

**Swarm Drones**

**Critical Design Review**

Spring 2019

**To:** Dr. Anthony Choi and Dr. Hodge Jenkins

**Team Members:**

Roy Wood

Ethan Daily

Daniel Minch

Table of Contents

[Table of Contents 2](#_Toc5620684)

[List of Tables 4](#_Toc5620685)

[List of Figures 4](#_Toc5620686)

[Glossary 5](#_Toc5620687)

[Executive Summary 6](#_Toc5620688)

[1. Introduction 7](#_Toc5620689)

[1.1 – Background 7](#_Toc5620690)

[1.2 – Project Description 7](#_Toc5620691)

[1.3 – Project Deliverables/Motivations 8](#_Toc5620692)

[2. Final Design Characterization 8](#_Toc5620693)

[2.1 – Execution of Project Phases 8](#_Toc5620694)

[2.2 – Successes, Failures, and Revisions 10](#_Toc5620695)

[2.3 – Final Design Specifications 11](#_Toc5620696)

[Firmware & Software 12](#_Toc5620697)

[Computers & Other Digital Controllers 12](#_Toc5620698)

[Power Supplies 13](#_Toc5620699)

[Mechanical Parts 13](#_Toc5620700)

[3. Tests and Checks Performed 15](#_Toc5620701)

[3.1 – Equipment Checks 15](#_Toc5620702)

[3.2 – System Tests 15](#_Toc5620703)

[4. Results and Discussion 16](#_Toc5620704)

[4.1 – Test Results 16](#_Toc5620705)

[Maximum Velocity (Nominal) 16](#_Toc5620706)

[Stimulus Response – Boot Time 17](#_Toc5620707)

[Path Coherence, Arrival Time Discrepancy, and Swarm Tightness 18](#_Toc5620708)

[4.2 – Changes to Preliminary Design 20](#_Toc5620709)

[4.3 – Revised Budget 20](#_Toc5620710)

[5. Discussion of Safety/Sustainability 21](#_Toc5620711)

[5.1 – Operational Safety 21](#_Toc5620712)

[5.2 – Considerations for Sustainability 21](#_Toc5620713)

[Motor Life 21](#_Toc5620714)

[Environmental Exposure 22](#_Toc5620715)

[Creep 22](#_Toc5620716)

[6. Conclusions and Critiques 22](#_Toc5620717)

[7. Recommendations 23](#_Toc5620718)

[Appendix A 24](#_Toc5620719)

[Components Received from Previous Project 24](#_Toc5620720)

[Appendix B 25](#_Toc5620721)

[Circuit Diagram 25](#_Toc5620722)

[Software Diagram 27](#_Toc5620723)

[Appendix C 27](#_Toc5620724)

[Matlab Source Code 27](#_Toc5620725)

[Appendix E 31](#_Toc5620726)

[Drawings 31](#_Toc5620727)

[Appendix F 34](#_Toc5620728)

[Equipment Inventory 34](#_Toc5620729)

[Nominal Battery Voltage 34](#_Toc5620730)

[Motor Function 34](#_Toc5620731)

[Radio Telemetry Function 35](#_Toc5620732)

[RaspberryPi and Pixhawk Compliance 35](#_Toc5620733)

[Command Computer Connection 35](#_Toc5620734)

[RaspberryPi Connection 35](#_Toc5620735)

[Appendix G 35](#_Toc5620736)

[Appendix H 36](#_Toc5620737)

List of Tables

[Table 1.1 **-** Design Requirements 7](#_Toc5620831)

[Table 1.2 - Deliverables of the Swarm Drones Project 8](#_Toc5620832)

[Table 2.1 - Firmware and Software Components 12](#_Toc5620833)

[Table 2.2 - Digital Components of Rovers and Command Station 13](#_Toc5620834)

[Table 2.3 - Mechanical and Electrical Power Supplies 13](#_Toc5620835)

[Table 2.4 - Mechanical Components of Rover Assembly 13](#_Toc5620836)

[Table 3.1 – List of Equipment Checks 15](#_Toc5620837)

[Table 3.2 – List of Tests and measured Parameters 16](#_Toc5620838)

[Table 4.2 – Budget Breakdown per Rover Unit 20](#_Toc5620839)

[Table 5.1 – Safety Recommendations 21](#_Toc5620840)

List of Figures

[Figure 2.1 – Radio controller rover for phase one testing 9](#_Toc5620822)

[Figure 2.2 – Fully equipped rover for autonomous testing 10](#_Toc5620823)

[Figure 2.3 – Final Rover Design 14](#_Toc5620824)

[Figure 2.4 – Final Rover Design (side view) 14](#_Toc5620825)

[Figure 4.1 – Travel vs. Time w/ Full Throttle 17](#_Toc5620826)

[Figure 4.2 - Time Study of System Initialization 17](#_Toc5620827)

[Figure 4.3 – Rover’s Path of Travel for Mission 3 18](#_Toc5620828)

[Figure 4.4 – Rover’s Path of Travel for Mission 1 19](#_Toc5620829)

[Figure 4.5 – Paths of Two Rovers on a Swarm Mission 19](#_Toc5620830)

# 

# Glossary

**Firmware** - Permanent software programmed into the hardware’s read-only memory

**Robot** - Autonomous Unit capable of a complex series of actions

**Drone** - UAV (Unmanned Aerial Vehicle)

**Rover** - UTV (Unmanned Terrestrial Vehicle)

**Apsync** - A distribution of the raspberry-pi operating system Rasbian that comes with many features for autonomous drone control

**MavLink** - a protocol used for communicating with small unmanned vehicles used to transmit information.

**ArduPilot** - An open source firmware developed for flight controllers to natively pilot planes, copters, rovers, and submarines

**Socket** - A Python package that allows for the transfer of data over networks

**Backend** - The Python script that runs on the command computer that connects all of the RaspberryPi’s together along with the map client

**Frontend** - The Python script that runs on the raspberry-pi that sends and receives data from the pixhawk and Backend and stores commands for the rover to execute.

**CAD** - computer aided design

**Python** - A high-level all purpose programming language

**UDP** - User Datagram Protocol, a network level protocol that uses a best-effort method. Does not require a handshake to establish a connection

**MatLab** - A software built by MathWorks to create a numerical computing environment implemented with scripts

**.csv file** - A delimited text file that sorts information into columns.

**Telemetry** - Automated wireless communication process that relies on specific sending and receiving equipement

**Motor controller -** A cricuit board that acts as an intermediary between the flight controller and the motors. It interprets a signal from the flight controller into signals that drive the motors.  
**Flight controller -** A circuit board developed to control motors and read sensors relating to arial and ground vehicles.

Executive Summary

The client, Dr. Anthony Choi, Assistant Professor at Mercer University’s School of Engineering, Department of Electrical and Computer Engineering, had shown interest in autonomous swarm robots to expand the university’s collection. Swarm robots are able to travel in unison and communicate their location to one another. Dr. Choi uses the robots to promote Mercer University’s School of Engineering during events and special occasions.

R.E.D. Design has received this project from Dr. Choi via the Mercer School of Engineering. R.E.D. Design has appointed Roy Wood, Ethan Daily, and Daniel Minch, a group of engineers specializing in fabrication, computer science, and mechanical design, to further develop the project to new and further ground.

R.E.D. Design has created a new swarm system that utilizes ground rovers using companion computers and wireless networking. This configuration results in the instant transfer of commands while also allowing heavy computations to be made quickly by both the command and the companion computers due to the breakdown of workload. Connecting multiple drones is trivial due to each drone only requiring the ability to connect to a wireless access point to become part of the swarm. By utilizing network protocol and having a central server connected to an access point, R.E.D. Design hopes to pave the way for an Internet of Things swarm system. Connection to any number of drones at once would be possible by using dynamically assigned identification of each rover with address resolution. The rovers will be intelligent as each rover is equipped with a companion computer that is constantly communicating with the other rovers and will be able to adjust its course based on the information available. R.E.D. Design was able to conduct multiple missions using the system. The results of the missions are given in the next paragraph.

During initial testing, the nominal max velocity was calculated to be 1.113 ft/s. The stimulus response with boot time was measured to be 64.33 seconds. The path stability of Mission 1 was 1.7353, which means the rover travels along a path that is about 1.73 times longer than the straight-line distance. The arrival time discrepancy was 236.5416 seconds, which can be described as the amount of time the rover actually took to get to the location minus the expected amount of time to get to the location. The path stability of Mission 3 was 1.2056, which is about 120% of the actual distance to the location. The arrival time discrepancy was 94.1668 seconds. The Swarm Mission contained two paths. The first path had a path stability of 1.3203, which is about 132% of the actual distance to the location. The arrival time discrepancy of the first path was 148.26 seconds. The second path had a path stability of 2.1344, which is about 213% of the actual distance to the location. The average swarm tightness of the Swarm Mission was calculated to be 18.4 meters, relative to the initial swarm tightness at 8 meters. All missions were successful.

Most importantly to improve the performance of the system, the GPS satellite navigation must be reevaluated. R.E.D. Design recommends the integration of Real Time Kinematic (RTK) satellite navigation. This would improve the resolution of the vehicles’ GPS location down to the centimeter level. Additionally, by utilizing network protocol and having a central server connected to an access point, R.E.D. Design hopes to pave the way for an Internet of Things swarm system. Connection to any number of drones at once would be possible by using dynamically assigned identification of each rover with address resolution.

# Introduction

## 1.1 – Background

Robots are becoming more prevalent in everyday life for a growing percentage of today’s society. Juxtaposed to the simple, repetitive tasks for which robots were initially employed, robots are now able to fabricate complex, convoluted designs with precision to the thousandth of an inch thanks to manufacturing processes such as Computer Numerical Control Additive/Subtractive Manufacturing. The use of robotics has increased due to its ability to simplify industrial processes; most notably, Amazon, the largest online-retailer corporation, has employed drones to improve their delivery services.

The client, Dr. Anthony Choi, Assistant Professor at Mercer University’s School of Engineering, Department of Electrical and Computer Engineering, had shown interest in autonomous swarm robots to expand the university’s collection. Swarm robots are able to travel in unison and communicate their location to one another. The current collection of robots consists of drones, rovers, and autonomous underwater vehicles. Dr. Choi uses the robots to promote Mercer University’s School of Engineering during events and on special occasions.

The team continued the work of a senior design team from 2016. The previous team’s vehicle served as a starting point for the development of the autonomous vehicles. The vehicles constructed by the previous team did not operate; each lacked some components as they had been recycled for different projects. Hence, the team salvaged all components for the implementation of a new design. The previous team recommended re-examining potential flight controllers. A list of the components used from the previous senior design team can be found in Appendix A [1].

## 1.2 – Project Description

R.E.D. Design, a team of undergraduate engineering students, has been tasked by Dr. Anthony Choi of Mercer University to design, fabricate, and program autonomous vehicles that exhibit swarm behavior using Global Positioning Systems. Dr. Choi requires that the project satisfies the following requirements seen in Table 1.1.

Table 1.1 **-** Design Requirements

|  |  |
| --- | --- |
| **Specification** | **Description** |
| Mobility | Capable of traversing a field with minor obstacles |
| GPS | Utilizes GPS to locate target and calculate route |
| Autonomous | Executable with only a user-designated location |
| Swarm Size | Minimum of three (3) units per swarm |
| Mission Range | Minimum mission range of 200 yards |

## 1.3 – Project Deliverables/Motivations

The project will deliver all of the items seen in Table 1.2.

Table 1.2 - Deliverables of the Swarm Drones Project

|  |  |  |
| --- | --- | --- |
| **Deliverable** | **Description** | **Quantity** |
| Autonomous Vehicle | Equipped with microcontroller, GPS sensor, communications array, and movement control modules | 3 |
| Command Computer | A portable computer programmed with software to control the swarm | 1 |

# Final Design Characterization

The following section will describe the equipment and methods used to achieve the final design of the Swarm System. More detailed diagrams and specifications may be found in the drawings in Appendix B and Appendix C respectively.

## – Execution of Project Phases

The final design, in compliance with the project’s preliminary design review, was produced through the execution of four phases. In short, these phases were established with consideration of the overall technological hierarchy of the swarm system, which will be discussed in the course of Section 2. The four phases are as follows:

Phase 1: Build Rover for Manual Control

Phase 2: Implement Autonomous Control

Phase 3: Implement an Autonomous Team of Vehicles

Phase 4: Execute Swarm Missions

To begin Phase 1, one of the rovers received from the previous project was stripped of all auxiliary electronics, leaving only the chassis, motors, and stock wiring. The leads to each motor were removed from the chassis’ central terminal so that each could be examined individually. Each motor was then supplied 6 volts to check for functionality; greater details of this check will be discussed later. Upon verifying the functionality of all six motors, each was rewired to the chassis’ terminal; the terminal was wired to the left and right output terminals of the Sabertooth motor controller. To provide manual control, a two-channel radio receiver was wired to the Sabertooth’s input terminals. Throttle was then applied using the two-channel radio transceiver paired with the receiver. At this point, the rover could only be controlled to travel forward/backward; however, this milestone opened the opportunity basic dynamic testing, concluding Phase 1. The rover, as of Phase 1, can be seen in Figure 2.1.

