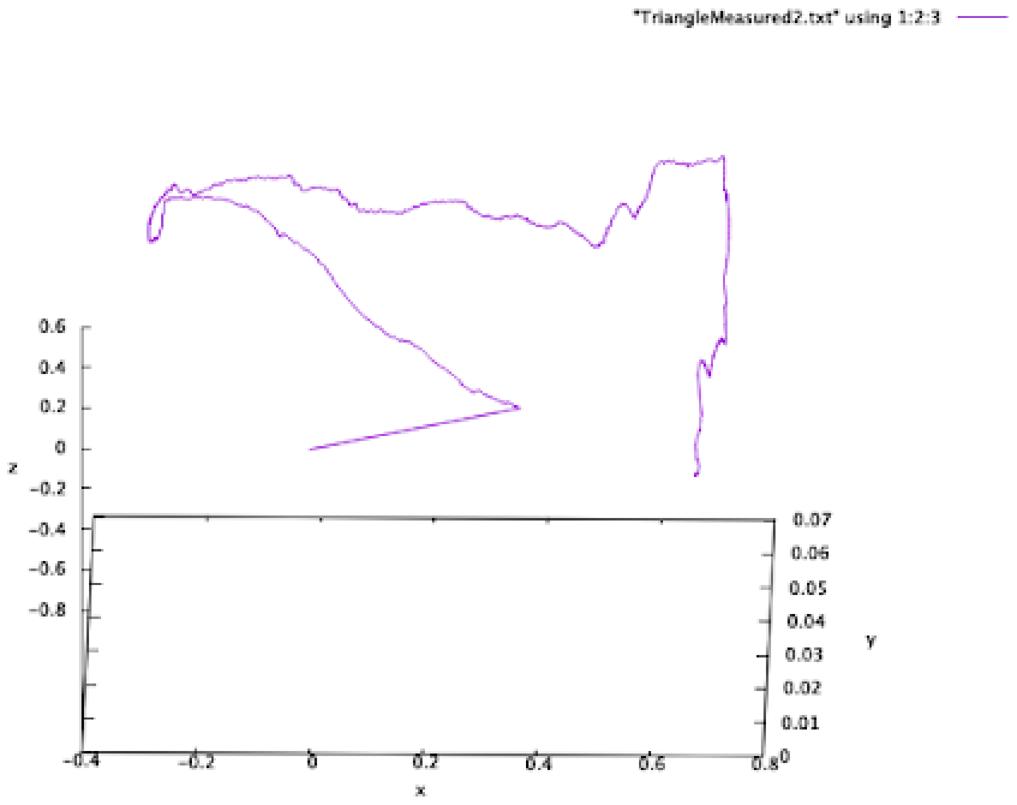


Figure 5.20: Original right triangle taken from the Vive

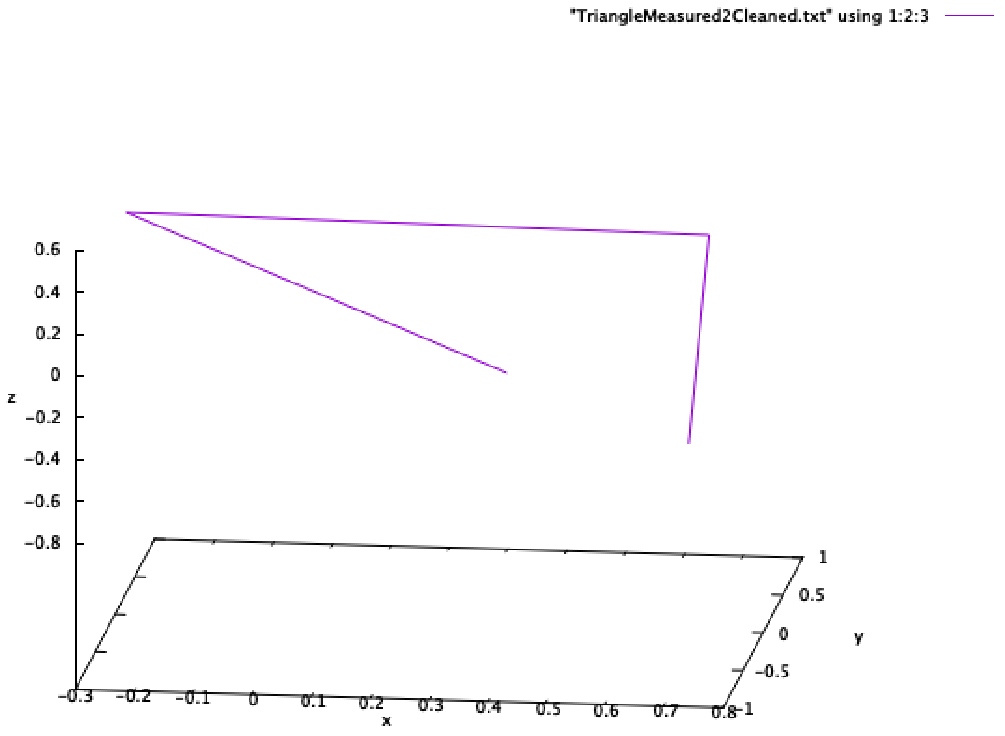


Figure 5.21: Cleaned up datapoints for the right triangle

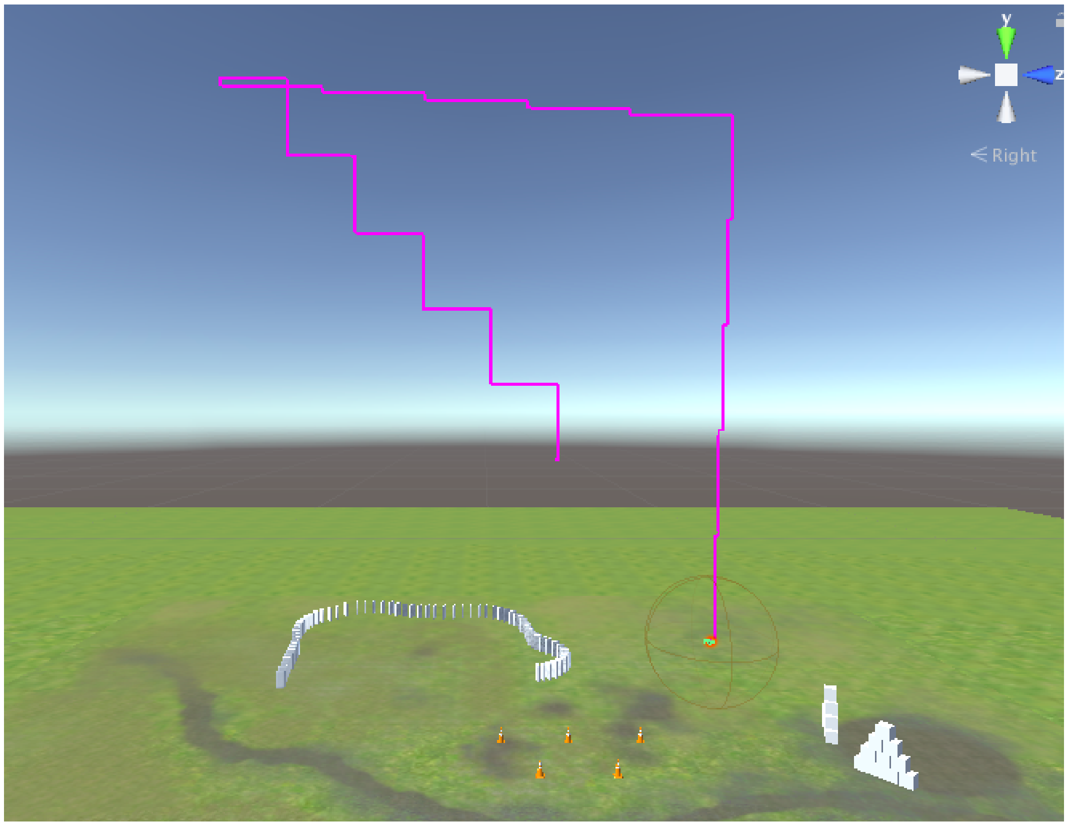


Figure 5.22: Simulated flight path based on the generated Flail program for the cleaned up right triangle

