6-10-2020

Adaptive Navigation Utilizing a Drone Cluster

Zach Cameron

Brendan Engh

Thomas Kambe

Aditya Krishnan

SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Date: June 10, 2020

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Zach Cameron
Brendan Engh
Thomas Kambe
Aditya Krishnan

ENTITLED

Adaptive Navigation Utilizing a Drone Cluster

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
BACHELOR OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

Christopher A Kitts — Jun 10, 2020
Christopher A Kitts (Jun 10, 2020 11:52 PDT)
Thesis Advisor 1: Christopher Kitts

Sally L. Wood — Jun 10, 2020
Sally L. Wood (Jun 10, 2020 12:30 PDT)
Thesis Advisor 2: Sally Wood

Nam Ling — Jun 10, 2020
Nam Ling (Jun 10, 2020 23:16 PDT)
Computer Science and Engineering Dept Chair

Shoba Krishnan — Jun 10, 2020
Shoba Krishnan (Jun 10, 2020 12:42 PDT)
Electrical Engineering Dept Chair

Adaptive Navigation Utilizing a Drone Cluster

by

Zach Cameron
Brendan Engh
Thomas Kambe
Aditya Krishnan

Submitted in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Electrical and Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 10, 2020

Adaptive Navigation Utilizing a Drone Cluster

Zach Cameron
Brendan Engh
Thomas Kambe
Aditya Krishnan

Department of Computer Science and Engineering
Department of Electrical and Computer Engineering
Santa Clara University
June 10, 2020

ABSTRACT

The current infrastructure for drone piloting involves drones flying in a predetermined path to find an object, source, or target with a known location. Drones should be able to determine an optimal flight path mid-flight in order to find an undefined source or target given existing parameters that can be analyzed from incoming sensor data. However, a framework for such a feat is almost non-existent. Our project aims to build such a framework that allows for the control of and communication among multiple drones, called a cluster, and allows for in-flight analysis and processing of sensor information to determine how to progress towards finding a source. This second objective of adaptive navigation is when the cluster of drones report their sensor data to a central ground station, allowing for a gradient calculation to be executed. The cluster of drones can follow this gradient to progress towards a source with a previously unknown specific location or path to the location.

Acknowledgments

Thank you to The SCU School of Engineering Undergraduate Programs for providing financial support.
Thank you for the SCU Robotic Systems Lab for providing financial support and a work space to execute our project.
Thank you to Dr. Kitts and Dr. Wood for providing their expertise and support throughout the project.
Thank you to Mike Rasay for his help in acquiring images, help with team management, and help with designing our overall system architecture
Thank you to Anne Mahacek Hunter for her help with ordering and acquiring products, helping to properly train each of our team members, and helping us to acquire a space within the lab to work.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the last few years, we have seen a rapid evolution within the consumer and commercial drone industry. Due to more energy-dense batteries, better motor technology, smaller electrical components, and superior manufacturing, multi-copter drones have been adopted within many industries. A 2019 report estimates that the drone service industry will grow from 4.4 billion USD in 2018 to 63.6 billion USD by 2025 [6]. A specific drone system that fits into this category is the DJI AGRAS T16 shown in Figure 1.1. This drone is capable of autonomously mapping a farm and spraying individual crops 40-60 times faster than manual spraying [1]. This drone is just one of many multi-copter drone solutions. The wide range of multi-copter-involved markets also includes photography and filming, security and defense, mapping, and surveying. Drones have been successfully utilized in all these markets but with a combination of certain advanced capabilities such as multi-drone operation, their potential can be fully realized.

Figure 1.1: DJI AGRAS T16 agricultural drone [1]

Multi-drone applications take advantage of multiple drones to complete a task quicker than a single drone or perform a functionality not possible with a single drone. Multi-drone research is an emerging topic in the research and develop realm and some systems are starting to come online within commercial markets. Examples include the 2018 Olympic

drone show put on by Intel's Shooting Star drone light show, and Embention's Veronte autopilot and drone system used for mapping fires [2] [7]. The Intel drone swarm as shown in Figure 1.2 uses software to choreograph all the drone's flights before take-off whereas the Ebention's drone system uses more complicated control techniques that allow for in-flight formation control.

Figure 1.2: Intel's 2018 Olympic Shooting Star drone light show [2]

The difficulty with such a system is having manageable control for a pilot and reliable communication with each of the drones. Challenges that must be overcome to implement a multi-drone system include, cost, logistics, maintenance, safety, and communication. Communication is especially important as these flight systems need to function simultaneously and in an efficient and intuitive way without causing overload. Once these challenges are meet the challenges of formation control arise.

Many techniques exist for the control of a formation of multi-copters, each with varying degrees of complexity dependent on the specific application. In the case of the Ebention's system, they use two relatively simple approaches. The first being a leader-follower approach, where all nodes follow a master node. The second is a flock approach, where each node follows its closest node. Another approach to formation control is using a Virtual Bodies and Artificial Potentials (VBAP) formation [8]. In this formation, field potentials from the environment and other drones are used to maintain a formation. Finally the approach that is relevant to our work is using the robotic cluster control technique that will be explained later in this paper.

1.1 Adaptive Navigation

Traditionally to navigate a field we would use a technique known as mowing the lawn. This technique involves having a drone exhaustively map the environment in question. An illustration of this technique can be seen in Figure 1.3 where the drone will need to follow the dashed contour to "mow the lawn". There are many applications for which standard "mow the lawn" navigation over a region makes sense. For these applications, it is important to collect and possibly merge sensor data collected over the entire region. Applications for this approach include things like

systematic mapping of a region and visual search and rescue operations.

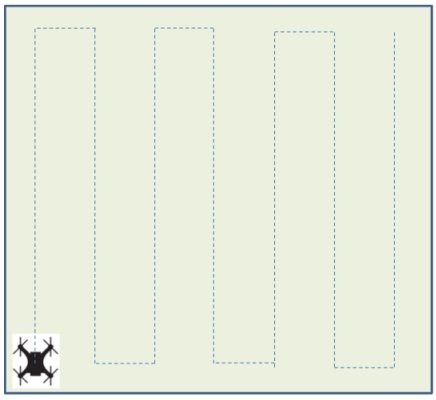


Figure 1.3: Mowing The Lawn [3]

Many applications for drone clusters exist. However, for such applications, exhaustive navigation of an entire region is unnecessary. For example, consider scenarios and situations where a defined source or target is known and measurable but its location is unknown. The goal of using a drone cluster is not to maximize coverage with the formation but is instead to use the formation to get an instantaneous estimate of field characteristics such as the gradient. In such a scenario, running through the terrain using a mow-the-lawn technique is not as effective, especially if finding the source requires a sense of urgency. In such cases, a drone cluster can be used to efficiently locate the source.