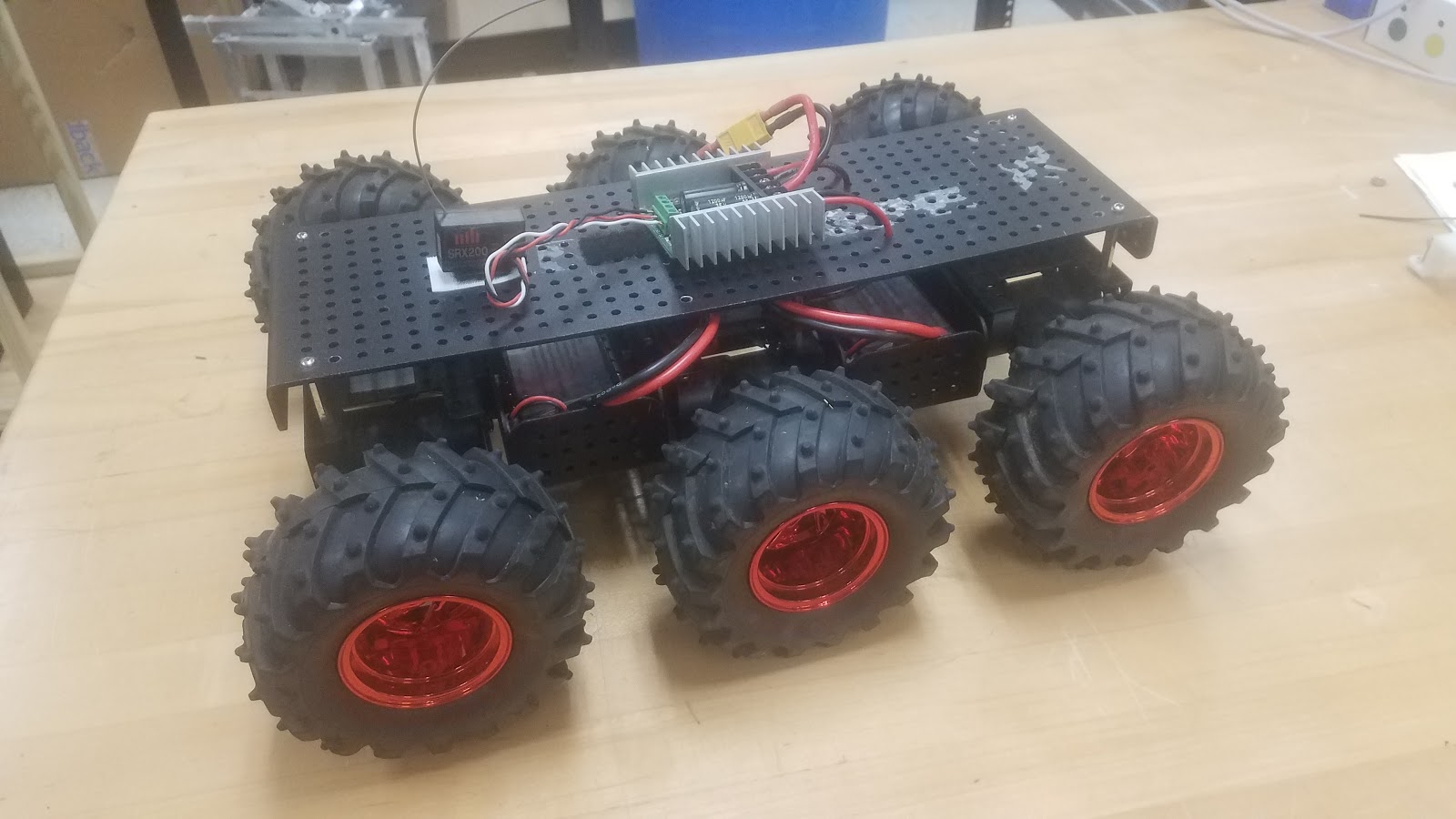


Figure 2.1 – Radio controller rover for phase one testing

Phase 2 began with the fine-tuning of the Pixhawk flight controller. The firmware settings from the previous project were wiped from the Pixhawk’s memory, allowing for the installation of updated ArduPilot firmware. The new firmware was then modified specifically to the hardware in use. The RaspberryPi was setup using apsync, a distribution of Rasbian. Wireless Network Card drivers and developed Python scripts were added to the operating system. The backend software was installed onto the command computer and tested.

Before attempting a mission with live GPS updates, the team opted to run simple, hard-coded commands from the front end on the RaspberryPi. For example, one of these tests commanded the rover to travel north for 10 seconds, turn around, and then travel south for 10 seconds. These hard-coded commands proved successful in that they demonstrated the ability of a rover to act autonomously; however, the team observed high error in the rover’s sense of direction.

Commands from the command computer were then executed to test the response time, accuracy, and reach of the signal of the system. A test consisted of marking a home location that the rover started in and giving it a command to move close to an observable landmark on the map software. Other commands would be added in after the rover reached its first destination at rapid intervals to test the response time of the rover to a change in the commands. The tests proved that the rovers could be driven autonomously with the command computer using the map client; however, the team observed a high error in GPS coordinate readings between the map client and the companion computer readings. The rover from Phase 2 can be seen in Figure 2.2, below.



Figure 2.2 – Fully equipped rover for autonomous testing

Phase 3 is essentially a repeated procedure of the first two phases combined. The preparation of the additional rovers followed the same construction, equipment checks, and connection checks as Phase 1 and 2. The procedure that was unique to Phase 3 was threading the back-to-front communication between the command computer and RaspberryPi. Threading allowed the command computer to address backend commands to the rovers as individuals.

Phase 4 was the team’s assessment of the system as a whole, assessing the capability of multiple rovers to travel coercively to swarm a designated GPS location. Ultimately, the team was unable to implement swarm behaviors and did not complete Phase 4 of the project; however, using front-end records of the rovers’ paths, the team was able to evaluate all of their desired parameters.

## – Successes, Failures, and Revisions

The first failure occurred early in Phase 1 in the form of malfunctioning DC motors. Preliminary equipment checks revealed that certain motors were spinning much slower than others, some not spinning at all. Further investigation of the problem suggested that 4 of the 18 motors had bad brushes. Also, by dissecting the gearboxes of the seized motors, the team discovered that two motors contained pinions with broken teeth. At this point, there was little to be done other than to replace what was broken. A set of six functioning motors were pooled to assemble a single chassis; ten new motors were purchased to compensate for the six that had failed as well as to provide spares.

The remainder of Phase 1 was successful; radio control was achieved for one rover and the team began to conduct dynamic tests. The early stages of Phase 2 were also successful. Installing the firmware to the flight controller and achieving compliance between flight controller and companion computer proved to be relatively straight-forward. Also, as mentioned the team ran simple, hard-coded commands from the front end. At this point, Phase 2 came to an abrupt halt.

The team dealt with many small problems when first developing the networking aspects of the frontend and the backend. One problem was the proper forwarding of messages to the command computer from the RaspberryPi. This was resolved by confirming the host IP address and ports as well as adding port forwarding within the configuration of the wireless router. Another failure within the network was the improper closing of connections and ports which left a process running in the background of the command computer. These processes would block network traffic and disable the backend software upon boot of the backend.

Phases 1 and 2 were accomplished by a RaspberryPi-compatible USB Wi-Fi adapter that was donated to the project by a team member; however, the team accidentally purchased three USB Wi-Fi adapters that were not inherently compatible with RaspberryPi, although thought to be when purchased. This proved to be a rabbit hole of Linux kernels, headers, drivers, makefile, dkms, all of it. Ultimately after days of searching, the team came across a package that was able to identify the chipset of the Wi-Fi adapter and provide the necessary software to utilize the adapter with the team’s RaspberryPi.

Phase 3 was conducted to produce a second autonomous rover, but not a third. When preparing to install the front-end program on the final RaspberryPi, the micro SD port was broken. This failure occurred in close proximity to the final phase of testing; therefore, the team was unable to test three rovers, simultaneously. However, the team did test two vehicles to assess the capability of the back-end to command multiple units. Also, the third rover was tested with a functioning RaspberryPi to verify its autonomous capability.

The two-vehicle test was successful. Both rovers were able to flawlessly connect to the server, receive data, and send data. The back-end was able to handle two clients and successfully threaded both individually. The only problem with the test came the actual movement of the rovers. One rover was able to follow fairly close to a straight line to the target location; conversely, the other rover was unable to find the correct direction, intermittently changing course. Overall, both rovers were able to come close to the correct bearing by the end of the test.

## 2.3 – Final Design Specifications

The system is comprised of a command station and three rover assemblies. The command station is comprised of a laptop computer and a wireless router. Each rover contains a wide variety of components, ranging from mechanical housings to digital control systems. This section is dedicating to the characterization of the contribution of each system component, be it on or off the rover. The computer runs a Python program running a server and map client. This program, referred to as the “backend,” communicates to each ground unit with a Python User Datagram Protocol (UDP) server via Wi-Fi. Backend commands are received by each ground unit with an onboard RaspberryPi computer. Each RaspberryPi runs a Python program that contains the actual mission commands that act with the input of the rover’s GPS location. The following sections contain the components and component description for all parts used in the system.

### Firmware & Software

The following table contains the firmware components and the software components that are used in the system.

Table 2.1 - Firmware and Software Components

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Firmware/Software** |
| ArduPilot | An open source drone firmware built for autonomous flight | Firmware |
| APSync | A distribution of Rasbian that comes with many tools for autonomous flight | Software |
| Wireless Driver Install | A bash script that searches and installs wireless drivers on RaspberryPis | Software |
| Boot Code | A modified RaspberryPi boot script that ensures vital software gets initialized at boot | Software |
| Front End | The Python script that drives the RaspberryPi that sends and receives data to and from the command computer and flight controller | Software |
| Back End | The Python script that drives the command computer that sends and receives data to and from the RaspberryPis | Software |

### Computers & Other Digital Controllers

Table 2.2 lists the digital components that are used by the system.

Table 2.2 - Digital Components of Rovers and Command Station

|  |  |  |
| --- | --- | --- |
| **Device** | **Description** | **Quantity** |
| Sabertooth 2x25 | 2 channel motor controller providing differential drive with input signals from the Pixhawk flight controller | 3 |
| Pixhawk | central control system of each rover directing travel with feedback from the GPS module | 3 |
| RaspberryPi 2B | on-board companion computer communicating the back-end commands to the flight controller | 3 |
| uBlox 3DR | onboard GPS module providing position updates to the flight controller | 3 |
| Linksys WRT1900ACS | command station Wi-Fi router broadcasting back-end commands to the RaspberryPi | 1 |
| Dell Precision M4800 | Central command station computer (laptop) both communicating GPS updates and sending flight commands through a python server to the RaspberryPi companion | 1 |

### Power Supplies

The following table contains both the supplier of electrical (internal energy) and mechanical (kinetic energy) power.

Table 2.3 - Mechanical and Electrical Power Supplies

|  |  |  |
| --- | --- | --- |
| **Device** | **Description** | **Quantity** |
| Pololu DC Gearmotor | 6V brushed DC motors combined with 75:1 metal spur gearbox providing power to the rovers’ wheels via a 4mm driveshaft | 18 |
| Yowoo 5200 | 2 cell 7.4 V lithium polymer batteries connected in parallel to power the motors as well as all digital devices onboard the rover | 6 |

### Mechanical Parts

Table 2.4 explores the mechanical components that are used in the system.

Table 2.4 - Mechanical Components of Rover Assembly

|  |  |  |
| --- | --- | --- |
| **Item** | **Description** | **Quantity** |
| Wild Thumper 6WD | The chassis on which the rovers are built; composed of aluminum body panels, 6 motor housings, 6 wheels w/ rubber tires, and torsion spring suspension | 3 |
| Auxiliary Housings | 3D printed PLA fixtures that aid in mounting electrical/digital devices to the rover chassis | 6 |
| Fastening | 304 stainless steel M4 hex bolts and locking nuts securing auxiliary fixtures to rover chassis | 36 |

The summation of all of these components into a single assembly can be seen below in Figures 2.3 and 2.4.



Figure 2.3 – Final Rover Design



Figure 2.4 – Final Rover Design (side view)

# Tests and Checks Performed

## – Equipment Checks

The following checks were conducted to assess the functionality of several system components. Table 3.1 lists purely the assessment of each check; greater details of each may be found in Appendix E.