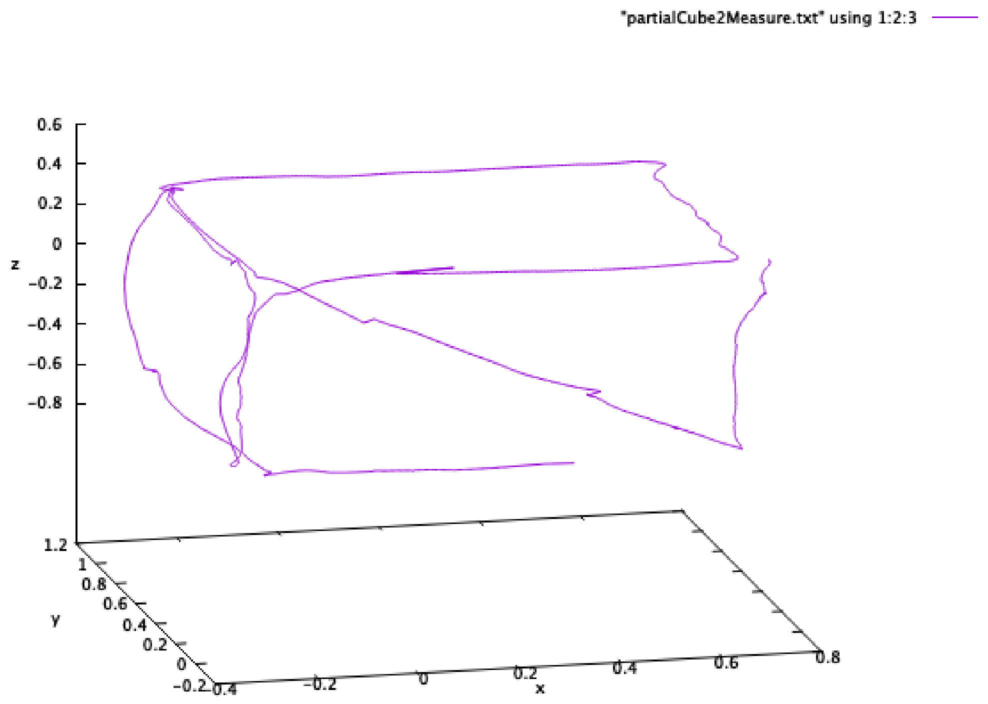


Figure 5.23: Original path taken from the Vive

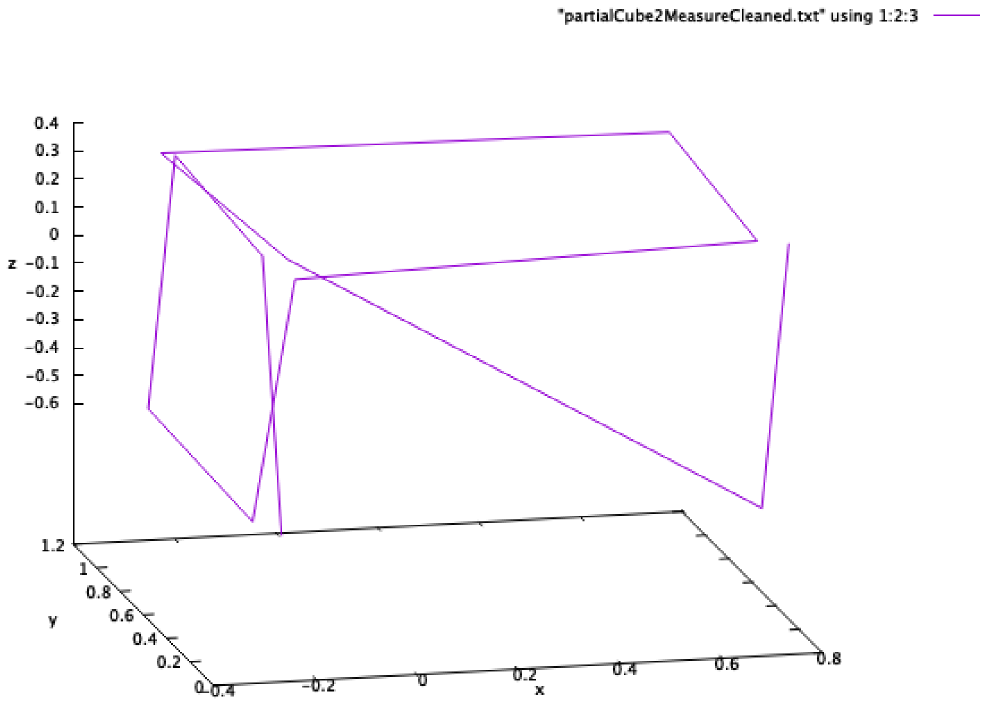


Figure 5.24: Cleaned up data points

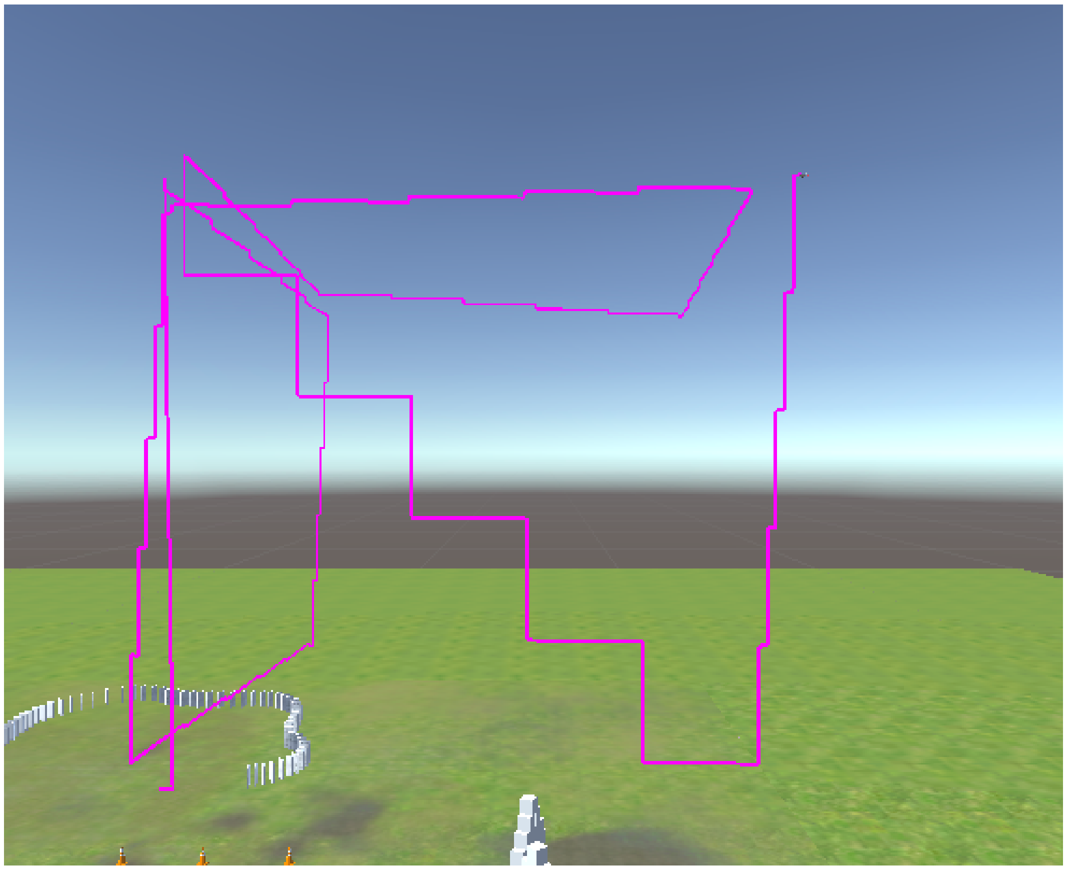


Figure 5.25: Simulated flight path based on the generated Flail program of the cleaned up cube

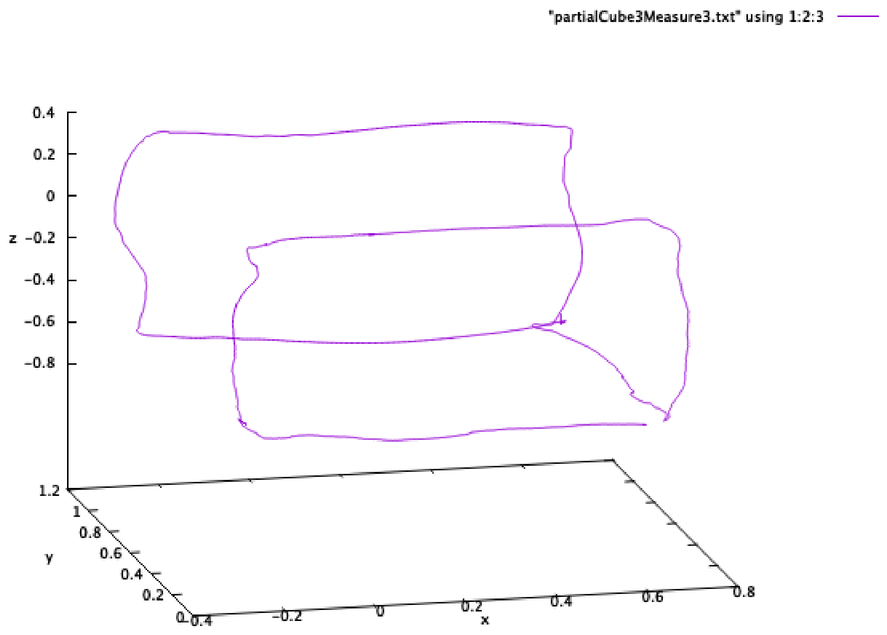


Figure 5.26: Original partial cube path taken from the Vive

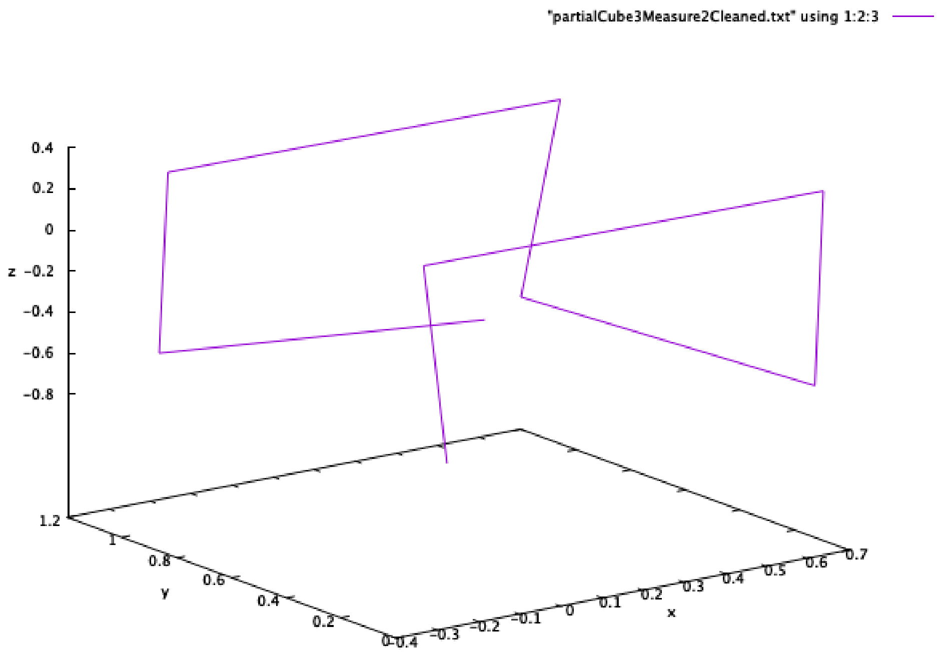


Figure 5.27: Cleaned up datapoints for the partial cube

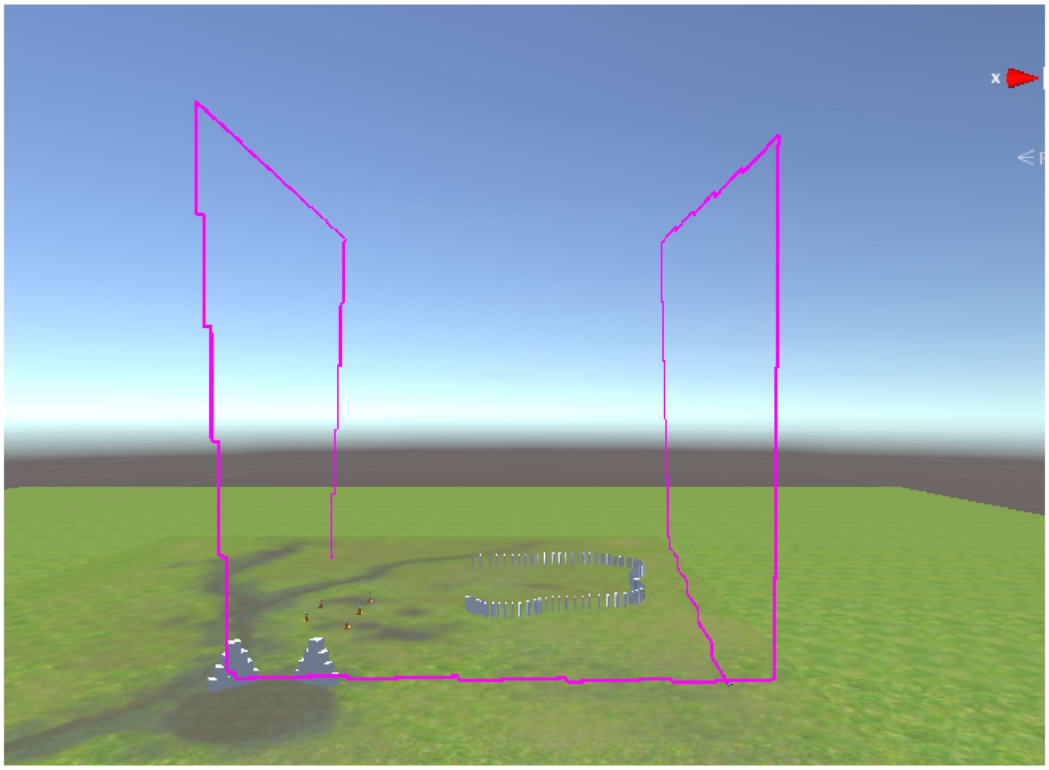
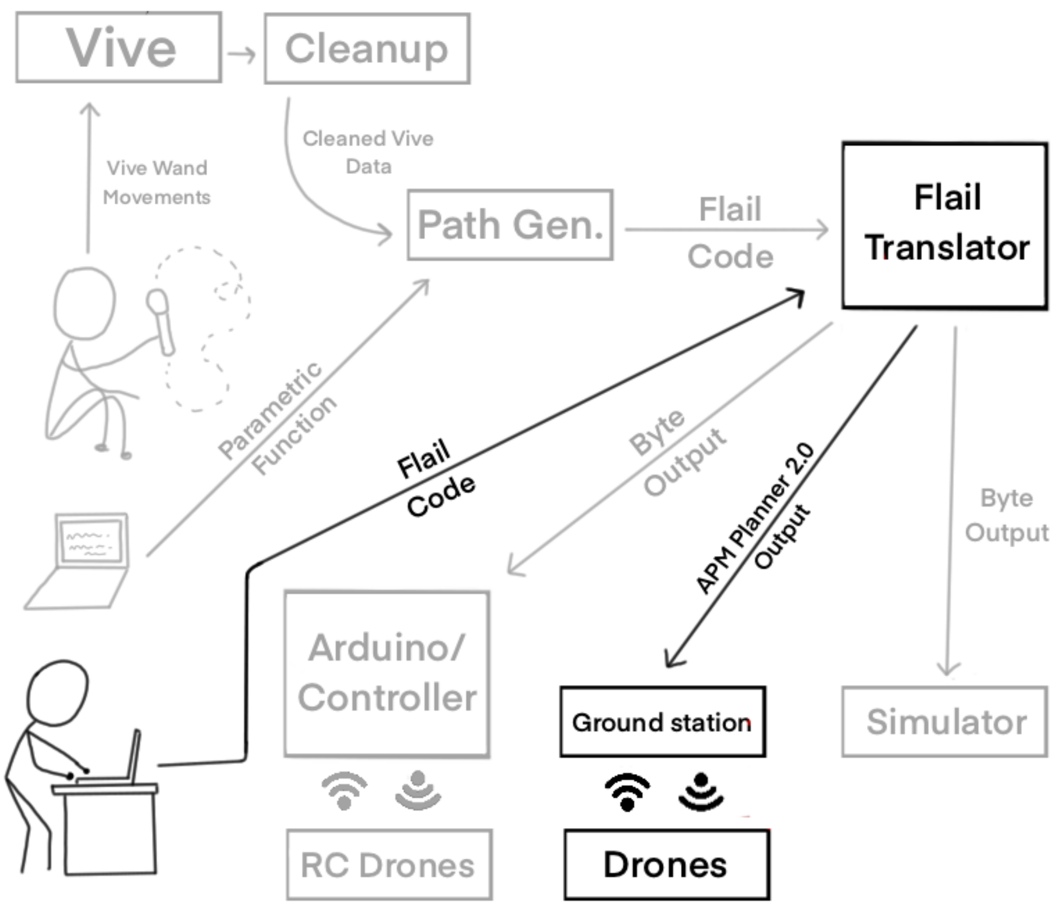


Figure 5.28: Simulated flight path based on the generated Flail program of the cleaned up partial cube

Chapter 6

Flail for GPS-based Drones



The last few chapters dealt with the construction of the Flail language and simulations to test it. We have yet to consider real life applications of Flail on physical drones beyond the microdrones in Section 3.2.1.