In the case of a scenario like pollution source-seeking, where pollution extends from the source with decreasing strength and with a measurable gradient, utilizing a controllable cluster formation enables the cluster to use appropriate sensors in order to collect simultaneous distributed data from which information about the structure of the field can be computed [3]. This results in greater efficiency. Compared to this, the process of mowing the lawn with a cluster still forces the mapping of terrain where, in this pollution-seeking case, might not be affected at all by the pollution. This can result in a waste of space searched by the cluster when attempting to locate the source of pollution.

In this case, if the cluster could adapt its direction of travel and possibly its geometry based on its sensor readings in order to more efficiently locate the pollution source. For example, the cluster could use its readings to compute the gradient of pollution concentration level and then move in that direction to locate the source. This technique is known as Adaptive Navigation. Such a technique could be expanded to other scenarios such as radiation-seeking, search/rescue operations, and even other types of environmental monitoring situations.

Adaptive Navigation involves the use of multiple sensors in a defined formation to generate the scalar values for a 2D or 3D field [4]. Once a gradient is calculated using all the sensor readings, the Adaptive Navigation technique moves the cluster closer to a target location. Adaptive Navigation is being explored by a number of researchers to find sources and navigate along contours in a variety of scalar fields. The RSL has extended these techniques to also

navigate along/to specific points of interest such as ridges, trenches, saddle points, and front lines [4]. An example of this technique can be seen in Figure 1.4, where a cluster of robots navigates to find a peak within a 2D temperature field. The cluster of three drones starts at the bottom left and moves to the peak by calculating the gradient at each step.

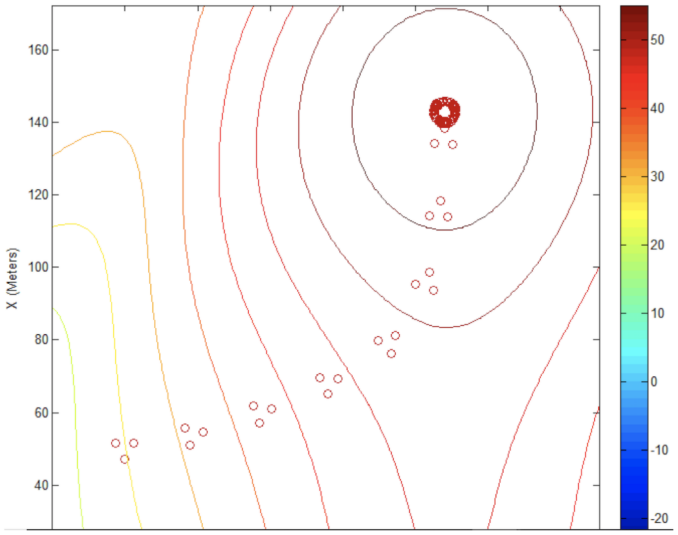


Figure 1.4: 2D Adaptive Navigation Example [4]

The clustering of robots has been pursued before and has been executed to variable degrees. Primarily, most clustering projects have existed in a two-dimensional plane [9]. This plane is normally well-documented and the robots are positioned to find a single point in that plane. An example of a project like this is the Decabots, a project within the SCU Robotics Systems Lab (RSL), where a variable number of rover-style robots form a cluster within a defined plane [10]. Using that cluster, the robots can take multiple sensor readings in parallel and perform Adaptive Navigation. However, three-dimensional Adaptive Navigation is rather limited. While there have been some projects that have explored this capability, it has not yet been achieved [11]. Its implementation, however, has many different potential advantages for applications such as environmental scanning, exploration, and search and rescue.

1.2 Project Objective

Our project aims to implement and test Adaptive Navigation and provide the necessary sensing, communication, and control systems for further development and research of Adaptive Navigation in three dimensions. The original objective for this project was to implement an end-to-end test bed for individual control, cluster control, and adaptive navigation for a four-drone system through a real-world field test. This constitutes using individual control to form a tetrahedral formation of four drones into a cluster that can navigate the terrain and calculated gradients based on RF sensor measurements to find the source of an RF beacon. This field test also depends on a multilayered control

architecture implemented through MATLAB and Simulink. This architecture involves three main control layers, the individual control layer, the cluster controller layer, and the adaptive navigation control layer, among which the pilot or operator can switch between seamlessly. The pilot or operator also have the option to decide which drone of group of drones are controlled by which layer and the drones are able to respond to both pilot commands and automatically generated commands from the next layer up. The last main objective was to ensure the implementation of all critical safety features to ensure both equipment and public safety when testing our system.

Due to the COVID-19 pandemic, we had to modify the objective related to the end-to-end test bed to use a simulation environment, which was not originally part of the plan. The need to quickly transition and merge our existing state-based controller in MATLAB with the new simulation environment resulted in the test bed being able to support up to a three-drone cluster that is able to be controlled as a unit or through the Adaptive Navigation control layer. This required the need to learn how to use and setup the simulation environment. This also involved integrating the controller with the simulation environment so that commands made through MATLAB changed the state of drones in the simulation.

Chapter 2

System Overview

2.1 Customer Needs

This system is being developed specifically for use with Santa Clara University's Robotic Systems Lab. As a result, we have followed their guidelines on project development and documentation. To be qualified to work within the RSL space, we have attended all training courses required to fully utilize the resources available within this space. This involved attending the Maker Lab and lithium polymer battery training sessions to gain access to the tools in the Maker Lab and the RSL's LiPo battery collection and charging station. Documentation of our project is stored on the project's Trac page and source code is stored in the lab's repository.

The system we have developed will be used to test the RSL's Adaptive Navigation project in a three-dimensional space. This project has previously been developed and tested in a two-dimensional space with the Decabots, and now a cluster of drones will be used to achieve a three-dimensional space. This will greatly expand the applicability of Adaptive Navigation to real-world issues.