Table 3.1 – List of Equipment Checks

|  |  |
| --- | --- |
| **Check** | **Assessment (yes/no)** |
| Equipment Inventory | All necessary equipment is attained for rover assembly and mission execution |
| Nominal Battery Voltage | All batteries measure the nominal voltage when charged |
| Motor Function | Each motor spins provided power |
| Radio Telemetry Function | Throttle applied by radio transceiver runs motors |
| RaspberryPi / Pixhawk Compliance | Flight controller receives and acknowledges MAVLink protocols from companion computer and companion computer reads flight controller attributes |
| Command Computer Connection | Command computer sends/receives packets to/from the router after using a ping command |
| RaspberryPi  Connection | Companion computer sends/receives packets to/from the router and command computer after using a ping command |

Note: Two additional checks were added in order to better clarify the compliance and operation of the system. Wi-Fi Connection Check was added in order to verify that the Wi-Fi is connected and working properly. Server Connection Check was also added to verify the compliance of the client and server.

## – System Tests

The following tests were conducted to assess, modify, and validate the performance of the system. Table 3.2, below, lists only the assessment of each test. The results of the tests, and their analysis, can be found in the next section, *Results and Discussion*.

Table 3.2 – List of Tests and measured Parameters

|  |  |
| --- | --- |
| **Test** | **Measured Parameter(s)** |
| Maximum Velocity | Time taken to travel specific intervals of distance at full throttle |
| Stimulus Response | Time for the RaspberryPi to respond to a user-provided stimulus (front-end interrupt) |
| Path Coherence | Actual distance traveled over the course of a mission as to be compared with an ideal value |
| Arrival Time Discrepancy | Total mission time as to be compared with an ideal value |
| Swarm Tightness | Average distance between multiple rovers at any given time during a mission |

Note: Both the Max Turning Radius Test and the Stopping Distance Test from the Testing Plan have been removed. The results of neither test have analogous parameters within the software to improve the drivability of the vehicle. The firmware of the Pixhawk directly modifies the automated turn radius of the vehicle.

# Results and Discussion

The results of each test are provided in the order in which they were conducted. Note that singular missions serve to provide data necessary for multiple evaluation criteria in the following section. Changes made to preliminary design and the costs associated are discussed thereafter.

## – Test Results

The following sections contain the results of the tests outline in Table 3.2, above.

### Maximum Velocity (Nominal)

The nominal maximum velocity of the rovers was calculated using the data seen in Figure 4.1 below. It should be noted that the nominal value is an average of the max velocity from all tests. The nominal value was chosen to better identify the expected max velocity during a real mission.

Figure 4.1 – Travel vs. Time w/ Full Throttle

The strong linear trend indicates that the rover reached its maximum velocity before the first lap time and maintained it for the course of the trial. As seen above, the average velocity of all trials came out to be 1.113 feet per second (0.759 miles per hour). With this result, the team able to normalize throttle settings in the front end using the vehicle.groundspeed dronekit command.

### Stimulus Response – Boot Time

The following test displays the amount of time that the RaspberryPi takes to boot the entire sequence of initializations, connections, and commands. The results of the test show that the expected stimulus response, with boot time, is 64.33 seconds.

Figure 4.2 - Time Study of System Initialization

Figure 4.2 reports the time taken by the RaspberryPi to fully initialize as well as time taken to connect to the vehicle.

### Path Coherence, Arrival Time Discrepancy, and Swarm Tightness

The parameters listed above may be categorized simultaneous, being that each were evaluated with the same method of data collection. This method utilized the front end’s main loop to write a CSV file of rovers’ GPS locations in time. CSV’s were then imported to MATLAB scripts containing the parameter calculations. Also, these files were imported to custom Google Maps to provide a plot of rovers’ travel.

The first assessment of a rover’s path of travel was conducted using simple go-to commands. In short, the rover was commanded to travel a discrete distance from its booted location. Below in Figure 4.3 is a plot of the rover’s location during the execution of the commands.

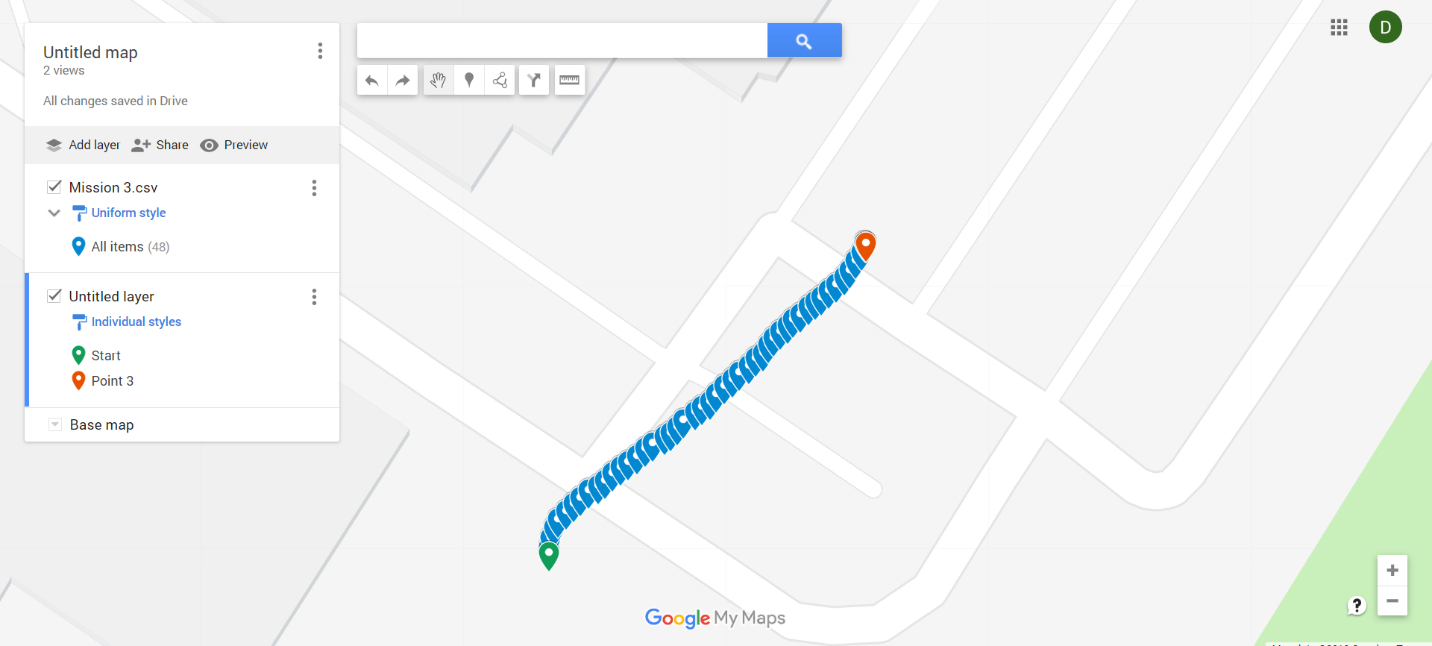


Figure 4.3 – Rover’s Path of Travel for Mission 3

The beginning and end of the test are represented by the green and red points respectively. The overlapping of points may be interpreted as the aspect of time. Overall, the path of the rover is well defined and very linear. There was very little over-travel; the path coherence was calculated to be 1.0256.

The next path evaluated, below in Figure 4.4, was one of an autonomous waypoint mission.

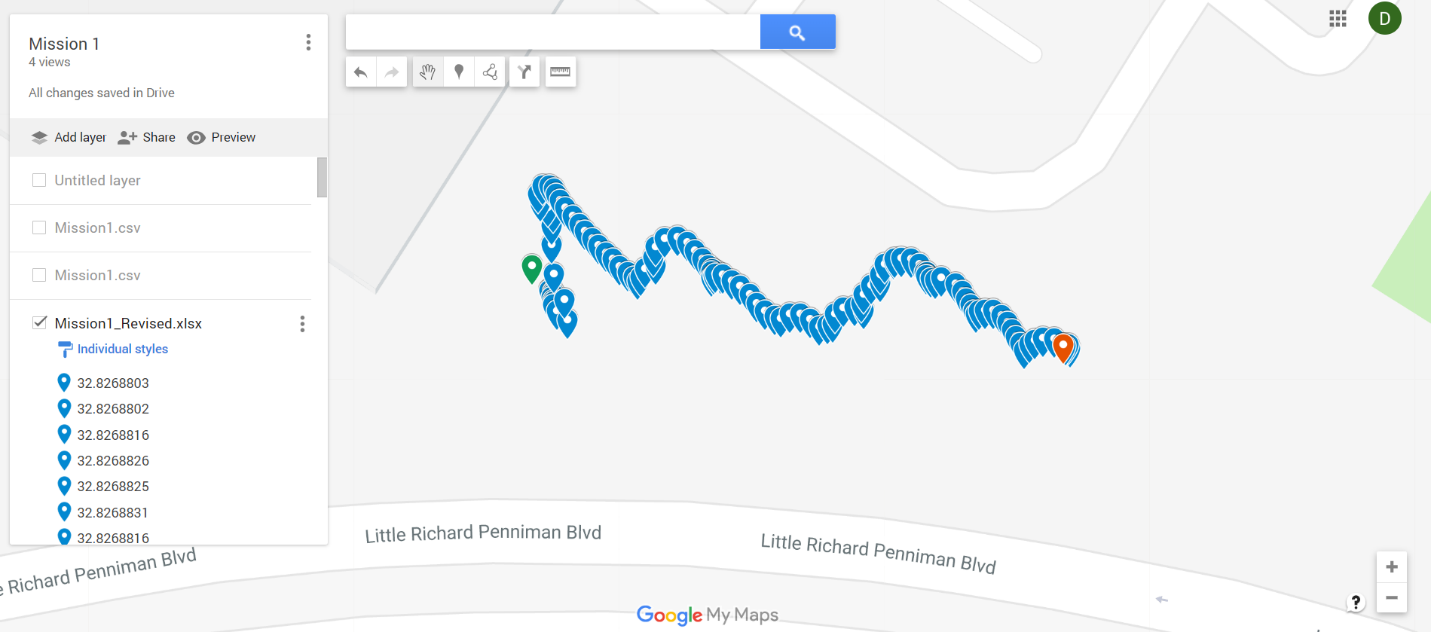


Figure 4.4 – Rover’s Path of Travel for Mission 1

The beginning and end of the test are represented by the green and red points respectively. The overlapping of points may be interpreted as the aspect of time. Areas of high point density represent occasions where the rover struggled to adjust its orientation. As it appears, the rover attempted to maintain its direction by making several corrections. This test proved that the rover was capable of tracking its location; however, the Path Stability was calculated to be 1.7353.

The next mission was the swarm mission with two vehicles. A map of the paths taken by each rover can be seen in Figure 4.5, below.

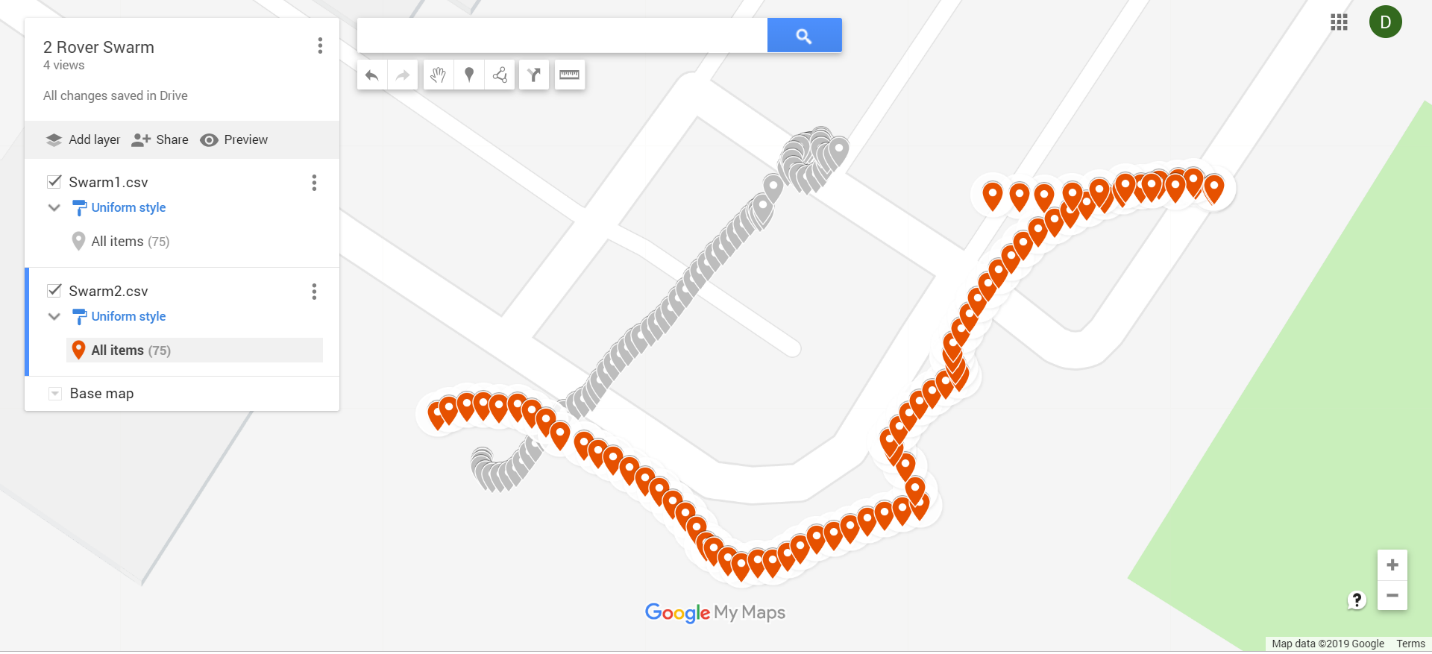


Figure 4.5 – Paths of Two Rovers on a Swarm Mission

The map of the mission shows that one rover was able to follow fairly close to a straight line to the target location; conversely, the other rover was unable to find the correct direction, intermittently changing course. The rover attempted to maintain its direction by making several corrections. Overall, the two-vehicle test was successful.