This chapter will focus on answering RQ4: What changes, if any, are necessary in order to get Flail to run on a commercial drone?

RQ4.1 How can we generate an output that is understandable to commercial drones?

There are a lot more limitations when flying a physical drone compared to a virtual one. Obstacles and weather, for example, are issues that might affect a drone's flight. Furthermore, controlling drones programmatically tends to be a bit complex as it's not the default way of flying them.

Companies sometimes include applications that allow users to select specific waypoints, or geo-referenced coordinates, to guide their drones.

These applications, however, tend to be brand specific and won't work for a drone of a different maker. To get around this, a Ground Station application was used. Ground station is a generalized waypoint application used to coordinate the flight trajectories of a number of different drones. Ground station applications are GPS based which prevents drones from being flown indoors.

RQ4.2 How can we generate an output file that is platform independent?

As mentioned, Flail was developed with the goal of being platform independent, allowing its output to be read by different drone models. APM Planner 2.0 was selected as the Ground Station intermediary between Flail and the drone. The option to generate an APM Planner readable output file was added to Flail.

This chapter will focus on the modifications needed to make the Flail translator output an APM Planner 2.0 readable file.

6.1 Platform Independence

One of the ideals behind Flail is to have the language be platform independent, or in other words, require minimal to no changes in order to function with different drone models or platforms. As Flail is a method used to describe trajectories, it doesn't necessarily have to be limited to drones. The output paths can be used to guide a number of different vehicles, like RC cars, for example. The only limitation is that certain vehicles, such as the aforementioned RC car, would be unable to perform the "ascend" and "descend" commands. This is easily dealt with by generating paths that would be limited to the XY plane.

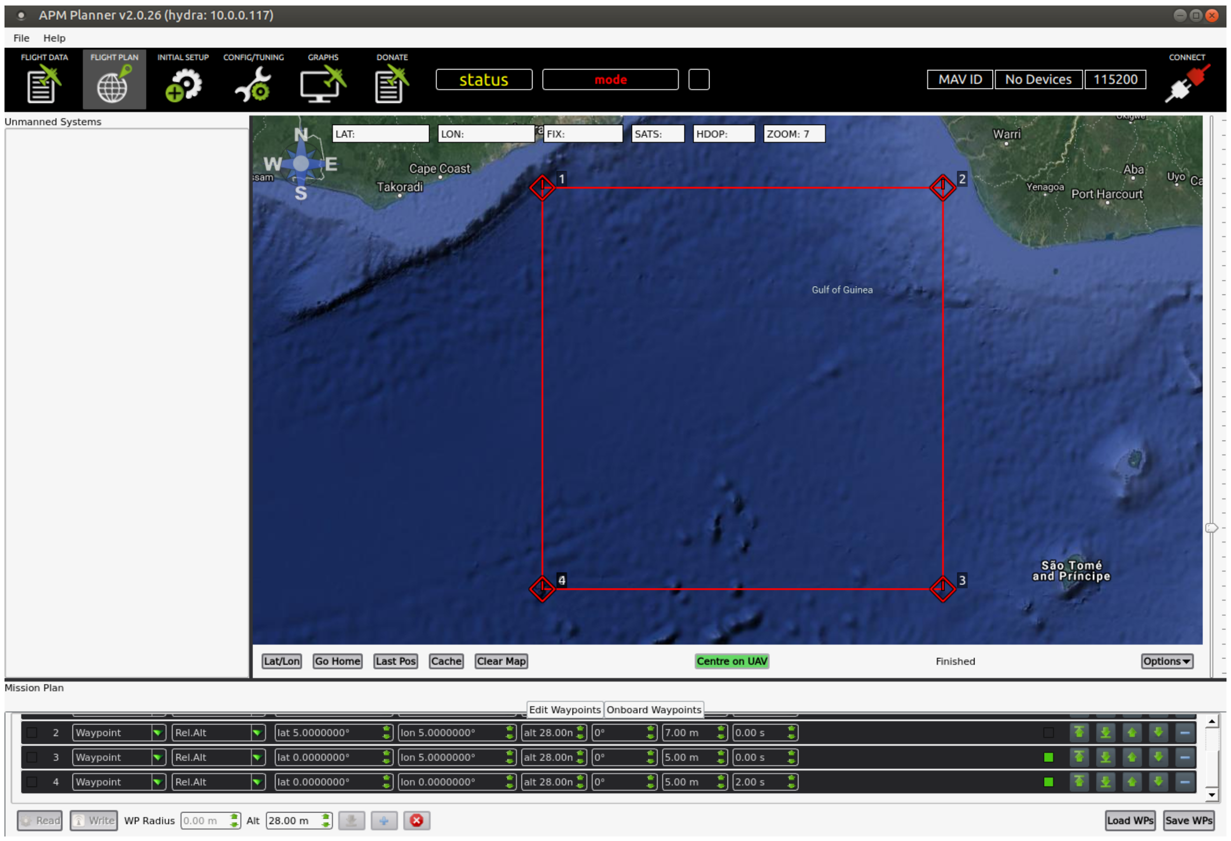
To make Flail compatible with different platforms, the translator had to be modified to generate an output that is interpretable by the designated platform.

To allow a number of different drone models to be reached, an additional output file was added to Flail. This file is readable by the selected Ground Station, an application that communicates with aerial vehicles via the Mavlink protocol. As such, the Ground Station is being used as an intermediary between Flail and potentially a number of different drone models.

6.1.1 Ground Station

Ground Station is typically software that runs on a "ground-based" computer, hence its name, that communicates with an UAV via wireless telemetry [51]. The benefits of using a Ground Station as a control medium for aerial vehicles is that, apart from displaying real time data, such as performance and position, it is also capable of controlling UAVs in flight.

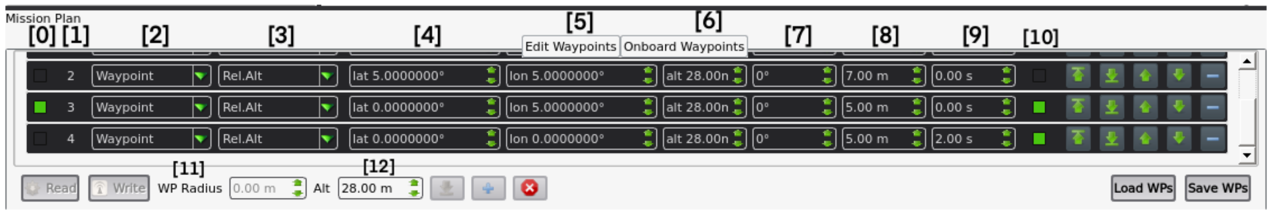
There are a number of different Ground Station software packages available. The more popular ones for desktop use include Mission Planner [52], APM Planner 2.0 [53], MavProxy [54], QGroundControl [55], and UgCS – Universal Ground Control System [56]. APM Planner 2.0 (APM 2.0) was chosen, as it's considered one of the better applications for Linux and MacOS computers [51]. APM 2.0 is an open source application for MAVlink based autopilots.



2	Waypoint	Rel.Alt	lat 5.0000000°	lon 5.0000000°	alt 28.00n	0°	7.00 m	0.00 s	
3	Waypoint	Rel.Alt	lat 0.0000000°	lon 5.0000000°	alt 28.00n	0°	5.00 m	0.00 s	🟩
4	Waypoint	Rel.Alt	lat 0.0000000°	lon 0.0000000°	alt 28.00n	0°	5.00 m	2.00 s	🟩

WP Radius 0.00 m Alt 28.00 m

Figure 6.1: Display example from APM Planner 2.0



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	Waypoint	Rel.Alt	lat 5.0000000°	lon 5.0000000°	alt 28.00n	0°	7.00 m	0.00 s	
🟩	3	Waypoint	Rel.Alt	lat 0.0000000°	lon 5.0000000°	alt 28.00n	0°	5.00 m	0.00 s	🟩
	4	Waypoint	Rel.Alt	lat 0.0000000°	lon 0.0000000°	alt 28.00n	0°	5.00 m	2.00 s	🟩

WP Radius 0.00 m Alt 28.00 m

Figure 6.2: APM Planner 2.0 flight plan options as seen in Figure 6.1.

Figure 6.2 displays the available flight options:

[0]: Mission start - Is this the initial coordinate for the mission?

[1]: Waypoint order - In what order should this waypoint be executed?

[2]: Action at waypoint - What action should the drone perform when it reaches this waypoint?

[3]: Coordinate frame - What type of altitude reference should the drone use for this waypoint?

[4]: Latitude in degrees - What is the latitude of this waypoint?

[5]: Longitude in degrees - What is the longitude of this waypoint?

[6]: Altitude in meters - What is the altitude (in meters) of this waypoint?

[7]: Yaw rotation - What is the drone's yaw rotation at this waypoint?

[8]: Uncertainty radius in meters - What radius should the drone use to specify whether it has reached its destination?

[9]: Time to loiter - How long should the drone wait when the waypoint is reached?

[10]: Continue automatically to next waypoint? - Should the drone move on to the next waypoint automatically?

[11]: Waypoint radius - What is the radius of the waypoint?

[12]: Default altitude - What is the default altitude when adding new waypoints?

Waypoints can be stored and loaded from a text file. The text file maintains the following format:
[waypoint order] [mission start] [coordinate frame] [action at waypoint] [time to loiter] [uncertainty radius in meters] [waypoint radius] [yaw rotation] [latitude in degrees] [longitude in degrees] [altitude] [continue automatically?]

For example, take the trajectory shown in figure 6.1. That mission file contains the following:

```
QGC WPL 110
0   1   0   16   0   5   0   0   0   0   0    1
1   0   3   16   4   2   0   5   5   0   28   1
2   0   3   16   0   7   0   0   5   5   28   0
```

```
3   0   3   16   0   5   0   0   0   5   28   1
4   0   3   16   2   5   0   0   0   0   28   1
```

Consider the first line of the file:

```
0   1   0   16   0   5   0   0   0   0   0   1
```

This line can be interpreted as a waypoint with the following characteristics:

0 - [waypoint order] - first waypoint to be executed

1 - [mission start] - this is the start of the mission

0 - [coordinate frame] - absolute altitude

16 - [action at waypoint] - treat this as a regular waypoint

0 - [time to loiter] - 0 seconds

5 - [uncertainty radius] - 5 meters

0 - [waypoint radius]- 0 meters

0 - [yaw rotation] - 0 degrees

0 - [latitude] - 0 degrees

0 - [longitude] - 0 degrees

0 - [altitude] - 0 meters

1 - [continue automatically?] - Yes

The other lines can be interpreted in a similar fashion.

6.2 The Addition of APM Planner 2.0 Output to Flail

A new output file was added to Flail, allowing the translator to generate an APM Planner 2.0 mission text file. For simplicity's sake, a few options were kept to their default values:

- Mission start: (1) for the first waypoint, (0) for all other waypoints;

- Coordinate frame: (0) for the initial waypoint so it maintains absolute altitude, (3) for all subsequent waypoints so they maintain relative altitude;

- Action at waypoint: (0) for all entries so that they are treated as default waypoints. (APM Planner 2.0 automatically treats the first waypoint entry as ”Home”);
- Time to loiter: (0) seconds;
- Uncertainty radius in meters: (5) meters;
- Waypoint radius: (0) meters;
- Continue automatically to next waypoint: (1) yes.