In addition to following lab policies, it was important to use the lab's existing MATLAB/Simulink suite of control capabilities. The project also garnered the use of off-the-shelf drones from the RSL. The project was also to be implemented such that it could support simple scalar field creation/sensing, support pilot, on-board, centralized cluster, centralized AN control options. The implementation was to also allow for multiple pilots, multiple UAVs with single pilot, and comply with FAA rules, regulations, and laws.

2.2 Requirements

The requirements for this project were split into three main categories: functional requirements, nonfunctional requirements, and design requirements. The requirements for each of these categories are shown in the below tables.

Functional Requirements
Form and maintain a cluster of up to 4 drones.
Form cluster of 4 drones in a tetrahedral formation.
Maintain and hold formation with side lengths between 10 to 100 meters within +/- 3 meters.
Maintain 4 simultaneous 2-way links between drones and ground station.
Drone response must stay below a 10 second delay.
Drones must contain sensors to read scalar field values every 3 seconds.
Drones are able to send sensor information to the ground controller every 5 seconds.
End-to-end communication loop running at a 40Hz minimum.

Table 2.1: Functional Requirements

The functional requirements dictate the requirements associated with the functionality of the system as it runs. The functional requirements also relate to the specifics of the system's usage that follows the customer's needs.

Nonfunctional Requirements
Communication between a ground station and drones must be efficient.
The safety of drones and surrounding people/environment is to be maintained.
The performance of the system must be high.
Cluster is easy to control and maintain.
Targets and sources can be identified by cluster efficiently and quickly.

Table 2.2: Nonfunctional Requirements

The nonfunctional requirements depicts the customer needs in regards to the efficacy of the system in particular focus on the quality of the system.

Design Requirements
No individual drone within the cluster can be open to failure.
Safety must be ensured and batteries operated properly.
Cluster is not prone to large failures.

Table 2.3: Design Requirements

The design requirements restrict the system's usage from a user interface and design perspective.

2.3 High-Level System Design

2.3.1 Concept of Operation

Use of Adaptive Navigation requires a cluster of drones controllable in a three-dimensional space. The system we are designing allows for a ground station to control four octo-quad-copter drones as a cluster entity [3]. Figure 2.1 describes the high-level process our system will follow to be able to form a cluster for Adaptive Navigation to recognize and control. The first step in creating the cluster involves the operator manually taking off each drone and flying it into formation. Once the current drone is in formation, control can be transferred to the autopilot software to hold that position in the formation. After each drone has been added to the cluster, it can then be identified as a single cluster

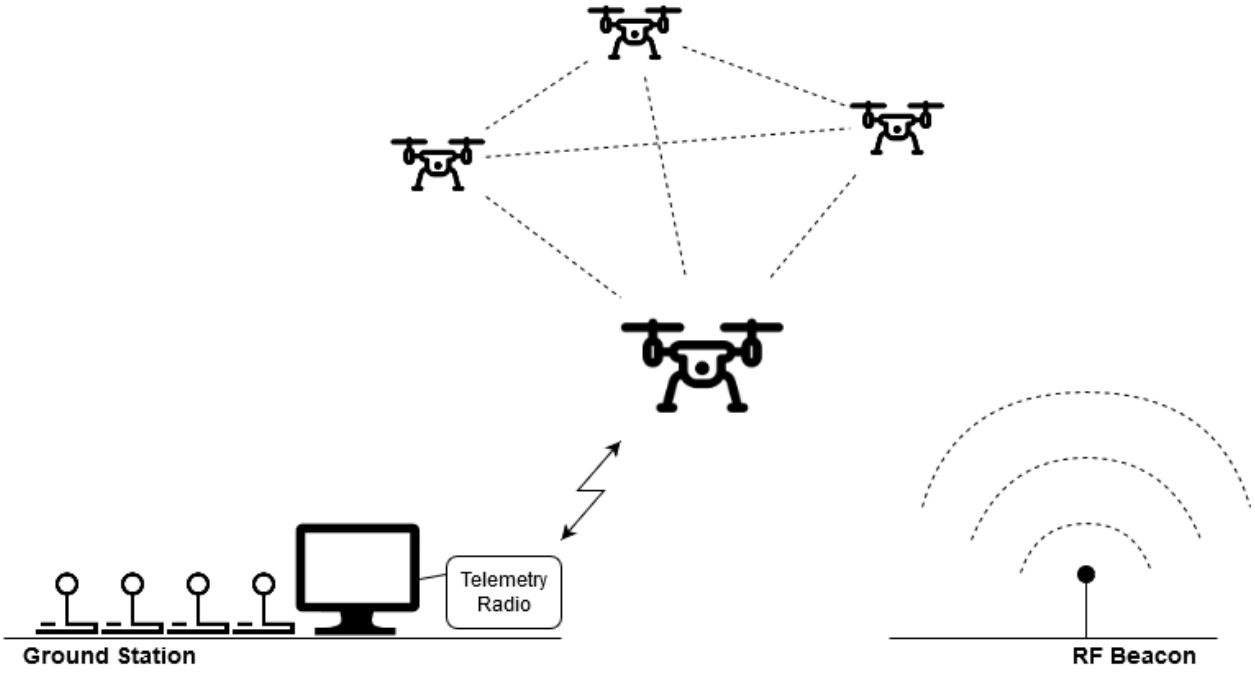


Figure 2.1: Conceptual Operation for 4-Drone Cluster

entity by the controller. This cluster of drones can then be directly flown by the operator to a designated starting point. The drone cluster can then be passed off to the Adaptive Navigation control system to complete the specified tasks. Once the Adaptive Navigation task has completed, the operator must take control of the drone cluster and fly it back to the landing zone. At this point, the operator takes control and lands each drone, until all drones have landed.

2.3.2 System Overview

To design our system specifically for Adaptive Navigation testing in a three-dimensional space, stable control of each drone in the cluster and valid sensor readings were required. In Figure 2.2, the high level architecture of the system implemented to achieve this is depicted. This system consists of three physical modules for complete operation: the ground station computer, the cluster of octo-copters, and the beacon. The ground station allows an operator to set the mode of control and generate flight commands for the drones. Each octo-copter collects and transmits sensor data to the ground station For this system to operate correctly, stable connection between each module must be established. The connection between each drone and the ground station handles the exchange of sensor data and flight control. Each drone is also recording its observed signal strength of the beacon during flight.