## – Changes to Preliminary Design

The changes to the team’s preliminary design were minimal. The first change was moving from a RaspberryPi 3 to a RaspberryPi 2B. The initial install of APSync failed to boot on a RaspberryPi 3 but succeeded on the 2B model. It was decided that a slight loss in processor time was acceptable in order to have a fully-compatible operating system on the RaspberryPi. Due to the change to a 2B, the team lost access to a built-in wireless network adapter and had to purchase external adaptors in order to connect wirelessly. The team also agreed that a dedicated wifi router would help boost the range as well as direct traffic between the server and clients.

## – Revised Budget

Table 4.2 – Budget Breakdown per Rover Unit

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **Quantity** | **Cost** | **Total** |
| Pixhawk 4 Microcontroller | 1 | $177.50 | $177.50 |
| APM 2.5 Power Module | 1 | $25.00 | $25.00 |
| 3DR uBlox GPS Module | 1 | $42.00 | $42.00 |
| Radio Telemetry | 1 | $23.00 | $23.00 |
| Raspberry Pi 2B | 1 | $40.00 | $35.00 |
| MicroSD 16GB | 1 | $6.28 | $6.28 |
| SaberTooth 2X25 | 1 | $124.99 | $124.99 |
| Lithium-Polymer Batteries | 2 | $32.89 | $65.78 |
| LiPo Battery Charger | 1 | $31.68 | $31.68 |
| Pixhawk Switch | 1 | $4.54 | $4.54 |
| 6-Wheel Drive Chassis (w/ 6 motors) | 1 | $444.95 | $444.95 |
| Wireless Network Adapters | 1 | $21.99 | $21.99 |
| Miscellaneous cables | 1 | $15.00 | $15.00 |
| **Total** |  |  | **$1017.71** |

Discussion of Safety/Sustainability

## 5.1 – Operational Safety

R.E.D. Design suggests the following safety recommendations for the operation of the system:

Table 5.1 – Safety Recommendations

|  |  |
| --- | --- |
| **Safety** | **Description** |
| Safety Switch | The rover comes equipped with a safety switch that is able to override all commands on the pixhawk and immobilize the vehicle. In case of erratic behaviors, turn off the safety switch. |
| Traffic/Surrounding Awareness | The rover is an exciting bit of engineering; however, one must always maintain spatial awareness. If in a parking lot, take extra precautions to avoid traffic. |
| Electrical Shock | The electrical system of the rover is mostly exposed. If any modifications are made to any connections or open electricals, turn off the power. |
| Li-Po Batteries | The dangers of Li-Po batteries are real. If not properly charged, the batteries have a potential to ignite, or worse, explode. The integrity of the battery must be checked constantly. If a Li-Po battery begins to expand, dispose of the battery properly immediately. |

## 5.2 – Considerations for Sustainability

With such an involved design, it is necessary to discuss the factors that affect the sustainability of the system as a whole. Outlined below are factors that weigh heavily on the approximate lifespan of the swarm system from this point on. The team made careful observations of various components as they were used continually. By extrapolating these observations back to the inception of this project, the team compiled approximations for the following matters.

### Motor Life

As mentioned in Section 2, the team experienced two modes of motor failure: bad brushes (presumed) and broken pinions. Brush failure was most likely due to contaminants, such as dirt and moisture, as well as a high number of discharge cycles. As for the pinions, it was observed that individual teeth had sheared off, resulting almost certainly from a high-magnitude, fully reversing load.

Erosion

Over the course of the project, the rovers were disassembled and reassembled on a daily basis. As consequent of the troubleshooting and debugging, the main chassis and auxiliary housings experienced noticeable wear. For example, swapping motors resulted in gouges along the inner rim of the plastic motor housings. Also, areas of the aluminum chassis eroded from the unfastening and refastening of body panels.

### Environmental Exposure

Rovers must have clear sky access to achieve a GPS lock, therefore it is assumed that all missions are to be outdoors. This being said, there are several areas of the rovers’ design that are vulnerable to damage from environmental exposure.

The two greatest threats to the rovers are moisture and ultraviolet (UV) radiation. Although much of the onboard equipment is encased to some degree, there are several exposed electrical connections. Most notable are the serial connections between the motor controller, flight controller, GPS module, and RaspberryPi atop the rover chassis. Water has an opportunity to contact these connections, be it from rain, tire spray, or, terrifyingly, complete submersion. Also, UV exposure has the opportunity to degrade the many plastics spanning the rover body.

### Creep

Creep in the rovers’ chassis was a frequent problem with assembly and disassembly. Most notably, the aluminum plates atop the rover chassis did not fit properly to its mounting posts. It is believed that improper equipment mounting from previous assemblies induced internal stresses on the chassis that manifested in steady deformation over the time of project inactivity.

Deformation in body panels is minimal; therefore, the team does not see creep causing the chassis to fail any time soon. However, the parts created using 3D printing technology are predicted to be highly influenced by creep. This combined with the UV wear on the parts is predicted to cause rapid deformation over time.

# Conclusions and Critiques

Dr. Anthony Choi has expressed his desire for the development of a system of GPS guided robots that swarm a designated location at the click of a button. The project, on-going in recent years, has seen the implementation of small ground rovers with user-inputted GPS waypoints. R.E.D. Design has received this project from Dr. Choi via the Mercer School of Engineering. R.E.D. Design has appointed Roy Wood, Ethan Daily, and Daniel Minch, a group of engineers specializing in programming, computer science, and mechanical design, to further develop the project to new and further ground.

R.E.D. Design has created a new swarm system that utilizes ground rovers using companion computers and wireless networking. This configuration results in the instant transfer of commands while also allowing heavy computations to be made quickly by both the command and the companion computers due to the breakdown of workload. Connecting multiple drones is trivial due to each drone only requiring the ability to connect to a wireless access point to become part of the swarm. By utilizing network protocol and having a central server connected to an access point, R.E.D. Design hopes to pave the way for an Internet of Things swarm system. Connection to any number of drones at once would be possible by using dynamically assigned identification of each rover with address resolution. The rovers will be intelligent as each rover is equipped with a companion computer that is constantly communicating with the other rovers and will be able to adjust its course based on the information available. R.E.D. Design was able to conduct multiple missions using the system. The results of the missions are given in the next paragraph.

During initial testing, the nominal max velocity was calculated to be 1.113 ft/s. The stimulus response with boot time was measured to be 64.33 seconds. The path stability of Mission 1 was 1.7353, which means the rover travels along a path that is about 1.73 times longer than the straight-line distance. The arrival time discrepancy was 236.5416 seconds, which can be described as the amount of time the rover actually took to get to the location minus the expected amount of time to get to the location. The path stability of Mission 3 was 1.2056, which is about 120% of the actual distance to the location. The arrival time discrepancy was 94.1668 seconds. The Swarm Mission contained two paths. The first path had a path stability of 1.3203, which is about 132% of the actual distance to the location. The arrival time discrepancy of the first path was 148.26 seconds. The second path had a path stability of 2.1344, which is about 213% of the actual distance to the location.

While the potential for this system is very large, it has its drawbacks. The system is prone to a small range due to the wireless adapters and the wireless router. The system also suffers from a slight mapping error which requires a greater breakdown of the google maps API in order to implement more accurately. The system also uses the Wild Thumper Chassis which was found to be unstable and required many changes to function acceptably.

# Recommendations

Through all of the time spent working on the swarm system, R.E.D. Design recommends an improvement of the rover chassis, specifically to be able to handle more complicated terrain and protect from moisture and debris. Furthermore, more equipment should be added as the chassis is robust enough to seat more gadgets, such as video or echolocation, and to supplement such gadgets with integrated commands to the system. The system could accept the addition of aerial drones with few modifications; aerial drones are recommended to nullify the complex ground terrain. R.E.D. Design also recommends the integration of a mobile hotspot to allow for drone access outside of the range of the router. This would also pave the way for an Internet of Things integrated swarm system. Most importantly to improve the performance of the system, the GPS satellite navigation must be reevaluated. R.E.D. Design recommends the integration of Real Time Kinematic (RTK) satellite navigation. This would improve the resolution of the vehicles’ GPS location down to the centimeter level.

# References

1. Alosaimi K., Bennett K., DeWitt W., (2017). *Swarm Robotics: Senior Design Critical Review*

2. Pamir (2013), *Wild Thumper 6WD Chassis* [SLDASM]. Retrieved from <https://grabcad.com/library/wild-thumper-6wd-chassis-1/details?folder_id=4281587>

3. *Dagu Wild Thumper 6WD All-Terrain Chassis, Black, 75:1.* Pololu Corporation (2019). Retrieved from <https://www.pololu.com/product/1563>

4. *Ardupilot / ardupilot.* GitHub Incorporated (2019). Retrieved from <https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py>

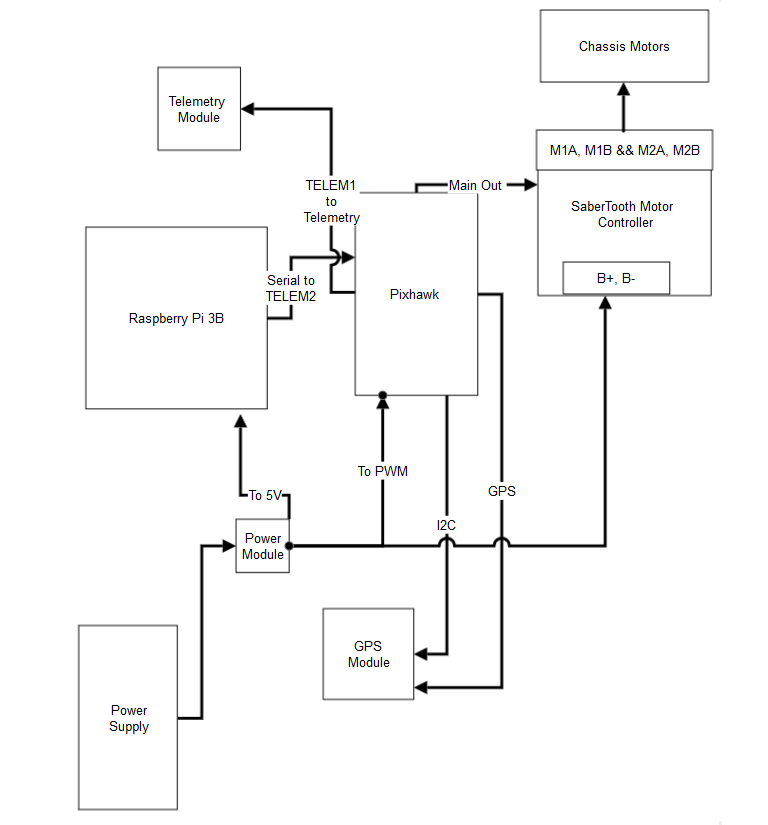
Appendix A

### Components Received from Previous Project

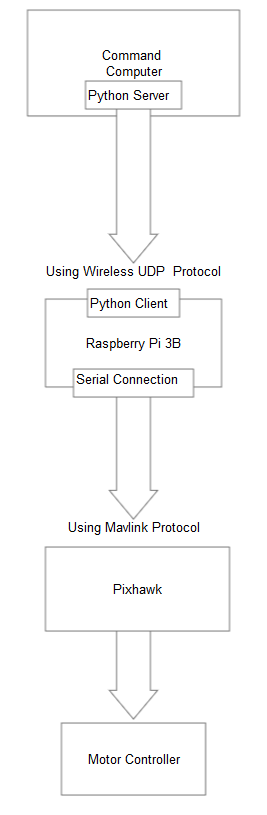
|  |
| --- |
| * Dagu Electronics “Wild Thumper” all-terrain chassis * Polulu brushed gearmotors (Supplied by Dagu Electronics) * Pixhawk flight controller * 3DR GPS module * SaberTooth motor controller * secondary telemetry |