While most options of the mission text file were kept to their default values; the drone yaw rotation, latitude, longitude, and altitude parameters were calculated for each waypoint.

To calculate the drone rotation at each waypoint, it was necessary to keep track of the drone's current orientation vector and rotation matrix. The orientation vector keeps track of the drone's current direction, while the rotation matrix keeps track of the drone's current rotation. Whenever a `YawLeft` or `YawRight` instruction is executed, the rotation matrix is updated to reflect the designated yaw rotation. The updated rotation matrix is then applied to the orientation vector to update the drone's current direction. Therefore, the drone's current yaw rotation can be calculated by taking the arctangent of the current vector as shown by the code below:

```
# The curVector keeps track of the drone’s current orientation
curVector = initalVector = [1, 0, 0, 0]

# The rotMatrix keeps track of the drone’s current rotation
# It is initialized as an identity matrix
rotMatrix = identity matrix
```

```
## rotation about the Z axis
ROT_Z = lambda z: [[cos(z), -sin(z), 0, 0],
                   [sin(z),  cos(z), 0, 0],
                   [     0,       0, 1, 0],
                   [     0,       0, 0, 1]]

# Multiplies matrix1 * matrix2
def matMul(matrix1, matrix2)

# Multiplies matrix * vector
def vecMul(matrix, vector)

# Converts the given radian value into degrees
def degrees(radian)

# Rotate the drone counter-clockwise about its z-axis
def yawLeft(angle):
    rotMatrix = matMul(self.rotMatrix, ROT_Z(radians(BAM2float(ang))))
    curVector = vecMul(self.rotMatrix,self.initialVector)

# Rotate the drone clockwise about its z-axis
def yawRight(angle):
     rotMatrix = matMul(self.rotMatrix, ROT_Z(radians(BAM2float(-ang))))
     curVector = vecMul(self.rotMatrix,self.initialVector)

# Calculate the drone’s current heading in the range [0,2pi).
```

```
def heading():
    return degrees(atan2(-curVector[1], -curVector[0]) + pi)
```

The drone's current altitude is calculated by updating an `altitude` parameter. This parameter is initialized to 0 and updated whenever an `ascend` or `descend` instruction is executed.

```
# The drone's current altitude
altitude = 0

# Ascend the drone by the given distance
def ascend(distance):
    altitude += distance

# Descend the drone by the given distance
def descend(distance):
    altitude -= distance
    if altitude < 0
        altitude = 0
```

To calculate the latitude and longitude coordinates, the movement instruction calls [1] had to be converted into x,y coordinates. To do so, the drone's current location had to be updated every movement function call:

```
# Initialize the drone's position to be at the origin of the
```

[1]Forward, Backward, Left, and Right instructions

```
# coordinate system
# This variable keeps track of the drone’s current position
curPoint = [0, 0, 0, 0]

# The curVector keeps track of the drone’s current orientation
curVector = initialVector = [1, 0, 0, 0]

# Scales the vector by the given ’distance’ amount
# Returns vector * distance
def vecScale(vector, distance)

# Returns vector1 + vector2
def vecAdd(vector1, vector2)

# Given a (float) distance value, ’dist’:
def updateCurrentPosition(dist):
    scaledVector = vecScale (curVector, dist)
    curPoint = vecAdd(curPoint, scaledVector)

latitude = curPoint[1]
longitude = curPoint[0]
```

Every calculated value is then stored in a file called *gps.txt* that follows the format described in the previous section.

6.3 Evaluation

For reasons discussed later, the evaluation for this chapter will be performed by analyzing how closely APM Planner 2.0 was able to recreate the given trajectories.

6.3.1 Ground Station Waypoint Generation

As shown previously, APM Planner 2.0 is able to visually display the expected drone route by marking the given waypoints onto a map (Figure 6.1).

One thing that had to be taken in consideration, however, is that latitude coordinates range from ($90°N \leftrightarrow 90°S$) (0° at the equator) and longitude coordinates range from ($180°E \leftrightarrow 180°W$) (0° at the Greenwich meridian).

Therefore, the Cartesian coordinates have to be mapped to a rectangle with lower left corner at (-180, -90) and upper right corner at (180, 90). This is accomplished by an affine transformation. Besides, if we want to preserve angles, the applied scale has to be the same in x and y.

Cartesian coordinates do not convert directly into GPS coordinates, as the curvature of the Earth has to be taken in consideration. One can think of these as the parameters (θ, ϕ) in spherical coordinates and that a curve on the plane can be mapped onto the sphere in a similar way as texture mapping is applied in Computer Graphics.

Three tests were performed to analyse how closely this Ground Station would be able to re-create routes generated from the path generation script (given in Section 4.3) and Flail programs.

The first test involved a simple Flail program that directed a drone to fly in a square-like pattern:

```
# Drone is expected to be at the origin facing the positive x-axis
Forward(5);
```

```
Rotate(90);
Forward(5);
Ascend(20);

Rotate(90);
Forward(5);

Rotate(90);
Forward(5);

# Drone is now back at the origin
```

An APM Planner 2.0 output file was generated following the methods explained in Section 6.2. The resultant route is shown in Figure 6.5. The waypoint properties matched the expected values and the route is shown to accurately mark waypoints at GPS coordinates: (0, 0) $\rightarrow$ (5, 0) $\rightarrow$ (5, 5) $\rightarrow$ (0, 5) $\rightarrow$ (0, 0).

The second test used the path generation script to create a flight route that followed the "amoeba" polar function, as shown in Figure 6.6.

Finally, the third test used the path generation script to create a flight route that displayed the five petal rose polar function, as shown in Figure 6.7.

It is noticeable that the generated routes are very large, which is not desirable as drones are incapable of flying such routes in one go due to hardware limitations. Furthermore, it might be of interest to control where the drone passes through. For example, generating a route that passes through three distinct latitude and longitude coordinates, which will also affect the scale of the path. This can be done by using a 2D affine transformation to map three points of the original curve (frame A) onto the GPS coordinates of three distinct latitude/longitude pairs (frame B).

Let three points taken from the original curve be referred to as triangle RPQ. To acquire the first transformation matrix, convert this triangle to a vector format as shown by the homogeneous transformation matrix below:

$$\begin{bmatrix} P_x - R_x & Q_x - R_x & R_x \\ P_y - R_y & Q_y - R_y & R_y \\ 0 & 0 & 1 \end{bmatrix}$$

The mapping from triangle RPQ in frame A onto triangle R'P'Q' in frame B is given by the matrix:

$$T_{A \to B} = T^{-1} * \begin{bmatrix} P'_x - R'_x & Q'_x - R'_x & R'_x \\ P'_y - R'_y & Q'_y - R'_y & R'_y \\ 0 & 0 & 1 \end{bmatrix}$$

Apply transformation, T, to each point in the original curve to generate a transformed curve as shown in Figure 6.3. To further demonstrate this technique, the five petal rose depicted in Figure 6.7 underwent an applied transformation to constrain it to a few blocks within Calgary, as shown in Figure 6.4. To generate this curve, three latitude and longitude coordinates within Calgary were selected: [-114.070846, 51.048615, -114.071485, 51.049512, -114.070004, 51.049525] and three parameterized values were selected from the original curve: [0, $\pi/3, 2\pi/3$]. As the five petal rose is a polar function, the corresponding x/y values can be easily extracted from the selected values. These two frames, the latitude and longitude list and the coordinates from the original curve, can then be used to create a transformation matrix that can be applied to each point in the five petal rose.

6.3.2 Real World

The IRIS drone was used to test the Ground Station connection. Due to time constraints and safety concerns, I was unable to perform an outdoor flight test and only tested the

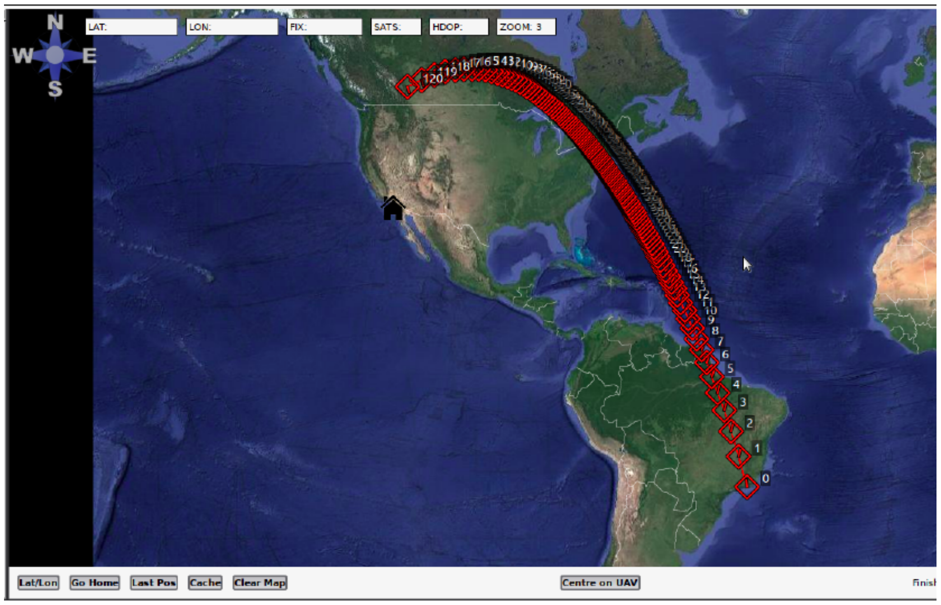
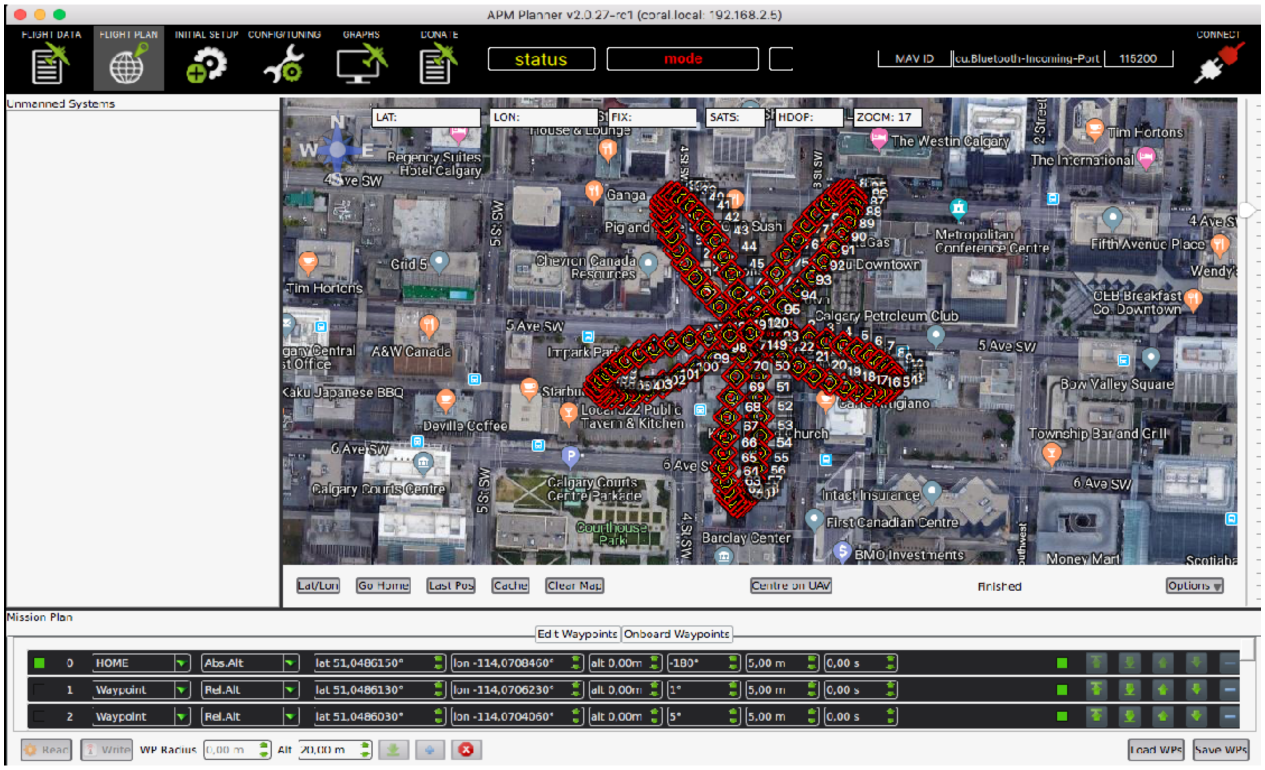


Figure 6.3: This parabola maps to Rio de Janeiro, New York, and Calgary. This route is used for illustrative purposes and is not assumed to be a suitable route for commercial drones due to battery limitations. If a smaller scale was desired, one could use latitude and longitude pairs that were closser to one another.