Our system is comprised of three main subsystems and relies on various forms of communication between these subsystems to keep the system operational. Our major subsystems outlined in Figure 2.3 consist of the drone and related hardware, a communication subsystem, and a control subsystem. To be compatible with our cluster controller,

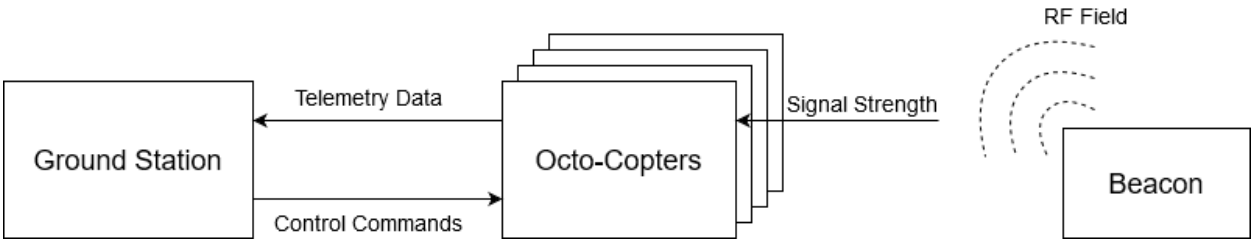


Figure 2.2: System Architecture

each drone must be equipped with a Pixhawk autopilot module and SiK telemetry radio. For testing purposes, our system uses four 3DR X8 octo-copters from the RSL. Movement control of each X8 is handled by an on board autopilot module and an Arduino Mega microprocessor has been installed to scan for RF signal strength. The communication subsystem includes an RF beacon and the ground station and handles the transfer of data throughout the system. To be able to test Adaptive Navigation capabilities in a three-dimensional situation, the beacon is used to create an RF field. The RF beacon is an Arduino Mega microprocessor broadcasting an RF signal using an XBee shield. The signal strength of this field is recorded by each drone to model a scalar field. The ground station handles all of the computation and individual control of each drone within the identified cluster. The control system was implemented using MATLAB and Simulink, a software suite within MATLAB, through the form of a state machine architecture. The next section outlines the communication process between the separate software components of the ground station.

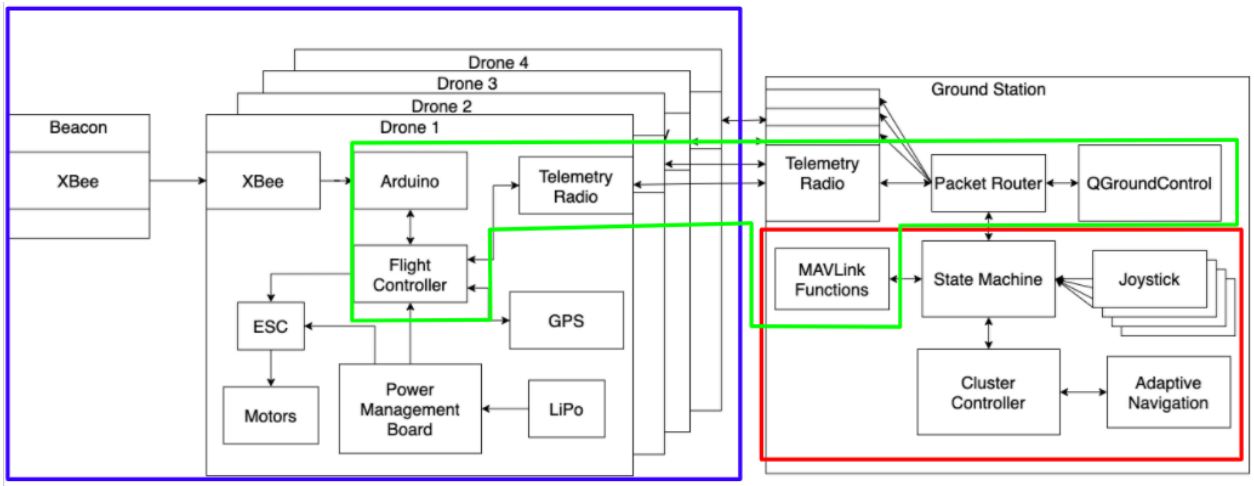


Figure 2.3: Component Block Diagram

2.3.3 Software Architecture

For our system, we are implementing a layered software architecture. Layering the separate modules will help limit coupling as much as possible, to ensure reliance on independent modules and easier validation. Validation was also

easier, as each layer was thoroughly tested before integrating with the adjacent layer. Additionally, we want to ensure that the drones are receiving the correct commands at all times, which is achieved by checking the command direction as they pass through the layers. Figure 2.4 displays how these layers are connected to control the flow of command packets within the system. All of the drones receive commands from and send telemetry data to the MAVLink Router, which acts as a router for all communication between the drones and the ground station [12]. QGroundControl is a flight control and mission planning software used to provide visual feedback to the operator, as well as enable direct control of any drone if required [13]. All command data packets are structured using the MAVLink communication standard and are generated within our Simulink Stateflow program. The Stateflow program generates these packets based on the current state. These commands can come from user control of an individual drone, control of the cluster, or Adaptive Navigation control of the cluster. Adaptive Navigation and the cluster controller generate the control information to be contained in the MAVLink packets based on which mode of control is currently active within Stateflow.