Appendix B

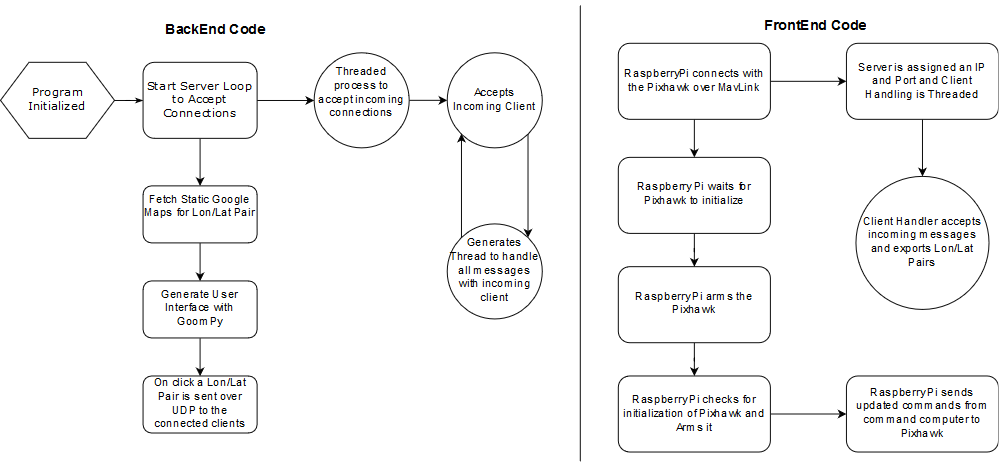
### Circuit Diagram



System Architecture



### Software Diagram



Appendix C

### Matlab Source Code

Function: advec.m

%import data and separate columns

M=dlmread('Mission1\_Revised.csv',',',1,0);

Ttot = M(end,3)-M(1,3);

lat = M(1:end,1);

long = M(1:end,2);

snom = 2\*0.169164;

vlat = advec(lat);

vlong = advec(long);

vres = zeros(size(vlat));

%sum all path vectors and calculate magnitude

for ind = 1:length(lat)-1

vres(ind) = sqrt((vlat(ind)).^2+(vlong(ind)).^2);

end

%calculate total path length, divide by total time

apath = sum(vres);

atime = apath/snom;

atd = abs(atime - Ttot);

disp(atd);

Script: Path\_Stab.m

%import data and separate columns

M=dlmread('Mission1\_Revised.csv',',',1,0);

lat = M(1:end,1);

long = M(1:end,2);

%calculate the ideal path length

ipath = sqrt((abs(lat(end)-lat(1))).^2+(abs(long(end)-long(1)).^2));

vlat = advec(lat);

vlong = advec(long);

vres = zeros(size(vlat));

%calculate magnitude of each path vector

for ind = 1:length(lat)-1

vres(ind) = sqrt((vlat(ind)).^2+(vlong(ind)).^2);

end

%calculate actual path

apath = sum(vres);

PS = apath/ipath;

disp(PS);

Script: Swarm\_T.m

%import data and separate columns

r1 = dlmread('Swarm1.csv',',',1,0);

r2 = dlmread('Swarm2.csv',',',1,0);

%convert degrees to meters

r1lat = 110570\*(r1((1:end),1));

r1long = 93367.82\*(r1((1:end),2));

r2lat = 110570\*(r2((1:end),1));

r2long = 93367.82\*(r2((1:end),2));

%determine all vectors between path points

for j = 1:length(r1)

dlat(j) = r1lat(j)-r2lat(j);

dlong(j) = r1lat(j)-r2lat(j);

rs2(j) = sqrt(((dlat(j)).^2)+((dlong(j).^2)));

end

%reference the initial distance

d0 = sqrt((dlat(1).^2)+(dlong(1).^2));

disp(d0);

%calculate mean distance

disp(mean(rs2));

Script ATD.m

%import data and separate columns

M=dlmread('Mission1\_Revised.csv',',',1,0);

Ttot = M(end,3)-M(1,3);

lat = M(1:end,1);

long = M(1:end,2);

snom = 2\*0.169164;

vlat = advec(lat);

vlong = advec(long);

vres = zeros(size(vlat));

%sum all path vectors and calculate magnitude

for ind = 1:length(lat)-1

vres(ind) = sqrt((vlat(ind)).^2+(vlong(ind)).^2);

end

%calculate total path length, divide by total time

apath = sum(vres);

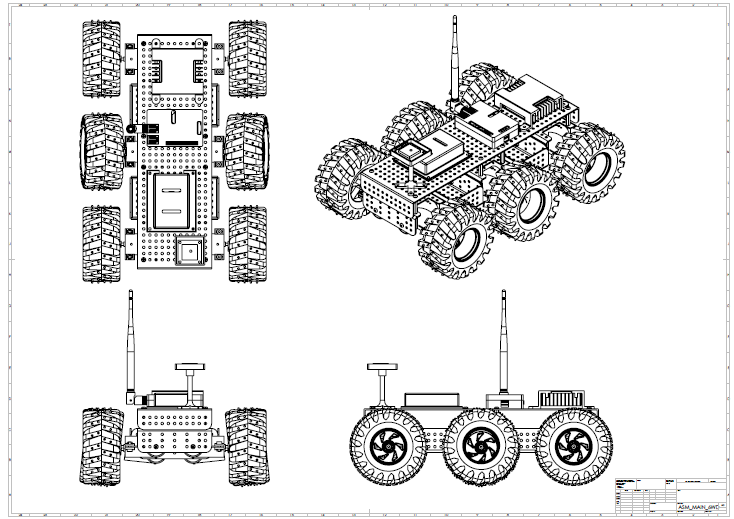
atime = apath/snom;

atd = abs(atime - Ttot);

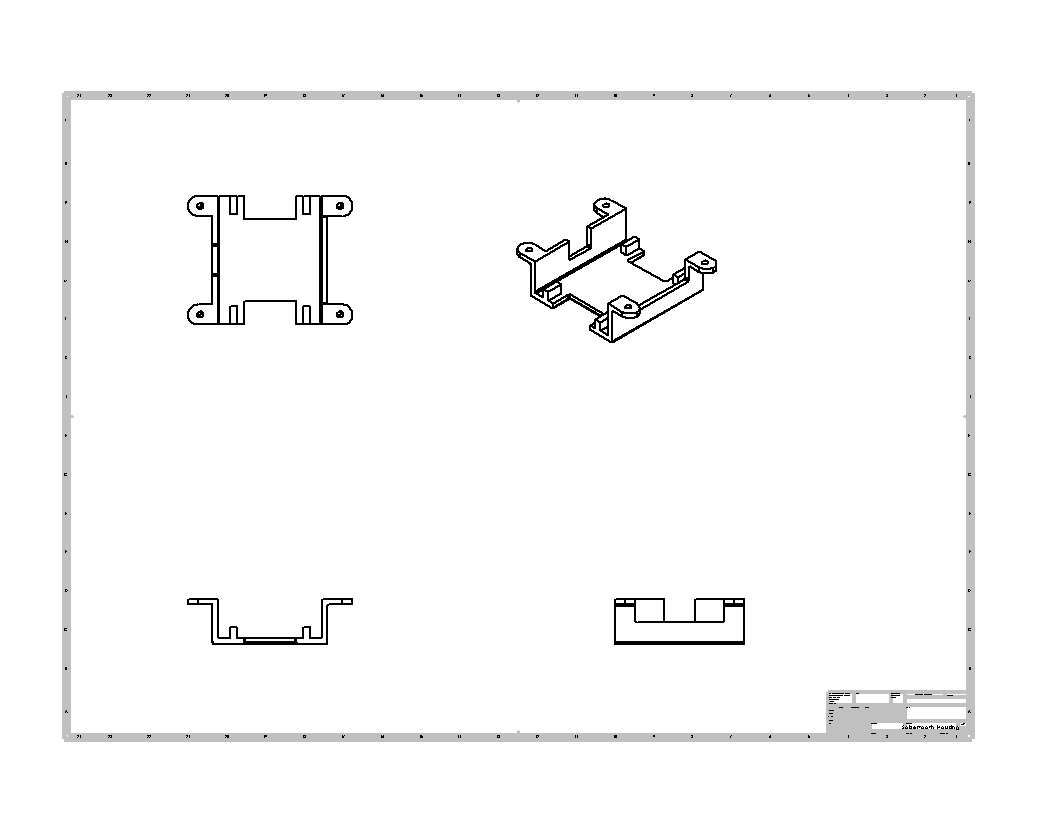
disp(atd);