Ground Station indoors. Therefore, the drone was unable to fly due to GPS being unusable within buildings. Nonetheless, this section will cover how to establish a connection between an Ardupilot compatible drone (such as IRIS) and APM Planner 2.0.

The drone must have a radio antenna in order to communicate with the Ground Station (the IRIS drone used a 3DR 915 MHz antenna). To connect the drone to the Ground Station requires the following [57]:

1. Connect the antenna receiver to a computer;
2. Power the radio attached to the drone by plugging in its battery;
3. Open the Mission Planner and go to the Initial Setup — Optional Hardware — SiK Radio page;
4. Select the correct COM port and set the baud rate to 57600 (Baud rates may vary depending on the used radio).



0	HOME	Abs.Alt	lat 51,0486150°	lon -114,0708460°	alt 0,00m	-180°	5,00 m	0,00 s
1	Waypoint	Rel.Alt	lat 51,0486130°	lon -114,0706230°	alt 0,00m	1°	5,00 m	0,00 s
2	Waypoint	Rel.Alt	lat 51.0486030°	lon -114.0704060°	alt 0.00m	5°	5.00 m	0.00 s

WP Radius 0,00 m Alt 20,00 m

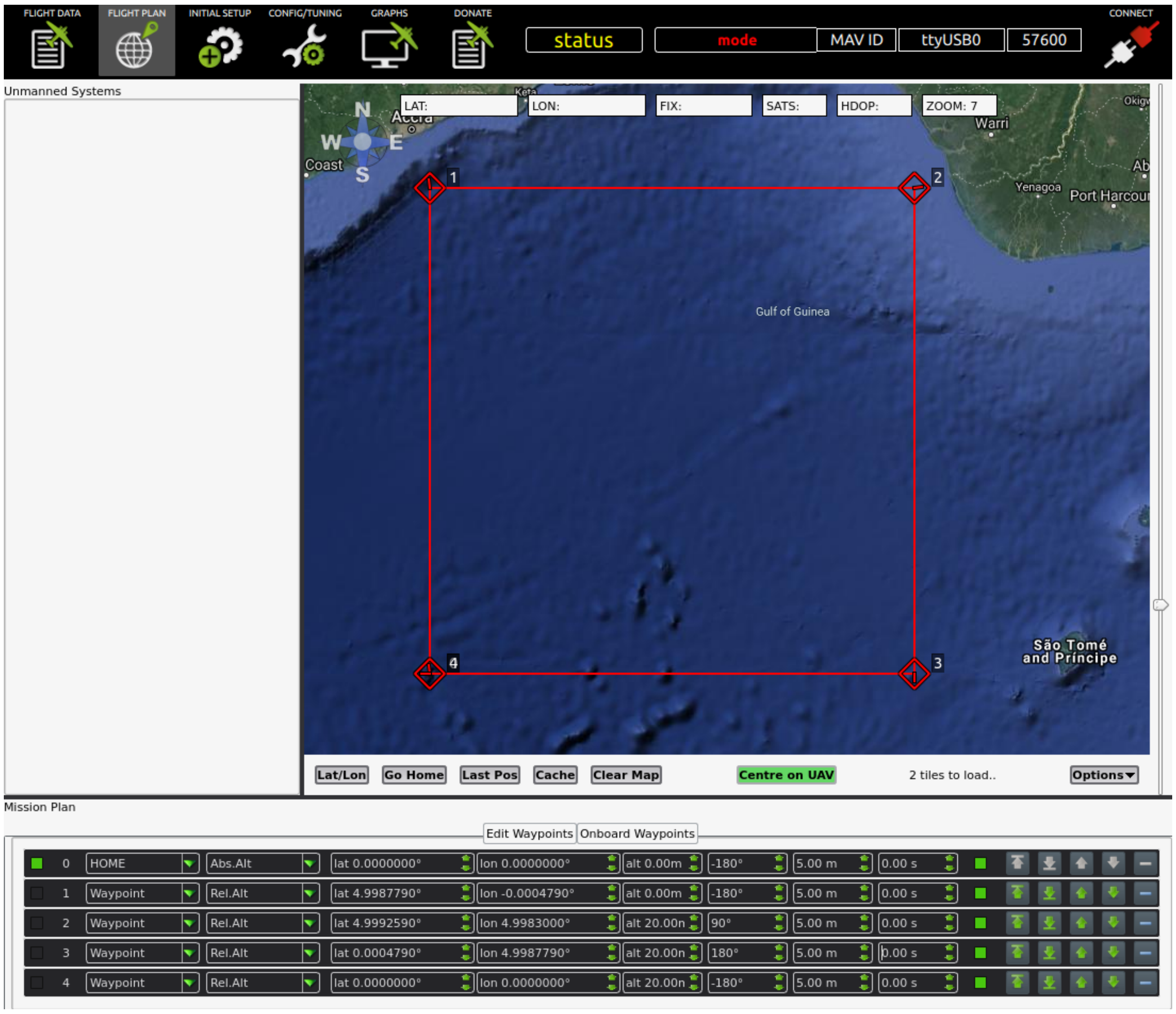
Figure 6.4: Mapping of the five-petal rose within Calgary. This route used three latitude and longitude pairs within Calgary to create a scaled down version of the original curve. Note that distortions may occur.

If the connection is successful, then the "Flight Plan" page in APM Planner 2.0 will show a pulsating "heartbeat". The drone's in flight data can be found under the "Graphs" page. This includes all movement functions, such as drone rotations, performed since its connection as shown in Figure 6.8. Finally, the "Flight Data" page will display information regarding the drone's current flight data (e.g., current altitude, current position, etc...).

As mentioned, the drone was kept indoors and therefore no satellite connections were established, (Figure 6.9) preventing drone lift-off.

6.4 Summary

This chapter covered the use of a Ground Station to facilitate the use of Flail in different drones. APM Planner 2.0 was selected as the chosen Ground Station. A new output file

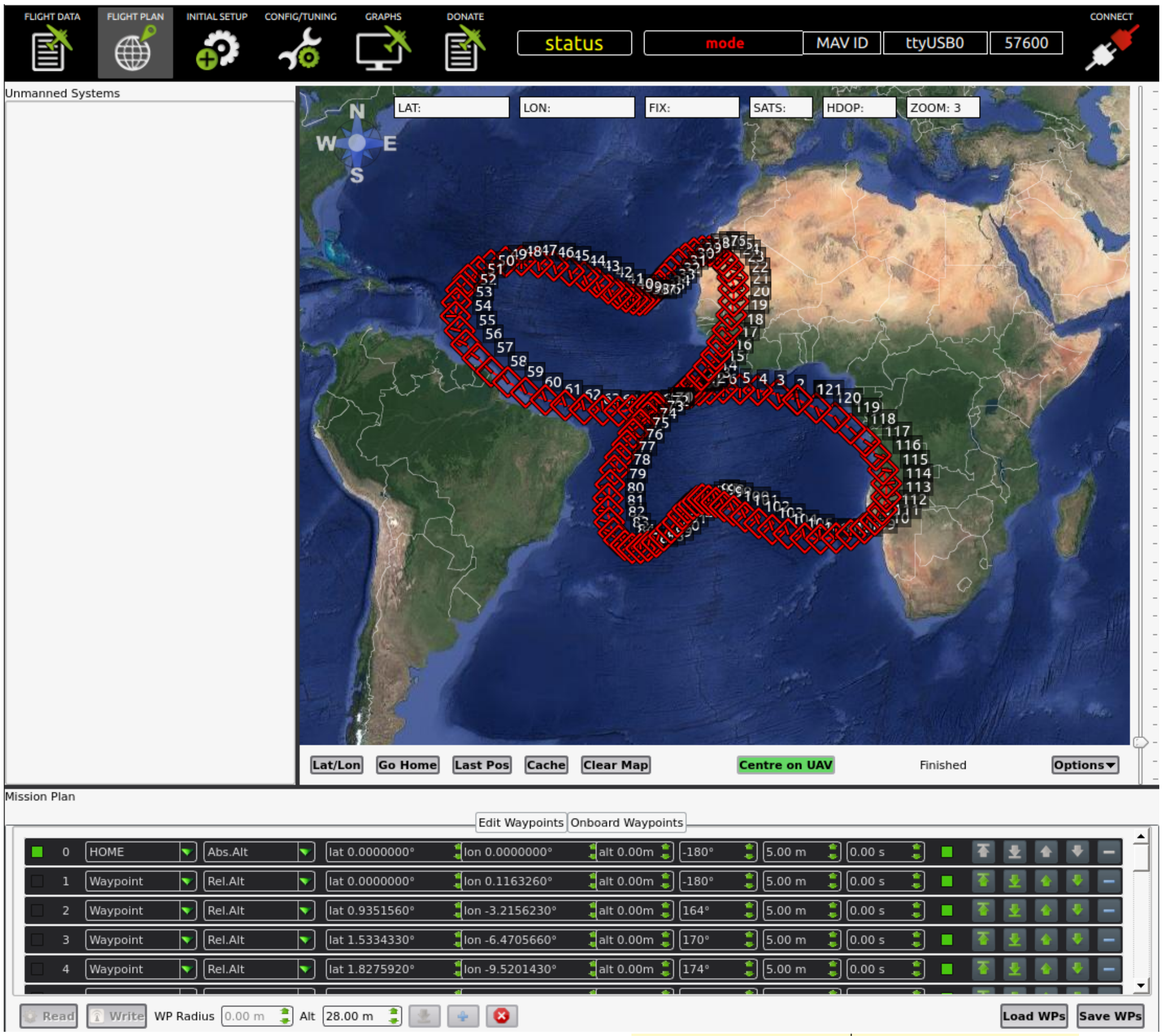


🟩	0	HOME	Abs.Alt	lat 0.0000000°	lon 0.0000000°	alt 0.00m	-180°	5.00 m	0.00 s
	1	Waypoint	Rel.Alt	lat 4.9987790°	lon -0.0004790°	alt 0.00m	-180°	5.00 m	0.00 s
	2	Waypoint	Rel.Alt	lat 4.9992590°	lon 4.9983000°	alt 20.00n	90°	5.00 m	0.00 s
	3	Waypoint	Rel.Alt	lat 0.0004790°	lon 4.9987790°	alt 20.00n	180°	5.00 m	0.00 s
	4	Waypoint	Rel.Alt	lat 0.0000000°	lon 0.0000000°	alt 20.00n	-180°	5.00 m	0.00 s

Figure 6.5: Resultant drone route generated from a Flail program that directs a drone to fly in a square-like pattern. Note that there is an ”extra” waypoint as the first entry. This denotes the starting position of the drone at the origin (0, 0).

type was added to Flail, to allow its outputs to be interpreted by the Ground Station.

An outdoor flight test was not performed due to time constraints and safety reasons. This chapter's evaluation was performed by checking the waypoint generations displayed in APM Planner 2.0. The waypoints were accurately translated between Flail and the Ground Station.