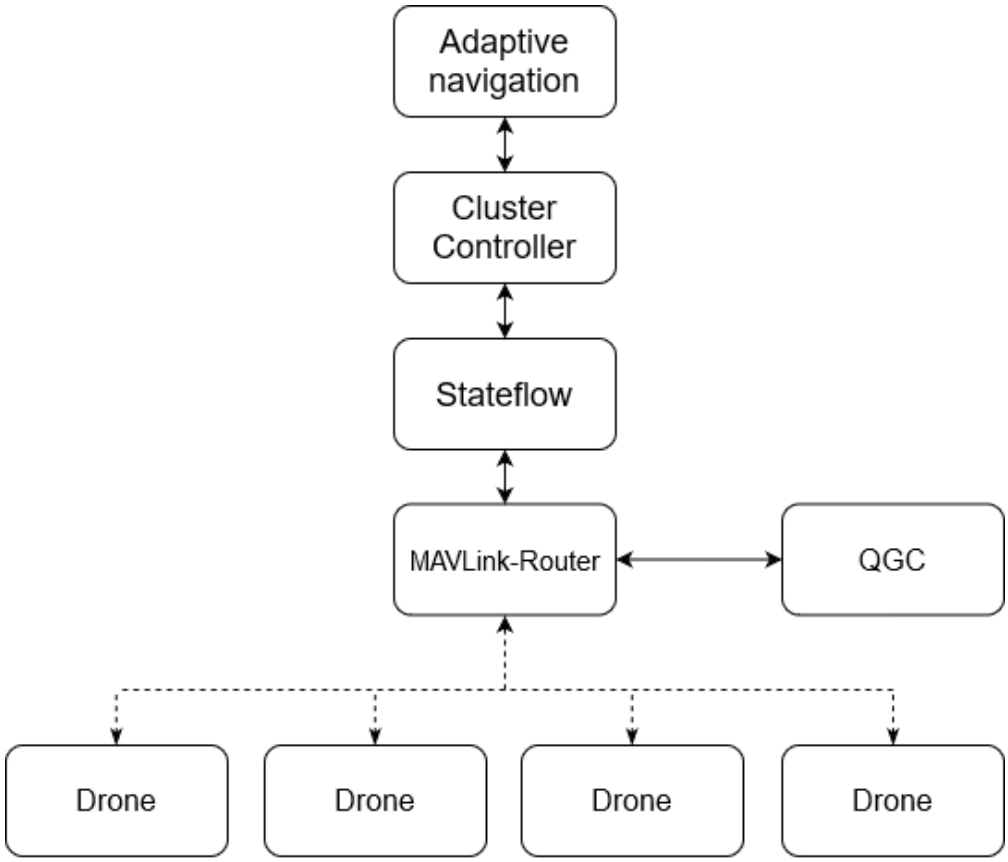


Figure 2.4: Software Architecture Model

2.4 Team and Project Management

2.4.1 Budget

We had a budget of $3500 for designing, building, and testing our project. We secured $1500 of internal funding from SCU's Robotics Systems Lab and received $2000 from the Undergraduate Programs of the School of Engineering for testing, FAA certifications, batteries, and upgrading the drones within the RSL. The upgrades to the drones were the most expensive component of this project as we upgraded four drones with new flight controllers, telemetry radios, and micro controllers for sensors. Additionally, we purchased multiple new joysticks and telemetry radios to pair with our ground station computer to effectively test and utilize our system.

Itemized Budget		
Item	Quantity	Price (Dollars)
FAA Pilot License	2	320.00
Pixhawk with GPS and Power Delivery	5	1249.75
Telemetry Radio	6	299.75
Xbee Shield	4	83.80
Xbee Radios	7	193.10
2.4 GHz Antennas	5	19.95
Arduino Mega	5	204.75
Joysticks	4	152.92
Game Controllers	3	60.35
Propellers	4 sets	87.96
Power Cables	1	9.95
9V Batteries	8	12.99
NiMH Batteries	8	219.96
XT-60 connectors	1	7.99
Standoffs	1	8.90
Total price		2932.12

Table 2.4: Budget Table

2.4.2 Project Timeline

The timeline depicted in Figure 2.5 illustrates the schedule of all the major phases within our project. Each task has been assigned to one of three major categories: Design, Implementation, and Documentation. Our design phase was exclusively during the Fall quarter of the academic year, and then it was not long until implementation began. Within implementation, there were a variety of development and testing tasks, causing this phase to be the most important in our project. Due to the nature of project and the challenges posed by COVID-19, much of the testing of our system had to be done well into Spring quarter of the school year. Finally, Documentation has been ongoing all year in order to effectively detail every step and aspect of the projects development and implementation.

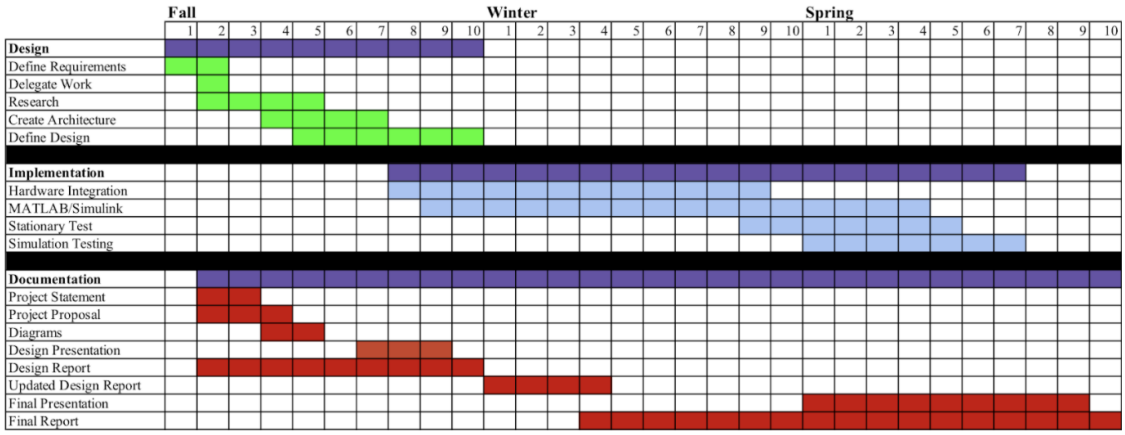


Figure 2.5: Project Schedule

2.4.3 Team Management

Zach Cameron

Zach focused on the integration of the overall system. As a result, much of his work was focused in multiple different areas. Zach began by designing the overall architecture of the system and helped to decide on the layered approach that was ultimately chosen. Zach worked to configure the telemetry radios for all of the drones and the ground station. Additionally, he explored methods to allow the control software the ability to send and receive MAVLink packets allowing for the ultimate control of the drones. He then finalized the manual control code to allow multiple drones the ability to be controlled by multiple controllers on the same system Furthermore, he collaborated with Aditya to help finish and test the cluster control equations for a two drone and three drone system. Finally, Zach integrated the Adaptive Navigation code base from the Robotic Systems Lab as the final layer of the control architecture to allow for two dimensional Adaptive Navigation with a three drone system. He then created a simulation to show final results of our system.