Appendix E

### Final Assembly Drawing

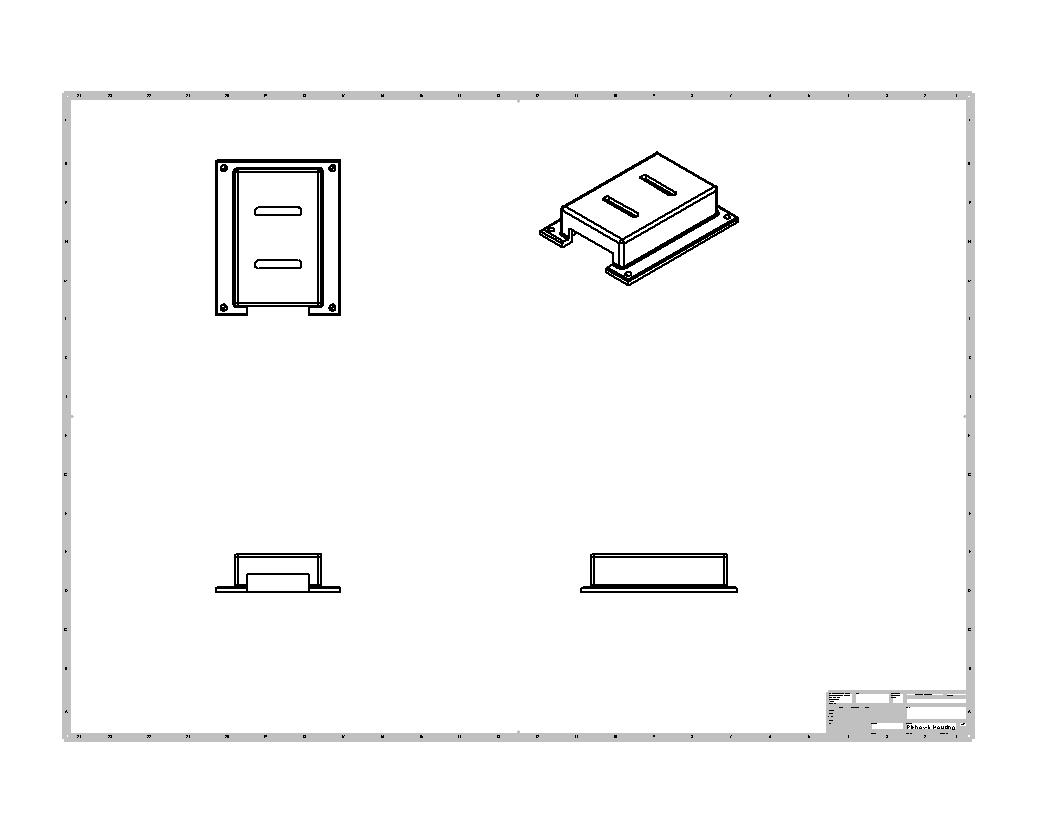


### Sabertooth Housing

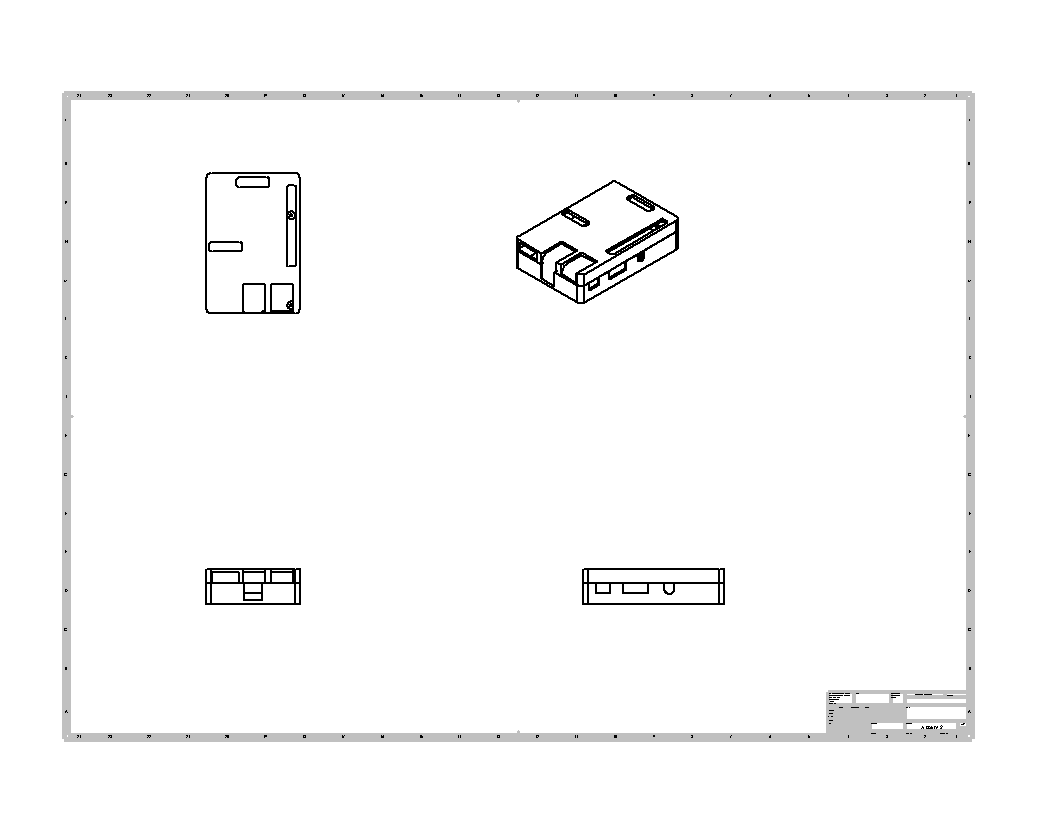


### P

### Pixhawk Housing

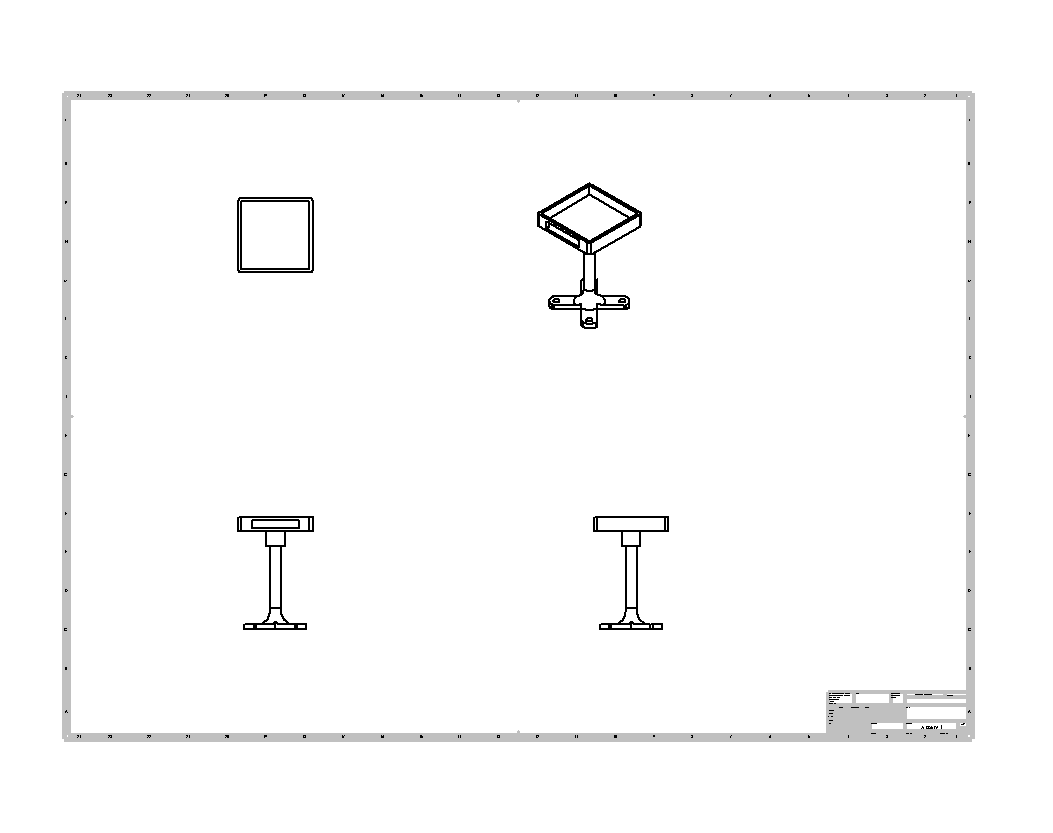


### RaspberryPi Case



### #

### 3DR Housing



Appendix F

### Equipment Inventory

A final rover parts list was created and each part was found and then marked as ready. Each part was ensured to be in good condition with all attachments necessary

### Nominal Battery Voltage

Each battery is marked for testing upon arrival. A battery is then charged to is nominal voltage using a smart LiPo charger. The batteries are checked with a voltage meter to see if the charges are correct and no over charging occurs.

### Motor Function

Each motor will, upon arrival, be tested for resistance to ensure construction is usable. The motors will then be given a sample charge in the range of the motor controller to test functionality and polarity.

### Radio Telemetry Function

The Radio Telemetry will be checked by disconnecting the digital components, ie the Pi and the Pixhawk and then wiring in a radio receiver directly into the motor controller. The Rover will be taken to a testing site and checked to see if throttle and turning are possible with the radio transceiver

### RaspberryPi and Pixhawk Compliance

The RaspberryPi was hardwired to the Pixhawk through the RaspberryPi’s serial ports and programmed with a simple python script. The RaspberryPi was hooked up to a monitor to allow for console reading and then started. The RaspberryPi must return Pixhawk attributes using MavLink Protocol commands

### Command Computer Connection

The command computer was booted and connected to the network developed for the swarm. The command computer was then given a command to ping the I.P. address of the router at the center of the network to see if packets were transmitted and received.

### RaspberryPi Connection

The RaspberryPi was booted and programmed to connect to the network developed for the swarm. The RaspberryPi was then programmed to ping the central router in the network as well as the I.P. address of the command computer. Packets were required to be sent and received to and from the network devices

Appendix G

Team Bios/Qualifications

R.E.D. Design consists of three members: Roy Wood, Ethan Daily, and Daniel Minch. Each member brought a unique perspective to the problem and each played a vital role in the success of the solution.

The first member is Roy Wood. Roy is a senior Mechanical Engineering and Computer Science student experienced in programming, algorithms, 3-D rapid-prototyping, manufacturing processes, and mathematical analysis. Roy has worked on a plethora of 3-D design projects and programming projects. Most notably, Roy was tasked to construct a 3-D design model of a local church in Macon, Georgia; the model has been since used for the reconstruction of the church. Roy Wood lead the drone development, to ensure the proper interaction between hardware and software, and time scheduling, to ensure the team is on-track to complete the project in a timely manner.

The second member is Ethan Daily. Ethan is a senior Computer Engineering student proficient in programming, microcontrollers, algorithms, and 3-D rapid-prototyping. Ethan has worked on multiple microcontroller projects, including an automatic electronic vaporizer re-filler. Ethan Daily will lead the software programming, to ensure that the drones fly intelligently, and the control center communicates effectively.

The final member is Daniel Minch. Daniel is a senior Mechanical Engineering student specializing in physical mechanics, manufacturing processes, 3-D design, and mathematical analysis. Daniel has excelled in projects involving both 3-D design for manufacturing and mathematical solutions to physical systems. Daniel Minch lead the design process and mathematical analysis, to ensuring stable, safe drone behavior.

Appendix H

**Python Backend**

import socket

import select

import sys

from threading import Thread

from thread import \*

from Tkinter import Tk, Canvas, Label, Frame, IntVar, Radiobutton, Button

from PIL import ImageTk

import math

from goompy import GooMPy

import csv

#Variables //////////////////////////////////////////////////////////////////////////////////

WIDTH = 800

HEIGHT = 500

LATITUDE = 32.826869

LONGITUDE = -83.648404

ZOOM = 17

MAPTYPE = 'OSM'

\_EARTHPIX = 268435456

\_TILESIZE = 640

\_PIXRAD = \_EARTHPIX / math.pi

\_DEGREE\_PRECISION = 4

pixels\_per\_meter = 2\*\*ZOOM / (156543.03392 \* math.cos(math.radians(LATITUDE)))

R=6378137

HOST = '192.168.1.145' #Change for Router

PORT = 50000

BUFSIZ = 2048

#Binding Server

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server.bind(("", PORT))

#server.listen(100)

list\_of\_clients = []

#Math Methods ///////////////////////////////////////////////////////////////////////////////

def \_pixToDeg(pix,zoom):

return pix \* 2 \*\* (21 - zoom)

def \_pixItLon(pix):

distOfPix = 40075016.686000 \* math.cos(LATITUDE)/((2 \*\* ZOOM)\*\_TILESIZE)

distance = ((pix-320)\*distOfPix)/(R\*math.cos(math.pi\*LATITUDE/180))

lonSelected = LONGITUDE + (distance \* 180/math.pi)

return lonSelected

def \_pixItLat(pix):

distOfPix = 40075016.686000 \* math.cos(LATITUDE)/((2 \*\* ZOOM)\*\_TILESIZE)

distance = (pix-316)\*distOfPix/R

latSelected = LATITUDE + (distance \*180/math.pi)

return latSelected

#Start UI Class //////////////////////////////////////////////////////////////////////////////

class UI(Tk):

def \_\_init\_\_(self):

print("Initializing")

Tk.\_\_init\_\_(self)

self.geometry('%dx%d+500+500' % (WIDTH,HEIGHT))

self.title('GooMPy')

self.canvas = Canvas(self, width=WIDTH, height=HEIGHT)

self.canvas.pack()

self.bind("<Key>", self.check\_quit)

self.bind('<B1-Motion>', self.drag)

self.bind('<Button-1>', self.click)

self.label = Label(self.canvas)

self.radiogroup = Frame(self.canvas)

self.radiovar = IntVar()

self.maptypes = ['OSM', 'GoogleSatellite', 'Satellite']

self.add\_radio\_button('OSM', 0)

self.add\_radio\_button('GoogleSatellite', 1)

self.add\_radio\_button('Satellite', 2)

self.zoom\_in\_button = self.add\_zoom\_button('+', +1)

self.zoom\_out\_button = self.add\_zoom\_button('-', -1)

self.zoomlevel = ZOOM

maptype\_index = 0

self.radiovar.set(maptype\_index)

self.goompy = GooMPy(WIDTH, HEIGHT, LATITUDE, LONGITUDE, ZOOM, MAPTYPE)

self.restart()

def add\_zoom\_button(self, text, sign):

button = Button(self.canvas, text=text, width=1, command=lambda:self.zoom(sign))

return button

def reload(self):

self.coords = None

self.redraw()

self['cursor'] = ''

def restart(self):

# A little trick to get a watch cursor along with loading

self['cursor'] = 'watch'

self.after(1, self.reload)

def add\_radio\_button(self, text, index):

maptype = self.maptypes[index]

Radiobutton(self.radiogroup, text=maptype, variable=self.radiovar, value=index,

command=lambda:self.usemap(maptype)).grid(row=0, column=index)

def click(self, event):

self.coords = event.x, event.y

print("X: " + str(event.x))

#print(event.x)

print("Y: " + str(event.y))

#print(event.y)

#print("\nLong: ")

#print(\_pixItLon(event.x))

#print("\nLLat: ")

#print(\_pixItLat(event.y))

#coods = str(\_pixItLat(event.y)) + ';' + str(\_pixItLon(event.x))

coods = str(event.x-315) + ";" + str(309-event.y)

coods = coods.encode()

broadcast(coods,HOST)

def drag(self, event):

#self.goompy.move(self.coords[0]-event.x, self.coords[1]-event.y)

#self.image = self.goompy.getImage()

#self.redraw()

self.coords = event.x, event.y

#print("X: " + event.x + "\nY:: " + event.y)

def redraw(self):

self.image = self.goompy.getImage()

self.image\_tk = ImageTk.PhotoImage(self.image)

self.label['image'] = self.image\_tk

self.label.place(x=0, y=0, width=WIDTH, height=HEIGHT)

self.radiogroup.place(x=0,y=0)

x = int(self.canvas['width']) - 50

y = int(self.canvas['height']) - 80

self.zoom\_in\_button.place(x= x, y=y)

self.zoom\_out\_button.place(x= x, y=y+30)

def usemap(self, maptype):

self.goompy.useMaptype(maptype)

self.restart()

def zoom(self, sign):

newlevel = self.zoomlevel + sign

if newlevel > 0 and newlevel < 22:

self.zoomlevel = newlevel

self.goompy.useZoom(newlevel)

self.restart()

def check\_quit(self, event):

if ord(event.char) == 27: # ESC

exit(0)

#END OF UI////////////////////////////////////////////////////////////////////////////////////////////////

#Server Methods //////////////////////////////////////////////////////////////////////////////////////////

def clientthread(conn, addr):

# sends a message to the client whose user object is conn

conn.send(addr[0])

fieldnames = ['Latitude', 'Longitude', 'TimeStamp']

with open('TestData' + addr[0] + '.csv', mode='w') as csv\_file:

writer = csv.DictWriter(csv\_file, fieldnames=fieldnames)

writer.writeheader()

writer.writerow({'Latitude': 123456, 'Longitude': 123456, 'TimeStamp': 123456})

while True:

try:

message = conn.recv(2048)

message = message.decode()

print(message)

temp\_data = message.split(";")

with open('TestData' + addr[0] + '.csv', mode='a') as csv\_file:

writer = csv.DictWriter(csv\_file, fieldnames=fieldnames)

writer.writerow({'Latitude': temp\_data[0], 'Longitude': temp\_data[1], 'TimeStamp': temp\_data[2]})