🟩	0	HOME	Abs.Alt	lat 0.0000000°	lon 0.0000000°	alt 0.00m	-180°	5.00 m	0.00 s	🟩
	1	Waypoint	Rel.Alt	lat 0.0000000°	lon 0.1163260°	alt 0.00m	-180°	5.00 m	0.00 s	🟩
	2	Waypoint	Rel.Alt	lat 0.9351560°	lon -3.2156230°	alt 0.00m	164°	5.00 m	0.00 s	🟩
	3	Waypoint	Rel.Alt	lat 1.5334330°	lon -6.4705660°	alt 0.00m	170°	5.00 m	0.00 s	🟩
	4	Waypoint	Rel.Alt	lat 1.8275920°	lon -9.5201430°	alt 0.00m	174°	5.00 m	0.00 s	🟩

WP Radius 0.00 m Alt 28.00 m

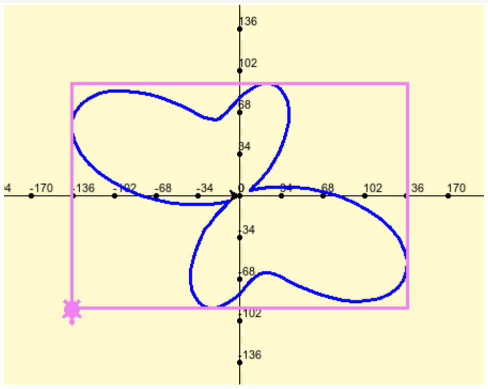
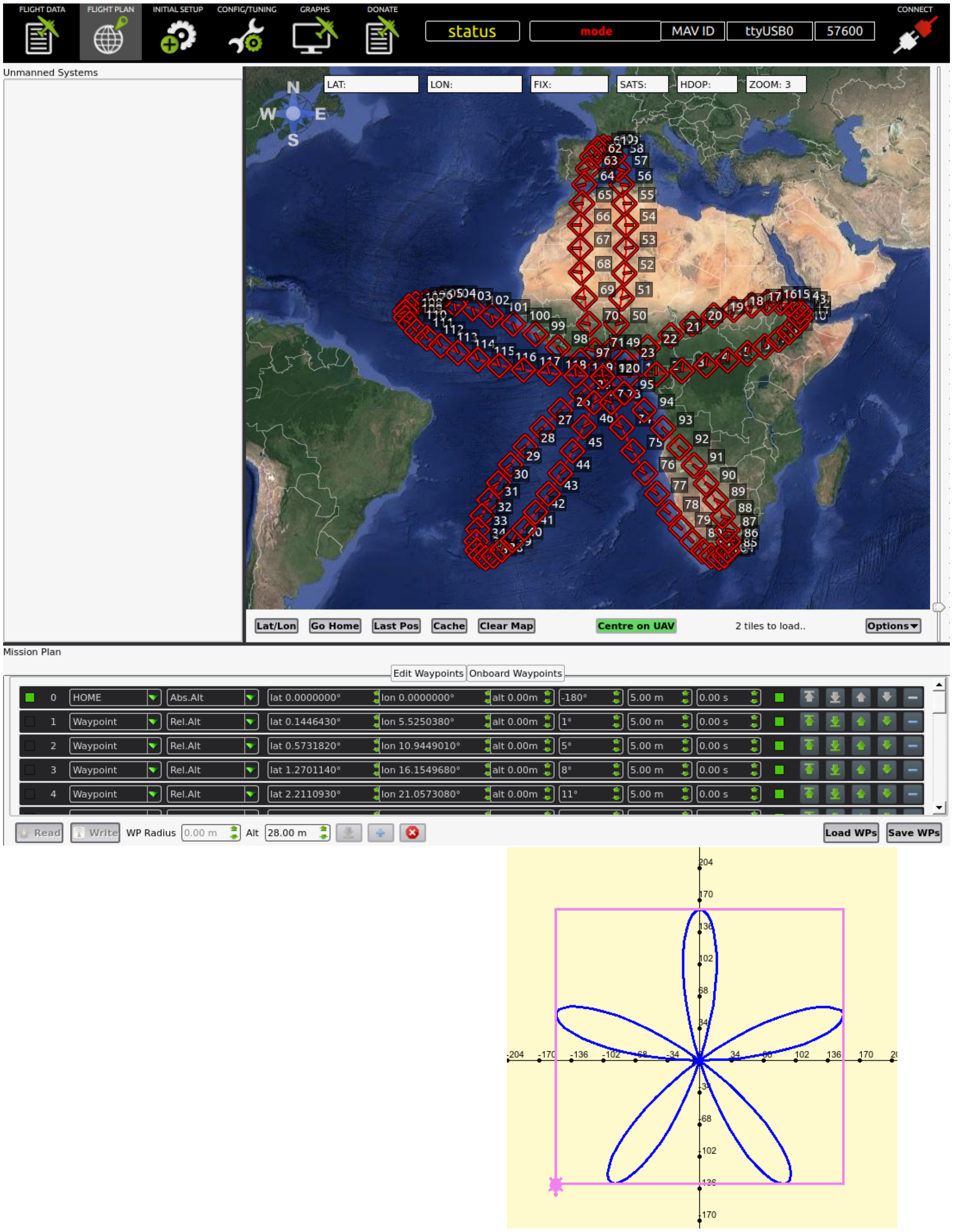


Figure 6.6: Re-creation of the amoeba polar function inset as a series of waypoints



WP Radius 0.00 m Alt 28.00 m

🟩	0	HOME	Abs.Alt	lat 0.0000000°	lon 0.0000000°	alt 0.00m	-180°	5.00 m	0.00 s
	1	Waypoint	Rel.Alt	lat 0.1446430°	lon 5.5250380°	alt 0.00m	1°	5.00 m	0.00 s
	2	Waypoint	Rel.Alt	lat 0.5731820°	lon 10.9449010°	alt 0.00m	5°	5.00 m	0.00 s
	3	Waypoint	Rel.Alt	lat 1.2701140°	lon 16.1549680°	alt 0.00m	8°	5.00 m	0.00 s
	4	Waypoint	Rel.Alt	lat 2.2110930°	lon 21.0573080°	alt 0.00m	11°	5.00 m	0.00 s

Figure 6.7: Re-creation of the five petal rose polar function inset as a series of waypoints

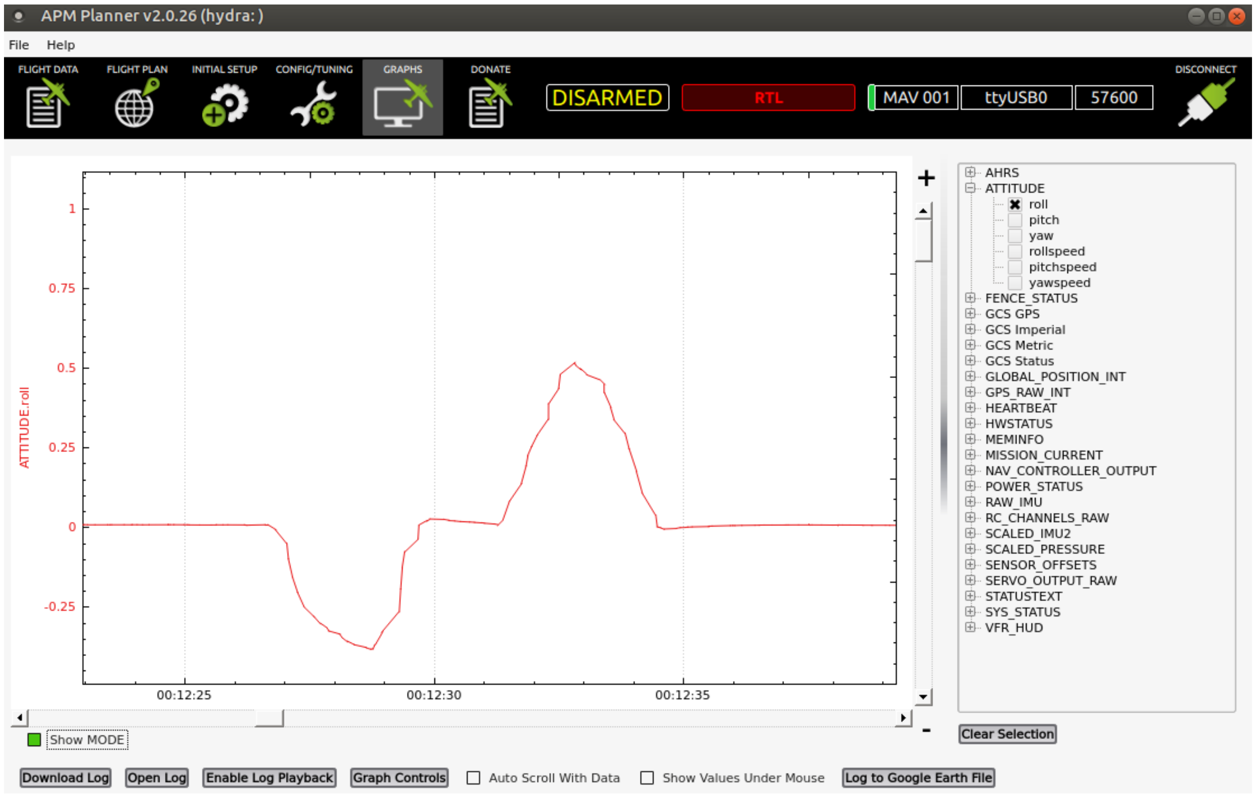
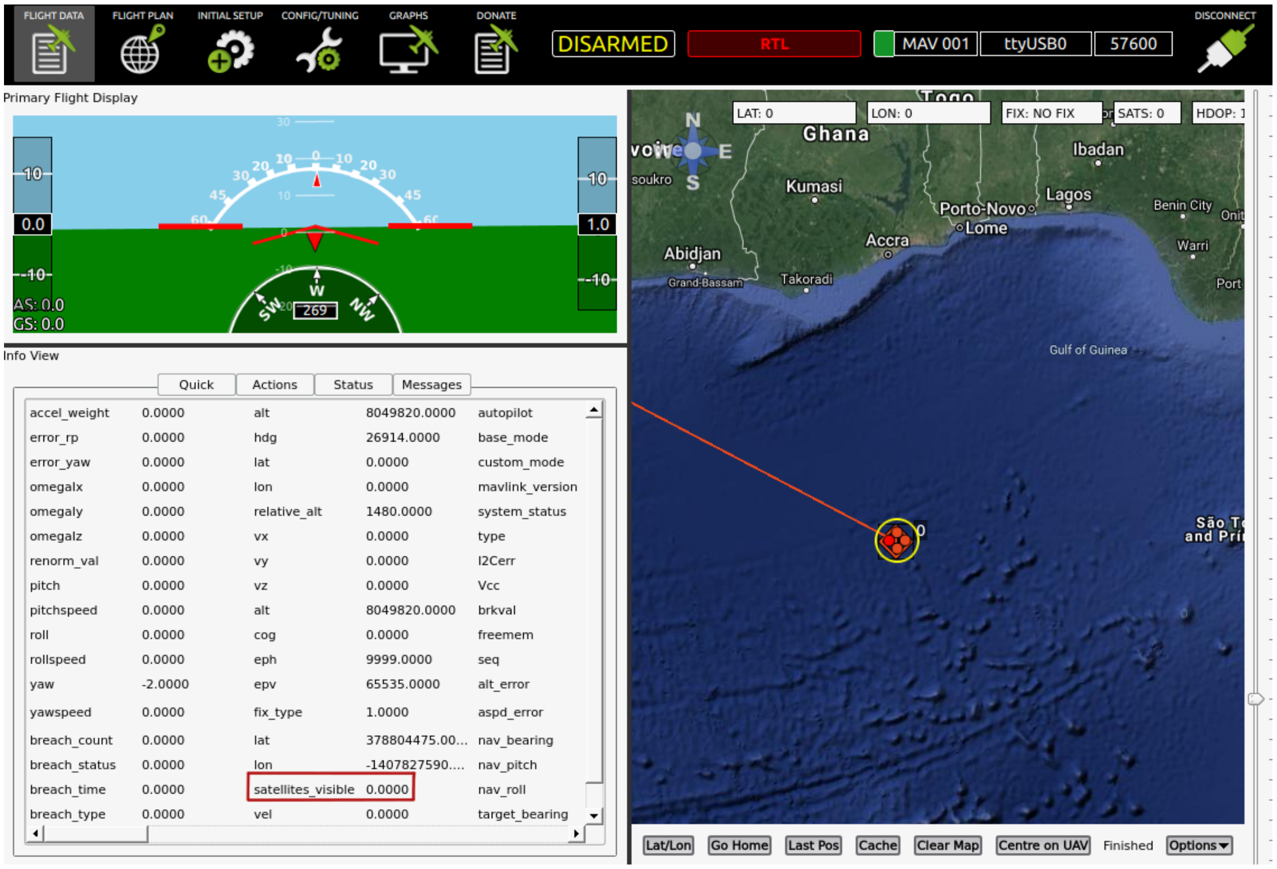


Figure 6.8: APM Planner 2.0 records drone features during flight. This graph demonstrates the drone's roll rotation. This was generated by picking up the drone and rolling it manually.



accel_weight	0.0000	alt	8049820.0000	autopilot
error_rp	0.0000	hdg	26914.0000	base_mode
error_yaw	0.0000	lat	0.0000	custom_mode
omegalx	0.0000	lon	0.0000	mavlink_version
omegaly	0.0000	relative_alt	1480.0000	system_status
omegalz	0.0000	vx	0.0000	type
renorm_val	0.0000	vy	0.0000	I2Cerr
pitch	0.0000	vz	0.0000	Vcc
pitchspeed	0.0000	alt	8049820.0000	brkval
roll	0.0000	cog	0.0000	freemem
rollspeed	0.0000	eph	9999.0000	seq
yaw	-2.0000	epv	65535.0000	alt_error
yawspeed	0.0000	fix_type	1.0000	aspd_error
breach_count	0.0000	lat	378804475.00...	nav_bearing
breach_status	0.0000	lon	-1407827590....	nav_pitch
breach_time	0.0000	satellites_visible	0.0000	nav_roll
breach_type	0.0000	vel	0.0000	target_bearing

Figure 6.9: No satellites can be found indoors

Chapter 7

Reflection, Future Work, and Conclusion

7.1 Research Questions

Flail was proposed as a language to programmatically define flight paths for drones. Four distinct research questions were addressed in the previous chapters.