Brendan Engh

Brendan's main focus for this project was on the hardware which included the drones, radios, scalar field sensor and beacon, and the power management system. Brendan began by identifying each of the hardware components necessary to make a flight ready system to perform Adaptive Navigation. This involved integrating and configuring each of the drones and components to act as one cohesive unit. He was responsible for examining four of the RSL's old 3DR X8 octo-quad-copters and equipping them with all our new components. This also include insuring proper power management. Brendan worked closely with Thomas to develop, calibrate, and test the scalar field sensors and they performed many of the stationary tests once the system was fully integrated. Additionally, Brendan completed necessary safety forms and pre-flight checklists as well as receiving his FAA Part 107 drone license.

Thomas Kambe

Thomas spent the first months of the project researching the structure and hierarchy of MAVLink packets. Additionally, he parsed the specific structure and contents of several sets of packets between a drone and QGroundControl. From this, the messages within the MAVLink 2 protocol used for data transfer were determined and implemented on all devices. Thomas was also key in integrating the MAVLink Router within the system to ensure proper routing between the telemetry radios and the ground station computer. Thomas worked with Brendan to integrate and test each scalar field sensor on each drone. Once the drones were ready, end-to-end communication between MATLAB and the Arduino was tested with Aditya. Once communication between the drones and ground station was established, Thomas implemented a standard set of messages for sensor data request and collection. In the final phase of the project, Thomas and Brendan worked together to perform system testing of a 2-drone and 3-drone system.

Aditya Krishnan

Aditya began by analyzing how a state diagram chart can be accomplished within MATLAB and Simulink using Simulink's Stateflow. He focused on building our state controller using Stateflow within Simulink and MATLAB for a 2-drone, 3-drone, and 4-drone setup for individual control and cluster control. Other work includes ensuring communication between the joysticks and MATLAB was functioning as required. Also, communication between MATLAB and the other functional components of the overall system was confirmed with Thomas. In addition, the forward and inverse kinematic relationships were modeled, implemented, and tested prior to merging with the state-based controller.

Chapter 3

Subsystem Design

3.1 Drones

3.1.1 3DRobotics X8

The Drones used for this project were four of the now discontinued 3DRobotics' X8 octo-quad-copters. The octo-quad-copter configuration includes four arms connected to the central hub with 2 motors on each arm, 1 motor faces up with a clockwise propeller blade, the other faces down with a counter clockwise propeller blade. The octo-quad configuration was used due to availability and it provided the proper lift to weight characteristics for stable flight with added weight from the sensors. Another key reason these drones were used for is that are expandable due to the open and flat frame that allows for mounting of almost any small to medium sized sensor.

Four of these drones were provided by the RSL, however, each drone needed significant upgrades and modifications before they could be used within our system. Each of the drones had previously been used with past RSL projects so the first step in setting up the drones was to remove all the unnecessary sensors and connections. Two of the drones provided can be seen in Figure 3.1. Although all the extra sensors were removed, each of the drones still had many functional components, but they still needed to be integrated with our flight controller.

This included the chassis, the power management board, each of the 20A ESC motors, and a XT-60 battery connector and voltage regulator. In addition to removing and upgrading multiple hardware components, some of the drones were missing screws, nuts, and standoffs. These items were purchased and included where applicable to ensure the structural integrity of the drone and all its parts.

Figure 3.1: Two 3DR X8 provided by the RSL

3.1.2 Flight Controller

The first decision within this project was to figure out which flight computer would work best for our system. We decided on the Pixhawk 4 shown in Figure 3.2a due to it being the newest of the Pixhawk family and being the best flight controller on the market for most drone projects. The Pixhawk 4 is a popular flight controller within robotics and is optimized to run PX4 which was our chosen autopilot software. Along with meeting all our of our needs and its suitable online documentation, the Pixhawk 4 has faster processing than the outdated 3DR Pixhawk 1 that came with each of the drones. The Pixhawk 4 came with all the necessary ports needed for GPS, telemetry radios, ESCs motors, and multiple other ports for a wide range of different sensors. The Pixhawk 4 came prepackaged with the GPS module show in Figure 3.2b and installation was as simple as plugging it into the Pixhawk's GPS port. This GPS module also provide us satisfactory location with +/- 3 meters.

Pixhawk 4	
+	-
Faster	Cost
More memory	New cables
Quick disconnect ports	Required tear down
Included GPS	
PX4 support	
Documentation	

Table 3.1: Pixhawk 4 Pros and Cons

The next consideration made was to decide on how we would communicate with our ground station. We noticed the older 3DR X8's and other projects within the RSL used 915MHz SiK telemetry radios so we decided to buy a pair of the updated version 2 SiK radios and tested their reliability. These radios are shown in Figure 3.2c. The v2 SiK telemetry radio supported the latest MAVLink communication protocol that we would use for the rest of our projects communication where as the previous telemetry radios on the X8's did not. For a one drone system these radios were also as simple as plugging them in to the Pixhawk's telemetry port to get communication between our drone and ground station software. We would later need to configure each radio to allow for multi-channeled communication for four drones. The Pixhawk 4 and PX4 autopilot provided all the necessary features and due to their open source nature their online documentation was critical in integrating our sensors, testing, and troubleshooting.

3.1.3 Autopilot

After deciding upon using the Pixhawk 4 flight controller for the on-board processor of each drone, the autopilot firmware needed to be chosen. The two most popular solutions considered for this project were Ardupilot and PX4. While Ardupilot offers safer autopilot navigation of a single drone system, PX4 proved able to support multiple drone instances with more stability. Usage of the PX4 flight stack also made interfacing with QGroundControl convenient

SiK v2 and GPS	
+	-
MAVLink 2 support	Costs
915 MHz	Difficult configuration
Range	
Documentation	
Accurate enough GPS	
Built in compass(GPS)	

Table 3.2: SiK v2 and GPS Pros and Cons

(a) Pixhawk 4 (b) GPS Module (c) Telemetry Radios

Figure 3.2: Components

as the ground station program was developed to work directly with the PX4 firmware. This flight stack serves as the direct controlling processor of each drone. Incoming control commands are parsed by the autopilot module into signals sent to the ESC modules and GPS readings are collected through the Pixhawk's serial interface.