# Calls broadcast function to send message to all

message\_to\_send = "<" + addr[0] + "> " + message

message\_to\_send = message\_to\_send.encode()

broadcast(message\_to\_send, conn)

except:

continue

"""Using the below function, we broadcast the message to all

clients who's object is not the same as the one sending

the message """

def broadcast(message, connection):

#print ("Enter Broadcast")

for clients in list\_of\_clients:

if clients!=connection:

try:

print(message)

#print(clients)

clients.send(message)

except:

clients.close()

# if the link is broken, we remove the client

#remove(clients)

"""The following function simply removes the object

from the list that was created at the beginning of

the program"""

def remove(connection):

if connection in list\_of\_clients:

list\_of\_clients.remove(connection)

def accept\_connections():

"""Sets up handling for incoming clients."""

while True:

conn, addr = server.accept()

list\_of\_clients.append(conn)

print addr[0] + " connected"

#conn.send(bytes("S;", "utf8"))

Thread(target=clientthread, args=(conn,addr)).start()

#Main ////////////////////////////////////////////////////////////////////////

if \_\_name\_\_ == "\_\_main\_\_":

#Server listens for 5 active connections

server.listen(5)

print("Waiting to connect to swarm....")

accept\_thread = Thread(target=accept\_connections)

accept\_thread.start()

UI().mainloop()

#Close the connections`

server.close()

**Python Frontend**

from dronekit import \*

from pymavlink import mavutil

#from \_\_\_future\_\_ import print\_function

import time, math

import socket

import select

import sys

import csv

from threading import Thread

#Example documentation: http://python.dronekit.io/examples/guided-set-speed-yaw-demo.html

def arm\_and\_takeoff(aTargetAltitude):

print("Basic pre-arm checks")

# Don't let the user try to arm until autopilot is ready

while not vehicle.is\_armable:

print(" Waiting for vehicle to initialise...")

time.sleep(1)

print("Arming motors")

# Copter should arm in GUIDED mode

vehicle.mode = VehicleMode("GUIDED")

vehicle.armed = True

while not vehicle.armed:

print(" Waiting for arming...")

time.sleep(1)

print("Taking off!")

vehicle.simple\_takeoff(aTargetAltitude) # Take off to target altitude

# Wait until the vehicle reaches a safe height before processing the goto (otherwise the command

# after Vehicle.simple\_takeoff will execute immediately).

while True:

print(" Altitude: ", vehicle.location.global\_relative\_frame.alt)

if vehicle.location.global\_relative\_frame.alt>=aTargetAltitude\*0.95: #Trigger just below target alt.

print("Reached target altitude")

break

time.sleep(1)

"""#

Convenience functions for sending immediate/guided mode commands to control the Copter.

The set of commands demonstrated here include:

\* MAV\_CMD\_CONDITION\_YAW - set direction of the front of the Copter (latitude, longitude)

\* MAV\_CMD\_DO\_SET\_ROI - set direction where the camera gimbal is aimed (latitude, longitude, altitude)

\* MAV\_CMD\_DO\_CHANGE\_SPEED - set target speed in metres/second.

The full set of available commands are listed here:

http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/

"""

def condition\_yaw(heading, relative=False):

"""#

Send MAV\_CMD\_CONDITION\_YAW message to point vehicle at a specified heading (in degrees).

This method sets an absolute heading by default, but you can set the `relative` parameter

to `True` to set yaw relative to the current yaw heading.

By default the yaw of the vehicle will follow the direction of travel. After setting

the yaw using this function there is no way to return to the default yaw "follow direction

of travel" behaviour (https://github.com/diydrones/ardupilot/issues/2427)

For more information see:

http://copter.ardupilot.com/wiki/common-mavlink-mission-command-messages-mav\_cmd/#mav\_cmd\_condition\_yaw

"""

if relative:

is\_relative = 1 #yaw relative to direction of travel

else:

is\_relative = 0 #yaw is an absolute angle

# create the CONDITION\_YAW command using command\_long\_encode()

msg = vehicle.message\_factory.command\_long\_encode(

0, 0, # target system, target component

mavutil.mavlink.MAV\_CMD\_CONDITION\_YAW, #command

0, #confirmation

heading, # param 1, yaw in degrees

0, # param 2, yaw speed deg/s

1, # param 3, direction -1 ccw, 1 cw

is\_relative, # param 4, relative offset 1, absolute angle 0

0, 0, 0) # param 5 ~ 7 not used

# send command to vehicle

vehicle.send\_mavlink(msg)

def set\_roi(location):

"""#

Send MAV\_CMD\_DO\_SET\_ROI message to point camera gimbal at a

specified region of interest (LocationGlobal).

The vehicle may also turn to face the ROI.

For more information see:

http://copter.ardupilot.com/common-mavlink-mission-command-messages-mav\_cmd/#mav\_cmd\_do\_set\_roi

"""

# create the MAV\_CMD\_DO\_SET\_ROI command

msg = vehicle.message\_factory.command\_long\_encode(

0, 0, # target system, target component

mavutil.mavlink.MAV\_CMD\_DO\_SET\_ROI, #command

0, #confirmation

0, 0, 0, 0, #params 1-4

location.lat,

location.lon,

location.alt

)

# send command to vehicle

vehicle.send\_mavlink(msg)

"""#

Functions to make it easy to convert between the different frames-of-reference. In particular these

make it easy to navigate in terms of "metres from the current position" when using commands that take

absolute positions in decimal degrees.

The methods are approximations only, and may be less accurate over longer distances, and when close

to the Earth's poles.

Specifically, it provides:

\* get\_location\_metres - Get LocationGlobal (decimal degrees) at distance (m) North & East of a given LocationGlobal.

\* get\_distance\_metres - Get the distance between two LocationGlobal objects in metres

\* get\_bearing - Get the bearing in degrees to a LocationGlobal

"""

def get\_location\_metres(original\_location, dNorth, dEast):

"""#

Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dEast` metres from the

specified `original\_location`. The returned LocationGlobal has the same `alt` value

as `original\_location`.

The function is useful when you want to move the vehicle around specifying locations relative to

the current vehicle position.

The algorithm is relatively accurate over small distances (10m within 1km) except close to the poles.

For more information see:

http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-longitude-by-some-amount-of-meters

"""

earth\_radius = 6378137.0 #Radius of "spherical" earth

#Coordinate offsets in radians

dLat = dNorth/earth\_radius

dLon = dEast/(earth\_radius\*math.cos(math.pi\*original\_location.lat/180))

#New position in decimal degrees

newlat = original\_location.lat + (dLat \* 180/math.pi)

newlon = original\_location.lon + (dLon \* 180/math.pi)

if type(original\_location) is LocationGlobal:

targetlocation=LocationGlobal(newlat, newlon,original\_location.alt)

elif type(original\_location) is LocationGlobalRelative:

targetlocation=LocationGlobalRelative(newlat, newlon,original\_location.alt)

else:

raise Exception("Invalid Location object passed")

return targetlocation;

def get\_distance\_metres(aLocation1, aLocation2):

"""#

Returns the ground distance in metres between two LocationGlobal objects.

This method is an approximation, and will not be accurate over large distances and close to the

earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py

"""

dlat = aLocation2.lat - aLocation1.lat

dlong = aLocation2.lon - aLocation1.lon

return math.sqrt((dlat\*dlat) + (dlong\*dlong)) \* 1.113195e5

def get\_bearing(aLocation1, aLocation2):

"""#

Returns the bearing between the two LocationGlobal objects passed as parameters.

This method is an approximation, and may not be accurate over large distances and close to the

earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py

"""

off\_x = aLocation2.lon - aLocation1.lon

off\_y = aLocation2.lat - aLocation1.lat

bearing = 90.00 + math.atan2(-off\_y, off\_x) \* 57.2957795

if bearing < 0:

bearing += 360.00

return bearing;

"""#

Functions to move the vehicle to a specified position (as opposed to controlling movement by setting velocity components).

The methods include:

\* goto\_position\_target\_global\_int - Sets position using SET\_POSITION\_TARGET\_GLOBAL\_INT command in

MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT frame

\* goto\_position\_target\_local\_ned - Sets position using SET\_POSITION\_TARGET\_LOCAL\_NED command in

MAV\_FRAME\_BODY\_NED frame

\* goto - A convenience function that can use Vehicle.simple\_goto (default) or

goto\_position\_target\_global\_int to travel to a specific position in metres

North and East from the current location.

This method reports distance to the destination.

"""

def goto\_position\_target\_global\_int(aLocation):

"""#

Send SET\_POSITION\_TARGET\_GLOBAL\_INT command to request the vehicle fly to a specified LocationGlobal.

For more information see: https://pixhawk.ethz.ch/mavlink/#SET\_POSITION\_TARGET\_GLOBAL\_INT

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_global\_int\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT, # frame

0b0000111111111000, # type\_mask (only speeds enabled)

aLocation.lat\*1e7, # lat\_int - X Position in WGS84 frame in 1e7 \* meters

aLocation.lon\*1e7, # lon\_int - Y Position in WGS84 frame in 1e7 \* meters

aLocation.alt, # alt - Altitude in meters in AMSL altitude, not WGS84 if absolute or relative, above terrain if GLOBAL\_TERRAIN\_ALT\_INT

0, # X velocity in NED frame in m/s

0, # Y velocity in NED frame in m/s

0, # Z velocity in NED frame in m/s

0, 0, 0, # afx, afy, afz acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle

vehicle.send\_mavlink(msg)

def goto\_position\_target\_local\_ned(north, east, down):

"""

Send SET\_POSITION\_TARGET\_LOCAL\_NED command to request the vehicle fly to a specified

location in the North, East, Down frame.

It is important to remember that in this frame, positive altitudes are entered as negative

"Down" values. So if down is "10", this will be 10 metres below the home altitude.

Starting from AC3.3 the method respects the frame setting. Prior to that the frame was

ignored. For more information see:

http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_local\_ned

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_local\_ned\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_LOCAL\_NED, # frame

0b0000111111111000, # type\_mask (only positions enabled)

north, east, down, # x, y, z positions (or North, East, Down in the MAV\_FRAME\_BODY\_NED frame

0, 0, 0, # x, y, z velocity in m/s (not used)

0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle

vehicle.send\_mavlink(msg)

"""

def goto(dNorth, dEast, gotoFunction=vehicle.simple\_goto):

Moves the vehicle to a position dNorth metres North and dEast metres East of the current position.

The method takes a function pointer argument with a single `dronekit.lib.LocationGlobal` parameter for

the target position. This allows it to be called with different position-setting commands.

By default it uses the standard method: dronekit.lib.Vehicle.simple\_goto().

The method reports the distance to target every two seconds.

currentLocation = vehicle.location.global\_relative\_frame

targetLocation = get\_location\_metres(currentLocation, dNorth, dEast)

targetDistance = get\_distance\_metres(currentLocation, targetLocation)

gotoFunction(targetLocation)

#print "DEBUG: targetLocation: %s" % targetLocation

#print "DEBUG: targetLocation: %s" % targetDistance

while vehicle.mode.name=="GUIDED": #Stop action if we are no longer in guided mode.

#print "DEBUG: mode: %s" % vehicle.mode.name

remainingDistance=get\_distance\_metres(vehicle.location.global\_relative\_frame, targetLocation)

print("Distance to target: ", remainingDistance)

if remainingDistance<=targetDistance\*0.01: #Just below target, in case of undershoot.

print("Reached target")

break;

time.sleep(2)

"""

"""#

Functions that move the vehicle by specifying the velocity components in each direction.

The two functions use different MAVLink commands. The main difference is

that depending on the frame used, the NED velocity can be relative to the vehicle

orientation.

The methods include:

\* send\_ned\_velocity - Sets velocity components using SET\_POSITION\_TARGET\_LOCAL\_NED command

\* send\_global\_velocity - Sets velocity components using SET\_POSITION\_TARGET\_GLOBAL\_INT command

"""

def send\_ned\_velocity(velocity\_x, velocity\_y, velocity\_z, duration):

"""#

Move vehicle in direction based on specified velocity vectors and

for the specified duration.

This uses the SET\_POSITION\_TARGET\_LOCAL\_NED command with a type mask enabling only

velocity components

(http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_local\_ned).

Note that from AC3.3 the message should be re-sent every second (after about 3 seconds

with no message the velocity will drop back to zero). In AC3.2.1 and earlier the specified

velocity persists until it is canceled. The code below should work on either version

(sending the message multiple times does not cause problems).

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_local\_ned\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_LOCAL\_NED, # frame

0b0000111111000111, # type\_mask (only speeds enabled)

0, 0, 0, # x, y, z positions (not used)

velocity\_x, velocity\_y, velocity\_z, # x, y, z velocity in m/s

0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle on 1 Hz cycle

for x in range(0,duration):

vehicle.send\_mavlink(msg)

time.sleep(1)

def send\_global\_velocity(velocity\_x, velocity\_y, velocity\_z, duration):

"""#

Move vehicle in direction based on specified velocity vectors.

This uses the SET\_POSITION\_TARGET\_GLOBAL\_INT command with type mask enabling only

velocity components

(http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_global\_int).