RQ1 - How should we design a time-based programming language that allows us to describe movement paths for drones?

RQ1.1 - How do we describe drone movements as instructions?

In this version, Flail used time based functions to control drone trajectories. Movements were given as (instruction, parameter) pairs, where the instruction was one of the ten available Flail commands (as shown in Chapter 3), and the parameter was either an intensity value (if using a movement instruction) or a duration value (if using a wait instruction).

An intensity value controls how fast the drone should move and was given as a value between 0.0 (no speed) and 1.0 (maximum speed). A duration value was given as an integer and was used to control the duration of the wait function. The benefit of the time-based version of Flail is that multiple instructions could be set at once. If a forward command was called along with a move right command, then the drone would be set to fly in a diagonal path until the instructions were unset, giving users a decent amount of flexibility on how to direct their drones.

RQ1.2 - What time based functions do we need?

Instructions were separated by wait functions. These were split into two categories: wait for a certain amount of seconds (`wait`) and wait for a certain amount of milliseconds (`waitMilli`). Wait instructions were used to control how long the previously set instructions would run for.

RQ2 : How do we create a distance based DSL capable of interacting with drones?

RQ2.1 - How can we specify distance in Flail and what kind of distance functions would we need?

In this version, Flail used distance based functions to control drone trajectories. A new distance mode was added, which, when activated, made Flail accept floating points as parameters and to treat each parameter as a distance to be traversed.

```
SetMode(distance); # Flail now expects distance parameters
Forward(5.5); # Move forward for 5.5 units
Left(2.0); # Move to the left for 2 units
```

RQ2.2 - How can we specify the scale of the trajectory?

The chosen scale will control the dimensions of the generated path, or in other words, how large the path will be. Different scalings will lead to different unit sizes. To help users control the dimensions of their trajectories, the path generation script (presented in Chapter 4) allows users to select the size of the used scale. If users are manually creating their Flail code, then they'd be responsible for selecting their own scale system.

RQ2.3 - How can we specify repeated movements?

A looping function was added to Flail to allow instructions to be repeated. This simple loop uses the following structure:

```
Repeat n {
    Instructions
}
```

Where the instructions between the braces will be repeated n times.

RQ3 - How can we use gestures to simplify the generation of Flail code?

Even though it is straightforward to use Flail to create simple flight paths, it becomes progressively harder when working with more complex trajectories. Flail requires users to manually calculate the drone's translations and rotations, as such, complicated trajectories will require more calculations than a simple one, which is tiring for users. To make the path generation process a bit less painful, the HTC Vive was used to allow users to use gestures to describe their desired paths. A user could move the Vive wand around and use its stored X,Y,Z coordinates to generate a Flail file.

RQ4 - What changes, if any, are necessary in order to get Flail to run on a commercial drone?

RQ4.1 - How can we generate an output that is understandable to commercial drones?

Ground Station was used as an intermediary between Flail and the drone. Ground Station communicates with UAVs via wireless telemetry and can specify routes through waypoints, or map markers that pinpoint locations the drone should fly to. One limiting factor, however, is that Ground Station is GPS based and as such forces drones to be flown outdoors, as GPS can not be captured indoors.

RQ4.2 - How can we generate an output that is platform independent?

Platform independence refers to an application that requires minimal to no changes to allow it to function in a different system or domain. The difficulty for Flail to achieve platform independence is that every drone has its own firmware, and each require a number of modifications in order to get Flail to run on it. To get around this, Ground Station was used as an intermediary. Flight paths can be defined as "waypoints" or markers on a map that denote where a drone should fly to.

7.2 Reflection

7.2.1 Language Design

Flail has two ways to describe flight patterns: a time-based version and distance-based version. Each version has its own advantages and disadvantages.

As shown in Chapter 2, a number of different languages follow a similar time-based approach to flight control as the one used in Flail (Node AR, Tynker's drone app). It is fairly intuitive to use time as a way to describe motion, however, it is limiting in its description of complex paths. It isn't easy to predict how far a drone will fly when using such a technique, limiting its use in more intricate scenarios. On the other hand, this approach allows multiple instructions to be set at once, giving users a bit more control over directional changes in the drone's flight path.

To overcome the lack of distance precision that occurs in the time-based version of Flail, a distance mode was added. In this mode, motion is described by distance, or by how far a drone will fly before changing directions. While this version adds precision to flight trajectories, it becomes cumbersome to use in more complex routes as users have to manually measure out each directional change.

As such, users have to decide which version of Flail will better suit their needs. If they require more control over the direction of the drone's flight then they should use the time-based version. However, if they require more precision in their flight paths, then they should use the distance-based version. This version also has the added benefit of working alongside the path generation script, which further facilitates the user's route planning.

7.2.2 Issues When Developing Flail

One issue that I did not consider for a while was the size of the Flail output files. Initially, outputs were generated based on the literal interpretation of (instruction, parameter) pairs. As Flail only outputs bytes, parameter values were limited to 255_{10}. If a parameter was larger than 256_{10}, it would be broken down into chunks of 256_{10} and each chunk would be passed in with an instruction byte. For example, `Forward (800)` would be translated to:

$$0x8, 0xFF, 0x8, 0xFF, 0x8, 0xFF, 0x8, 0x20$$

Which effectively quadrupled that lone forward instruction. The larger the parameter, the larger the output. Flail files got progressively larger and larger, and soon became unusable. The issue was fixed after I added the `repeatNextInstFor` command, which truncated duplicated instructions to 6 bytes by following the format shown below:

[repeatNextInstFor] [Number of repetitions] [instruction] [256] [Instruction] [Remainder]

Forward(800) for example, would be translated to:

$$0xC, 0x3, 0x8, 0xFF, 0x8, 0x20$$

This change was beneficial when binary angle measurements (Chapter 4) were used to convert floating point numbers into integers, as the conversions usually led to large integer values.

7.3 Future Work

Flail still requires some work in order to become a fully-fledged method for trajectory planning. These involve safety concerns and further development.

7.3.1 Exception Mechanism

There is one safety concern that has yet to be addressed: Flail is currently incapable of performing collision detection, and flies drones blindly. There is a large chance that a drone will crash if it is flown in a space with a lot of physical constraints, e.g., indoors.

An exception mechanism would have to be put in place which would allow drones to detect "exceptions" in its path and recalculate its route. Anything that would knock the drone off its intended path would be treated as an exception.

Imagine that a drone intended to fly in a square-like pattern but had a tree in its way. The drone would have to stop before crashing, divert away from the tree, and continue its intended path as shown in Figure 7.1.

7.3.2 Curve Editor Application

Chapter 5 presented the use of gestures as a more intuitive approach to path generation. That chapter, however, also demonstrated that each generated trajectory displayed a lot of noise, which made the paths unusable in their original state. The trajectories would have to be filtered and smoothed out before a drone could attempt to fly it. The cleaning procedure used to do so employed a set of filtering and smoothing algorithms to clean the paths. The addition of these algorithms reduced the level of intuitiveness of the proposed method, as users now had to be familiar with the smoothing algorithms before they were able to generate Flail code.

To ease the users into the process, an external application for curve editing is proposed. This application would allow users to edit paths and to apply whatever filtering algorithm

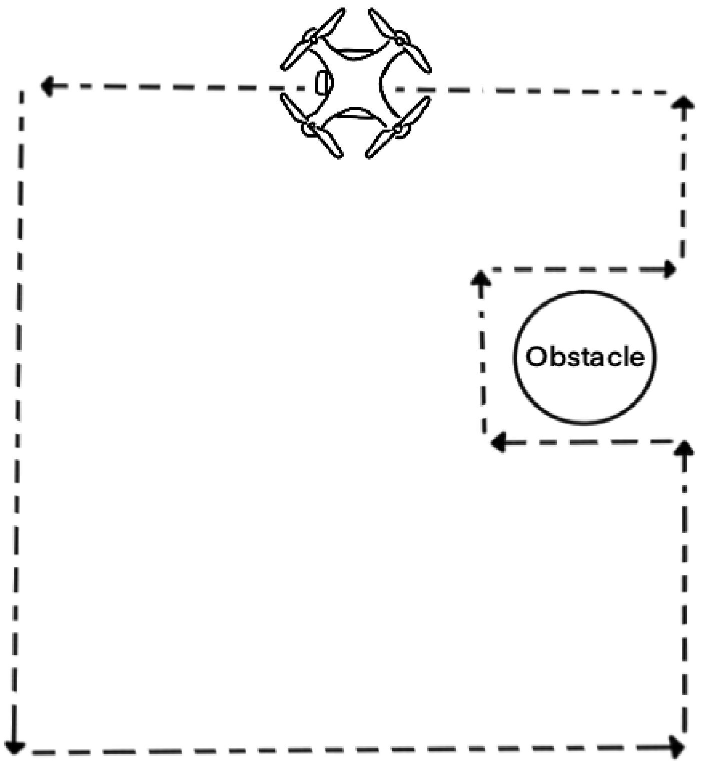


Figure 7.1: Possible path recalculation to prevent the drone from hitting the obstacle in its path

they deemed necessary. This application would have to be able to do the following:

1. Present a number of different smoothing and filtering algorithms for a user to choose from.
2. Allow users to manually move path points to different positions, i.e. allow users to edit the generated trajectory.
3. Display previews of the user's modified paths.
4. Generate Flail code based off of the modified trajectory.

This application would facilitate the path cleaning procedure, while also allowing users to further modify the trajectories as they desired, giving them more freedom to design their routes.

7.3.3 Further Development and Tests

To increase Flail's ability to interact with drones, it would have to be further developed to access all drone capabilities, which would include sensor data and video support. One possibility is to use Flail to describe sensor-centric paths, or paths that are defined by sensor functionality. For example, a route that is generated to maintain the camera facing an object.

One limitation with updating Flail to encompass drone sensors, however, is that not all drones have access to the same functionalities. As such, Flail would have to consider which drone model it is interacting with, before attempting to access specific controls.

Another Flail expansion that would be of interest is to implement "on trigger events", or conditional statements. Tynker's Parrot Drone app (shown in Chapter 2) is capable of performing certain actions when specific events are triggered. For example, making the drone lift off as soon at it touches the floor.

Conditional statements would give users more control over drone actions when it reaches specific parts of its flight trajectory. This would be especially useful if Flail had access to the drone's sensors, as users could then control different sensors at distinct times during the drone's flight.

Finally, Flail would require flight tests to examine how well drone hardware is capable of following the described paths. There are a number of things that could affect how accurately a drone was able to follow the given trajectory, including: weather conditions, how stable the drone actually is during its flight, the conditions of its thrusters, and so on.

Flight studies would help to analyse the possible limitations of the hardware when attempting to follow precise directions. This would also help users to further understand the limitations of using Flail to describe pinpoint distances.

7.4 Conclusion

There are six main contributions in this thesis. The first is the implementation of a DSL which offers an abstraction of the default drone controller functions into a set of commands. The second, the implementation of a simulator using the Unity graphics engine, to test flight trajectories without requiring an expensive real drone. The third, the presentation of technical details on how to modify the hardware of a Cobra RC drone controller to accept communication from an Arduino board. The fourth, the implementation of a path generator based on parametric curves, by using the paradigm of first person piloting. The fifth, the interactive creation of paths in a laboratory using the Vive hardware. And finally, a test of Flail's portability by allowing Flail to run on a Ground Station.