3.1.4 Scalar Field Sensor

To test Adaptive Navigation to locate an RF source, it was necessary for there to be an RF signal beacon and an RF signal receiver on each drone. It's also important that as the drones move closer to the beacon their signal strength would increase. For the RF transceiver we decided to use another popular radio in the RSL, the XBee radio module. We choose the XBee3 Pro 2.4 GHz radio because it was able to output it's received signal strength indicator(RSSI) and the 2.4 GHz frequency wouldn't overlap with our 915 MHz telemetry radios. The radio is also small, lightweight, and low power, whilst also giving us plenty of range. Another useful feature of the XBee is it could output its RSSI value either through appending the value to one of its packets on a UART serial connection or outputting a PWM signal through one of its pins. After we were able to configure the XBees using XCTU, a XBee firmware flasher software, we built our beacon. Using an Arduino, programmed to continuously send packets, a XBee shield and an existing

XBee Arduino library we completed our Beacon that would create our RF scalar field. This beacon can be seen in Figure 3.3 The XBee connected to the drone, however would prove to be more difficult.

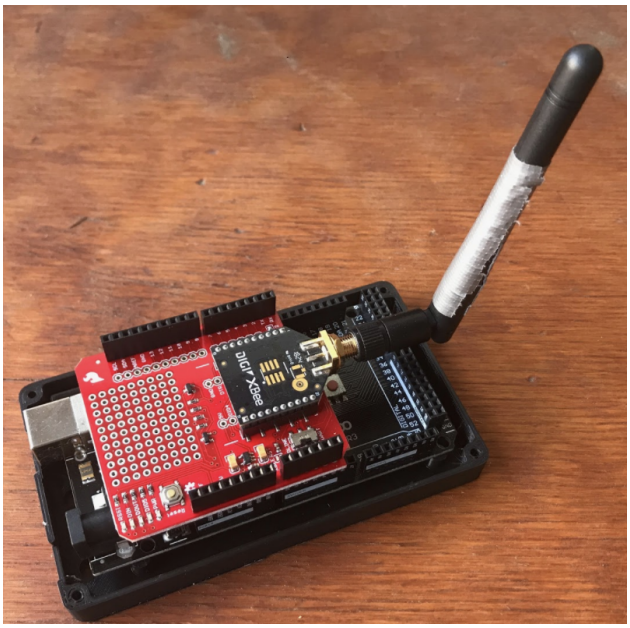


Figure 3.3: Scalar Field Beacon

The Pixhawk 4 and PX4 firmware is pre-configured to be compatible with many types of radio transceivers and with a large range of sensors, however, because we would be using a radio module that wasn't already configured in the PX4 firmware, we had to go through many design iterations. We knew it was possible to send our signal strength values to our ground station without communication through the Pixhawk but we didn't want to develop a separate communication link due to concurrency issues and developmental overhead. It was also important not to change the firmware as we wanted to keep the drones firmware update-able for future expansion and research using our project.

Initially we tried to directly input the PWM signal from the signal pin directly into the Pixhawk but it was quickly realized without changing the Pixhawk firmware this wasn't possible. After this we thought to set up a UART serial connection from the XBee to the Pixhawk but this would also require changes to firmware to unpack the signal strength value and then repackage it to send it through our MAVLink communication protocol. After some thought and research into the PX4 documentation we found that we could link a companion computer to the Pixhawk that could communicate through serial using our MAVLink protocol. The Arduino platform was chosen due to its ease of use and its expanse of libraries that can enable almost any sensor for Adaptive Navigation research. This capability simplifies the integration of new sensors into our system.

We initially settled on the Arduino UNO but later switched to the Arduino Mega, as our Arduino program became to large for the UNO. The Mega also offered us an extra two serial ports. The first port would be used to communicate

with the XBee radio, the second port would be used to connect to the Pixhawk and the third could be used for an additional sensor. Our code would allow us to unpack the XBee packets defined by IEEE 802.15.4 and the RSSI value would be collected and repacked into a MAVLink packet and sent to the Pixhawk, which would subsequently make its way to the ground station. To get a more reliable RSSI values we decide to take a moving average of the last 3 RSSI values and send that value to the ground station. In Figure 3.4 the final design can be seen with the XBee on top, connected to an XBee-Arduino shield, which is then slotted into the Arduino Mega with serial wires from the Pixhawk connected to the Arduino.

Figure 3.4: Scalar Field Sensor

3.1.5 Power Management

A critical component to each of the drones was powering the Pixhawk, the ESC motors, and the Arduino. Switching from the older Pixhawk 1 used by the 3DR X8s, to the newer Pixhawk 4 required changes to wiring. Because the newer Pixhawk 4 uses a new connector that allows for easier changing of pins, it was required that a drone was fully stripped down to reveal all the interconnects and wiring. Even though each of the Pixhawk 4 kits came with their own new power management board, we decided the time required to change and solder each of the new boards to the existing components wasn't practical when the older power management boards still provided all the necessary functionality with minor tweaks. A picture of the stripped down 3DR X8 can be seen in Figure 3.5. Once this was done it was noticed that the original power management board would likely still work to power each of the ESC motors but a specific 8 pin JST to 8x3 pin Dupont adapter was needed to connect each ESC motor to the Pixhawk(shown as the top circle in Figure 3.5). The tear down also revealed a 6 pin JST-GH to 6 pin DF13 was needed for the voltage regulator to connect to the newer 5V pixhawk power port(shown as the bottom circle in Figure 3.5). Once the first drone was built a battery was connected to verify that all the components powered on.