Note that from AC3.3 the message should be re-sent every second (after about 3 seconds

with no message the velocity will drop back to zero). In AC3.2.1 and earlier the specified

velocity persists until it is canceled. The code below should work on either version

(sending the message multiple times does not cause problems).

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_global\_int\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT, # frame

0b0000111111000111, # type\_mask (only speeds enabled)

0, # lat\_int - X Position in WGS84 frame in 1e7 \* meters

0, # lon\_int - Y Position in WGS84 frame in 1e7 \* meters

0, # alt - Altitude in meters in AMSL altitude(not WGS84 if absolute or relative)

# altitude above terrain if GLOBAL\_TERRAIN\_ALT\_INT

velocity\_x, # X velocity in NED frame in m/s

velocity\_y, # Y velocity in NED frame in m/s

velocity\_z, # Z velocity in NED frame in m/s

0, 0, 0, # afx, afy, afz acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle on 1 Hz cycle

for x in range(0,duration):

vehicle.send\_mavlink(msg)

time.sleep(1)

"""#

Functions to make it easy to convert between the different frames-of-reference. In particular these

make it easy to navigate in terms of "metres from the current position" when using commands that take

absolute positions in decimal degrees.

The methods are approximations only, and may be less accurate over longer distances, and when close

to the Earth's poles.

Specifically, it provides:

\* get\_location\_metres - Get LocationGlobal (decimal degrees) at distance (m) North & East of a given LocationGlobal.

\* get\_distance\_metres - Get the distance between two LocationGlobal objects in metres

\* get\_bearing - Get the bearing in degrees to a LocationGlobal

"""

def get\_location\_metres(original\_location, dNorth, dEast):

"""#

Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dEast` metres from the

specified `original\_location`. The returned LocationGlobal has the same `alt` value

as `original\_location`.

The function is useful when you want to move the vehicle around specifying locations relative to

the current vehicle position.

The algorithm is relatively accurate over small distances (10m within 1km) except close to the poles.

For more information see:

http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-longitude-by-some-amount-of-meters

"""

earth\_radius = 6378137.0 #Radius of "spherical" earth

#Coordinate offsets in radians

dLat = dNorth/earth\_radius

dLon = dEast/(earth\_radius\*math.cos(math.pi\*original\_location.lat/180))

#New position in decimal degrees

newlat = original\_location.lat + (dLat \* 180/math.pi)

newlon = original\_location.lon + (dLon \* 180/math.pi)

if type(original\_location) is LocationGlobal:

targetlocation=LocationGlobal(newlat, newlon,original\_location.alt)

elif type(original\_location) is LocationGlobalRelative:

targetlocation=LocationGlobalRelative(newlat, newlon,original\_location.alt)

else:

raise Exception("Invalid Location object passed")

return targetlocation;

def get\_distance\_metres(aLocation1, aLocation2):

"""#

Returns the ground distance in metres between two LocationGlobal objects.

This method is an approximation, and will not be accurate over large distances and close to the

earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py

"""

dlat = aLocation2.lat - aLocation1.lat

dlong = aLocation2.lon - aLocation1.lon

return math.sqrt((dlat\*dlat) + (dlong\*dlong)) \* 1.113195e5

def get\_bearing(aLocation1, aLocation2):

"""#

Returns the bearing between the two LocationGlobal objects passed as parameters.

This method is an approximation, and may not be accurate over large distances and close to the

earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py

"""

off\_x = aLocation2.lon - aLocation1.lon

off\_y = aLocation2.lat - aLocation1.lat

bearing = 90.00 + math.atan2(-off\_y, off\_x) \* 57.2957795

if bearing < 0:

bearing += 360.00

return bearing;

"""#

Functions to move the vehicle to a specified position (as opposed to controlling movement by setting velocity components).

The methods include:

\* goto\_position\_target\_global\_int - Sets position using SET\_POSITION\_TARGET\_GLOBAL\_INT command in

MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT frame

\* goto\_position\_target\_local\_ned - Sets position using SET\_POSITION\_TARGET\_LOCAL\_NED command in

MAV\_FRAME\_BODY\_NED frame

\* goto - A convenience function that can use Vehicle.simple\_goto (default) or

goto\_position\_target\_global\_int to travel to a specific position in metres

North and East from the current location.

This method reports distance to the destination.

"""

def goto\_position\_target\_global\_int(aLocation):

"""#

Send SET\_POSITION\_TARGET\_GLOBAL\_INT command to request the vehicle fly to a specified LocationGlobal.

For more information see: https://pixhawk.ethz.ch/mavlink/#SET\_POSITION\_TARGET\_GLOBAL\_INT

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_global\_int\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT, # frame

0b0000111111111000, # type\_mask (only speeds enabled)

aLocation.lat\*1e7, # lat\_int - X Position in WGS84 frame in 1e7 \* meters

aLocation.lon\*1e7, # lon\_int - Y Position in WGS84 frame in 1e7 \* meters

aLocation.alt, # alt - Altitude in meters in AMSL altitude, not WGS84 if absolute or relative, above terrain if GLOBAL\_TERRAIN\_ALT\_INT

0, # X velocity in NED frame in m/s

0, # Y velocity in NED frame in m/s

0, # Z velocity in NED frame in m/s

0, 0, 0, # afx, afy, afz acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle

vehicle.send\_mavlink(msg)

def goto\_position\_target\_local\_ned(north, east, down):

"""

Send SET\_POSITION\_TARGET\_LOCAL\_NED command to request the vehicle fly to a specified

location in the North, East, Down frame.

It is important to remember that in this frame, positive altitudes are entered as negative

"Down" values. So if down is "10", this will be 10 metres below the home altitude.

Starting from AC3.3 the method respects the frame setting. Prior to that the frame was

ignored. For more information see:

http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_local\_ned

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_local\_ned\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_LOCAL\_NED, # frame

0b0000111111111000, # type\_mask (only positions enabled)

north, east, down, # x, y, z positions (or North, East, Down in the MAV\_FRAME\_BODY\_NED frame

0, 0, 0, # x, y, z velocity in m/s (not used)

0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle

vehicle.send\_mavlink(msg)

def send\_ned\_velocity(velocity\_x, velocity\_y, velocity\_z, duration):

"""

Move vehicle in direction based on specified velocity vectors and

for the specified duration.

This uses the SET\_POSITION\_TARGET\_LOCAL\_NED command with a type mask enabling only

velocity components

(http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_local\_ned).

Note that from AC3.3 the message should be re-sent every second (after about 3 seconds

with no message the velocity will drop back to zero). In AC3.2.1 and earlier the specified

velocity persists until it is canceled. The code below should work on either version

(sending the message multiple times does not cause problems).

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_local\_ned\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_LOCAL\_NED, # frame

0b0000111111000111, # type\_mask (only speeds enabled)

0, 0, 0, # x, y, z positions (not used)

velocity\_x, velocity\_y, velocity\_z, # x, y, z velocity in m/s

0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle on 1 Hz cycle

for x in range(0,duration):

vehicle.send\_mavlink(msg)

time.sleep(1)

def send\_global\_velocity(velocity\_x, velocity\_y, velocity\_z, duration):

"""

Move vehicle in direction based on specified velocity vectors.

This uses the SET\_POSITION\_TARGET\_GLOBAL\_INT command with type mask enabling only

velocity components

(http://dev.ardupilot.com/wiki/copter-commands-in-guided-mode/#set\_position\_target\_global\_int).

Note that from AC3.3 the message should be re-sent every second (after about 3 seconds

with no message the velocity will drop back to zero). In AC3.2.1 and earlier the specified

velocity persists until it is canceled. The code below should work on either version

(sending the message multiple times does not cause problems).

See the above link for information on the type\_mask (0=enable, 1=ignore).

At time of writing, acceleration and yaw bits are ignored.

"""

msg = vehicle.message\_factory.set\_position\_target\_global\_int\_encode(

0, # time\_boot\_ms (not used)

0, 0, # target system, target component

mavutil.mavlink.MAV\_FRAME\_GLOBAL\_RELATIVE\_ALT\_INT, # frame

0b0000111111000111, # type\_mask (only speeds enabled)

0, # lat\_int - X Position in WGS84 frame in 1e7 \* meters

0, # lon\_int - Y Position in WGS84 frame in 1e7 \* meters

0, # alt - Altitude in meters in AMSL altitude(not WGS84 if absolute or relative)

# altitude above terrain if GLOBAL\_TERRAIN\_ALT\_INT

velocity\_x, # X velocity in NED frame in m/s

velocity\_y, # Y velocity in NED frame in m/s

velocity\_z, # Z velocity in NED frame in m/s

0, 0, 0, # afx, afy, afz acceleration (not supported yet, ignored in GCS\_Mavlink)

0, 0) # yaw, yaw\_rate (not supported yet, ignored in GCS\_Mavlink)

# send command to vehicle on 1 Hz cycle

for x in range(0,duration):

vehicle.send\_mavlink(msg)

time.sleep(1)

def preChecks():

print("Attributes")

print("-----------")

print("GPS: %s" % vehicle.gps\_0)

print("Battery %s" % vehicle.battery)

print("Last Heartbeat: %s" % vehicle.last\_heartbeat)

print(" ")

print("Pre-Arm Checks")

print("---------------")

while not(vehicle.is\_armable):

print("Waiting for vehicle to initialize...")

time.sleep(5)

def armVehicle():

print("Arming motors")

vehicle.mode = VehicleMode("GUIDED")

vehicle.armed = True

print("DEBUG")

while not(vehicle.armed):

print("Waiting to arm...")

print("Motors armed")

#-------------------------------------------------------------------

def handle\_client():

firstmessage = True

while True:

sockets\_list = [server]

read\_sockets, write\_socket, error\_socket = select.select(sockets\_list,[],[])

for socks in read\_sockets:

if socks == server:

message = socks.recv(2048)

message = message.decode()

#print(message)

if(firstmessage):

ID = message[10]

firstmessage = False

else:

temp = message.split(";")

#the\_loc.lat = float(temp[0])

#the\_loc.lon = float(temp[1])

lonTest = float(temp[0])/100

latTest = float(temp[1])/100

else:

#message = sys.stdin.readline()

message = vehicle.location.global\_frame.lat + ";" + vehicle.location.global\_frame.lon + ";" + str(time.time()-base\_time)

message = message.encode()

server.send(message)

#print(message)

#sys.stdout.write(message)

#sys.stdout.write("You: ")

#sys.stdout.flush()

message = ""

server.close()

#----------------------------MAIN-----------------------------------

connection\_string = "/dev/ttyAMA0"

ID = ""

base\_time = time.time()

print("Connecting to Vehicle")

vehicle = connect(connection\_string,baud=921600,wait\_ready = True)

alt = vehicle.location.global\_frame.alt

lonTest = 0

latTest = 0

home = LocationGlobalRelative(vehicle.location.global\_frame.lat,vehicle.location.global\_frame.lon,alt)

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

ip\_addr = '192.168.1.145'

port = 50000

time.sleep(15)

server.connect((ip\_addr, port))

Thread(target=handle\_client).start()

preChecks()

armVehicle()

print("AFTER armVehicle")

#set groundspeed

vehicle.groundspeed = 5

the\_loc = get\_location\_metres(home,40,40)

fieldnames = ['Latitude,Longitude,Timestamp']

with open('/boot/TestData.csv', mode='w') as csv\_file:

writer = csv.writer(csv\_file)

writer.writerow(fieldnames)

c\_time = time.time()

while True:

#print("IN WHILE LOOP")

vehicle.simple\_goto(the\_loc)

#if((time.time()-c\_time) > 5):

# the\_loc=home

#send\_ned\_velocity((vehicle.location.global\_frame.lon+lonTest),(vehicle.location.global\_frame.lat+latTest),0,(1))

with open('/boot/TestData.csv', mode='a') as csv\_file:

writer = csv.writer(csv\_file, delimiter=',')

writer.writerow([vehicle.location.global\_frame.lat,vehicle.location.global\_frame.lon,time.time()-base\_time])

#send\_ned\_velocity(2,0,0,3)

# print(str(the\_loc.lat) + ";" + str(the\_loc.lon))

time.sleep(2)

#def GetLocationOffsetMeter(origin, dNorth, dEast, alt):

#send\_ned\_velocity(2,0,0,5)

#send\_ned\_velocity(0,0,0,1)

#send\_ned\_velocity(-2,0,0,5)

"""#SimpleGoToNorth

print ("Setting up and going to...")

sampleLat = vehicle.location.global\_frame.lat+0.0002

while(vehicle.location.global\_frame.lat < sampleLat):

vehicle.simple\_goto(theLoc)

time.sleep(1)

print ("Setting up and going to...")

sampleLat2 = vehicle.location.global\_frame.lat-0.0002

theLoc2 = LocationGlobal(sampleLat2,vehicle.location.global\_frame.lon,32.839355)

while(vehicle.location.global\_frame.lat > sampleLat2):

vehicle.simple\_goto(theLoc2)

time.sleep(1)"""

#vehicle.mode = VehicleMode("HALT")