Chapters 3 and 4 described in detail the implementation of Flail as well as the evaluation process used to test the language. It was shown that, although the simulator was capable of correctly interpreting the Flail file given to it, the RC Cobra microdrone had trouble flying stably. The difficulty of the later stemmed from the drone itself, as it possessed weak engines and a delicate frame, which prevented it from partaking in continuous flights.

Chapter 5 demonstrated the use of an interactive interface to create flight paths. The Vive wand was used as the gesture medium and allowed users to describe paths that could later be converted into Flail code. The simulator was shown to accurately interpret and execute the instructions given to it based off of the Vive generated path.

Finally, chapter 6 presented the addition of an APM Planner 2.0 output file type to Flail. Which allowed the language to create flight trajectories that could be flown via the Ground Station. Flail was shown to be accurately converted into a readable output, however, it presented some scaling issues as the routes were generally to large for a commercial to fly. This was solved by the application of a 2D affine transformation which allows users to control the scale of the routes, as well as where the generated route would pass through.

Flail is an alternate way to design path trajectories for vehicles. While this research emphasized path generation for drones, Flail could also be used as a way to create paths for vehicles in general. Since Flail can describe motion in both 2D and 3D, it is capable of being used by any vehicle capable of movement in such dimensions. The Flail translator could be modified to output files for RC cars, for example.

Nonetheless, Flail still requires more work in order to become a fully fledged method for trajectory design. This includes collision detection to prevent crashes, the development of a curve editor application in order to help users to better plan their trajectories, and of course, better flight/movement tests to analyze how well hardware can replicate the proposed paths.

Bibliography

[1] T. Bradshaw, "Drones are impressive but are not yet a comfortable hobby," *Financial Times*, Apr 2016. Also available as `https://www.ft.com/content/4376eed0-0008-11e6-99cb-83242733f755` (visited on 05/18/2019).

[2] "Drones Light Up The Sky," *Intel Corporation.* Also available as `https://www.intel.ca/content/www/ca/en/technology-innovation/aerial-technology-light-show.html` (visited on 05/18/2019).

[3] "Intel Drone Light Show," *Air Force Medical Service.* Also available as `https://www.airforcemedicine.af.mil/News/Photos/igphoto/2001941709/` (visited on 05/18/2019).

[4] F. Hermans, M. Pinzger, and A. V. Deursen, "Domain-specific languages in practice: A user study on the success factors," *Model Driven Engineering Languages and Systems Lecture Notes in Computer Science*, vol. 5795, p. 423–437, 2009. Also available as `https://link.springer.com/chapter/10.1007/978-3-642-04425-0_33` (visited on 05/29/2019).

[5] Pinciroli, Adam, Beltrame, and Giovanni, "Buzz: An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms," *arXiv.org*, Aug 2015. Also available as `https://arxiv.org/abs/1507.05946` (visited on 06/05/2019).

[6] "Learn to Code with a Drone," *Tynker Coding for Kids.* Also avallable as `https://www.tynker.com/learn-to-code/code-this-drone/` (visited on 06/05/2019).

[7] “MAVLink Developer Guide.” `https://mavlink.io/en/` (visited on 06/20/2019).

[8] “ArduPilot.” `http://ardupilot.org/about` (visited on 05/18/2019).

[9] “Example: Simple Go To (Copter) - DroneKit-Python’s documentation!,” *3D Robotics*. Also available as `https://dronekit.netlify.com/examples/simple_goto.html` (visited on 05/20/2019).

[10] “World geodetic system 1984 (WGS84).” `https://confluence.qps.nl/qinsy/en/world-geodetic-system-1984-wgs84-29855173.html`. (visited on 07/11/2019).

[11] J. P. de Graaff, “Programming a Parrot AR.Drone 2.0 with Python - The Easy Way,” *PS-Drone*. Also available as `http://www.playsheep.de/drone/index.html` (visited on 06/22/2019).

[12] Manas, “AR.Drone Java API.” `https://github.com/manastech/javadrone` (visited on 05/20/2019), Dec 2014.

[13] “Autonomous Navigation Framework for Quadcopter,” *ROS.org*. Also available as `http://wiki.ros.org/Autonomous%20Navigation%20Framework%20for%20Quadcopter` (visited on 06/01/2019).

[14] L. Zhang, R. Merrifield, A. Deguet, and G.-Z. Yang, “Powering the world’s robots,” *Science Robotics*, vol. 2, no. 11, 2017. Also available as `https://robotics.sciencemag.org/content/2/11/eaar1868.full` (visited on 06/15/2019).

[15] F. Geisendörfer, “A node.js client for controlling Parrot AR Drone 2.0 quad-copters.” `https://github.com/felixge/node-ar-drone`. (visited on 05/21/2019).

[16] “Computational Human-Robot Interaction,”

[17] “Human-Drone Interaction: state of the art, open issues and challenges,”

[18] “Improving Collocated Robot Teleoperation with Augmented Reality,”

[19] "Designing Robots with Movement in Mind,"

[20] "Communicating Affect via Flight Path,"

[21] "Collocated Human-Drone Interaction: Methodology and Approach Strategy,"

[22]

[23] A. Billard and D. Grollman, "Robot learning by demonstration," 2013. Also available as `http://www.scholarpedia.org/article/Robot_learning_by_demonstration` (visited on 05/25/2019).

[24] B. Siciliano and O. Khatib, *Springer Handbook of Robotics: Bruno Siciliano*. Springer-Verlag Berlin Heidelberg. Also available as `https://www.springer.com/us/book/9783540303015` (visited on 05/25/2019).

[25] T. Lozano-Perez, "Robot programming," *Proceedings of the IEEE*, vol. 71, p. 821–841, Jul 1983.

[26] B. Dufay and J.-C. Latombe, "An Approach to Automatic Robot Programming Based on Inductive Learning," *The International Journal of Robotics Research*, vol. 3, p. 3–20, Dec 1984.

[27] C.-H. Chu, Y.-W. Liu, P.-C. Li, L.-C. Huang, and Y.-P. Luh, "Programming by Demonstration in Augmented Reality for the Motion Planning of a Three-Axis CNC Dispenser," *International Journal of Precision Engineering and Manufacturing-Green Technology*, 2019. (visited on 05/25/2019).

[28] A. Gaschler, M. Springer, M. Rickert, and A. Knoll, "Intuitive robot tasks with augmented reality and virtual obstacles," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Jun 2014. Also available as `https://www.researchgate.net/publication/285054127_Intuitive_Robot_Tasks_with_Augmented_Reality_and_Virtual_Obstacles` (visited on 05/27/2019).

[29] T. Lau, “Programming by demonstration: a machine learning approach,” *Computer science and engineering [209]*, 2001. Also available as http://hdl.handle.net/1773/6949 (visited on 05/25/2019).

[30] O. Erat, W. A. Isop, D. Kalkofen, and D. Schmalstieg, “Drone-Augmented Human Vision: Exocentric Control for Drones Exploring Hidden Areas,” *IEEE Transactions on Visualization and Computer Graphics*, april 2018. Also available as https://ieeexplore.ieee.org/abstract/document/8260942(visited on 05/21/2019).

[31] Cho, Kwangsu, Cho, Jeon, and Jongwoo, “Fly a Drone Safely: Evaluation of an Embodied Egocentric Drone Controller Interface,” *OUP Academic*, Sep 2016. Also available as https://academic.oup.com/iwc/article/29/3/345/2607841 (visited on 05/17/2019).

[32] N. Li, S. Cartwright, A. S. Nittala, E. Sharlin, and M. C. Sousa, “A Spatial Interface for Enhancing Human-UAV Awareness,” *Flying Frustum*, 2015. Also available as https://dl.acm.org/citation.cfm?id=2814956 (visited on 05/23/2019).

[33] A. Imdoukh, A. Shaker, A. Al-Toukhy, D. Kablaoui, and M. El-Abd, “Semi-autonomous indoor firefighting UAV,” *Semi-autonomous indoor firefighting UAV - IEEE Conference Publication*. Also available as http://ieeexplore.ieee.org/abstract/document/8023625 (visited on 05/27/2019).

[34] “What is Logo,” *LOGO FOUNDATION*. Also available as https://el.media.mit.edu/logo-foundation/what_is_logo/history.html (visited on 05/25/2019).

[35] Y. Li, Y. Li, Q. Hu, W. Wang, B. Xie, and X. Yu, “Sloshing resistance and gas–liquid distribution performance in the entrance of LNG plate-fin heat exchangers,” *Applied Thermal Engineering*, vol. 82, p. 182–193, 2015.

[36] H.-D. Boecker, “Interactive problem solving using Logo,” 1987. Also available as https://telearn.archives-ouvertes.fr/hal-00190546 (visited on 06/25/2019).

[37] "Control Applications (Turtle Graphics)," *IGCSE ICT - Turtle Graphics.* Also available as `https://www.ictlounge.com/html/control_applications_turtle_graphics.htm` (visited on 06/10/2019).

[38] G. Pate, "LOGO FAQ," 12 1995. Also available as `https://web.archive.org/web/20090310020335/http://www.erzwiss.uni-hamburg.de/Sonstiges/Logo/logofaqx.htm#FAQ2` (visited on 06/10/2019).

[39] "Binary scaling." `https://enacademic.com/dic.nsf/enwiki/3724299`. (visited on 06/7/2019).

[40] P. A. LaPlante, *Real Time Systems Design And Analysis*, ch. 7.5.3. - Binary Angular Measure. 1992.

[41] R. M. Maynard, *Electronics Technician Volume 06 - Digital Data Systems*, p. 295. 1997.

[42] G. Strang and E. Herman, *Calculus*. OpenStax, Rice University, 2016.

[43] P. A. LaPlante, *Software Engineering for Image Processing Systems*, pp. 154–155. CRC Press, first ed., 2003.

[44] "What is the recommended space for the play area?," *VIVE.* Also available as `https://www.vive.com/eu/support/vive/category_howto/what-is-the-recommended-space-for-play-area.html` (visited on 06/27/2019).

[45] "Unity Manual," *Unity Technologies*, 2019. Also available as `https://docs.unity3d.com/Manual/index.html` (visited on 06/27/2019).

[46] J.-W. Jung, B.-C. So, J.-G. Kang, D.-W. Lim, and Y. Son, "Expanded Douglas–Peucker Polygonal Approximation and Opposite Angle-Based Exact Cell Decomposition for Path Planning with Curvilinear Obstacles," *Applied Sciences*, vol. 9, no. 4, p. 638, 2019.

[47] J. Li and S. Chen, "The Cubic α-Catmull-Rom Spline," *Mathematical and Computational Applications*, vol. 21, no. 3, p. 33, 2016.

[48] "A Class of Local Interpolating Splines," *ScienceDirect*, Jun 2014. Also available as `https://www.sciencedirect.com/science/article/pii/B9780120790500500205` (visited on 07/5/2019).

[49] P. Lucidarme, "Catmull-Rom splines," *Robotics, Teaching & Learning*, Jan 2014. Also available as `https://www.lucidarme.me/catmull-rom-splines/` (visited on 07/3/2019).

[50] L. M. Surhone, M. T. Timpledon, and S. F. Marseken, *Root Mean Square Deviation.* Betascript Publishing, 6 2015.

[51] "Choosing a Ground Station," *ArduPilot.* Also available as `http://ardupilot.org/copter/docs/common-choosing-a-ground-station.html` (visited on 07/11/2019).

[52] "Mission Planner Home," *ArduPilot.* Also available as `http://ardupilot.org/planner/` (visited on 07/13/2019).

[53] "APM Planner 2 Home," *ArduPilot.* Also available as `http://ardupilot.org/planner2/` (visited on 07/13/2019).

[54] "MAVProxy: A UAV ground station software package for MAVLink based systems," *ArduPilot.* Also available as `http://ardupilot.github.io/MAVProxy/` (visited on 06/29/2019).

[55] "QGC - QGroundControl - Drone Control." `http://qgroundcontrol.com/`. (visited on 07/11/2019).

[56] "Achieve more with UgCS," *UgCS.* Also available as `https://www.ugcs.com/` (visited on 07/15/2019).

[57] "Configuring a Telemetry Radio using Mission Planner," *ArduPilot.* Also available as `http://ardupilot.org/copter/docs/`

`common-configuring-a-telemetry-radio-using-mission-planner.html` (visited on 07/23/2019).