The main power supply for all the drones would be a 20000mAh 4S lithium polymer (LiPo) battery connected through an XT-60 adapter, but once we had to start working from home due to safety concerns we switched to a less volatile

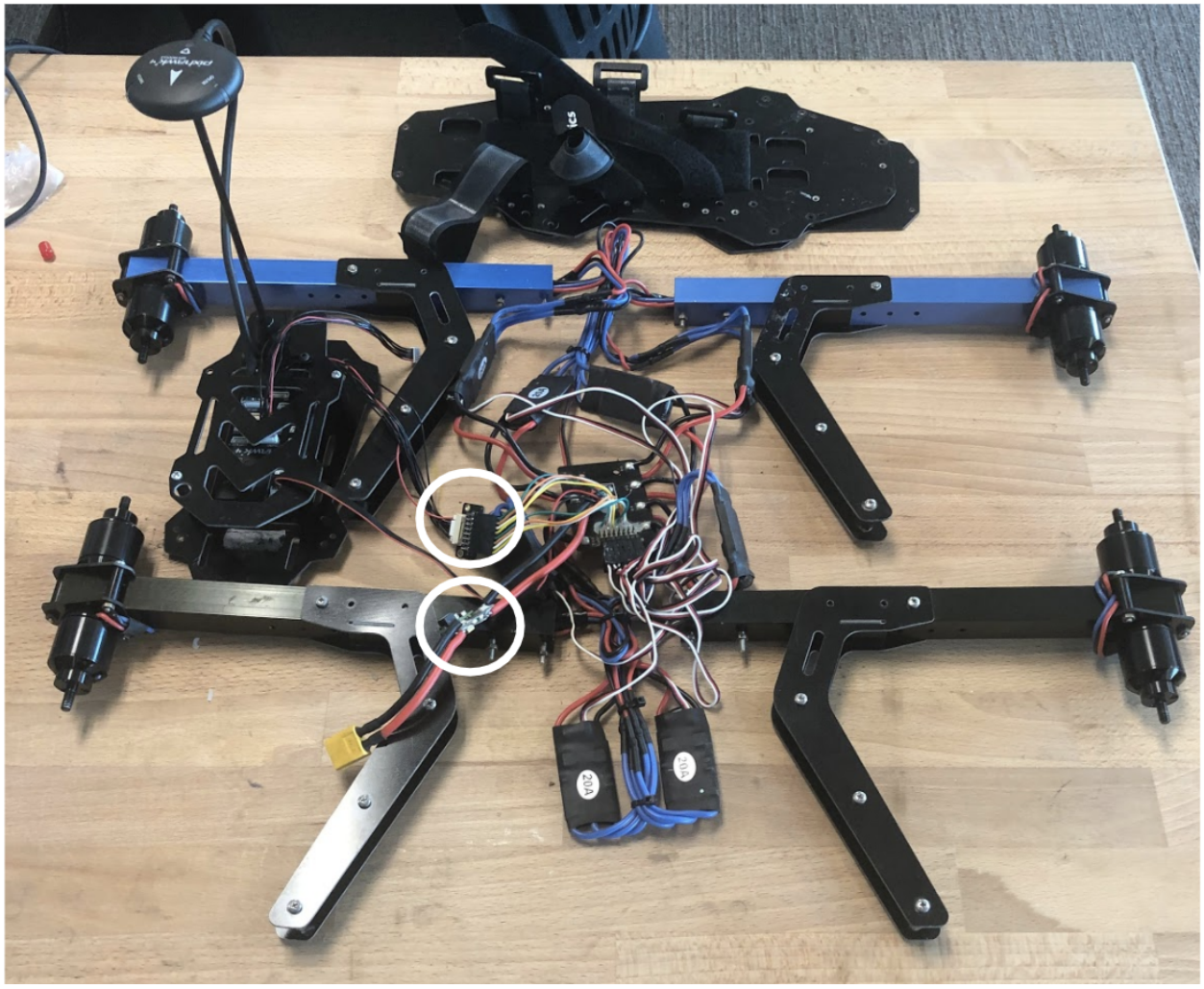


Figure 3.5: Tear down of the 3DR X8

nickel–metal hydride(NiMH) battery. To achieve similar characteristics to the LiPo we connected two 6S 3800mAh NiMH batteries in series. LiPo batteries are the battery of choice for most drone systems due to their high energy density and high discharge rates. NiMH are less energy dense so they would unlikely be used for any flights but they also provided the necessary discharge rates for our drones. Both batteries were used for stationary tests in different stages of the project.

The last component that needed to be powered was the Arduino Mega. Instead of trying to connect another voltage regulator in series with our main power cable to provide the needed 9 volts to the Mega, we decided to use a simple 9V battery. The 9V battery holder was placed below the Arduino Mega and attached to the main frame of the drone. This battery case also came with a convenient on/off switch for when drones are being stored. The completed drone that was built is shown in Figure 3.6

Figure 3.6: The first completed drone

3.2 Communication

Stable communication to each drone is essential to safe operation of the cluster. Our implemented network focuses on ensuring all drones actively send and receive all data necessary for operation. Achieving this requires a constant stream of telemetry data from the drones and control updates from the ground station, and the MAVLink 2 protocol proved to be the best option for the structure of this data. Our network consists of six endpoints and a shared message protocol intended for drone usage that allows for all of this data to be easily accessible. We also utilize a router for packet routing between the drones and ground station. These six endpoints are QGroundControl, MATLAB, and each drone in the cluster. Figure 3.7 shows each of these endpoints as well as the direction and type of packets exchanged between these points. Each endpoint begins by broadcasting heartbeat packets to identify other MAVLink devices within the network. A heartbeat packet consists of the sender's unique identification, the version and dialect of MAVLink being used, and the type of device. Once each device has registered the other devices, data can begin to be transferred between the drones and ground station. Packets can only be sent to registered devices. If a drone does not respond to the initial heartbeat message from the ground station, the control program will not proceed. The on-board Pixhawk

for each drone handles packaging and transmitting all sensor data. In our case, this sensor data involves readings from the Pixhawk's IMU, the connected GPS module, and the scalar field sensor. This telemetry data is recorded and transmitted at a constant rate, specified by the ground station sending the initial request command. Telemetry data from all drones is received within MATLAB to generate the next set of control commands. This data is also sent to QGroundControl so that the operator can view this data in a graphical interface. The control subsystem running within MATLAB handles generating all control commands, which are then transmitted to each drone through the router. The Pixhawk on each drone handles accepting these control packets or routing them to the attached sensors. This section will go into further detail on the functionality and packet structure of each of our endpoints within this network.

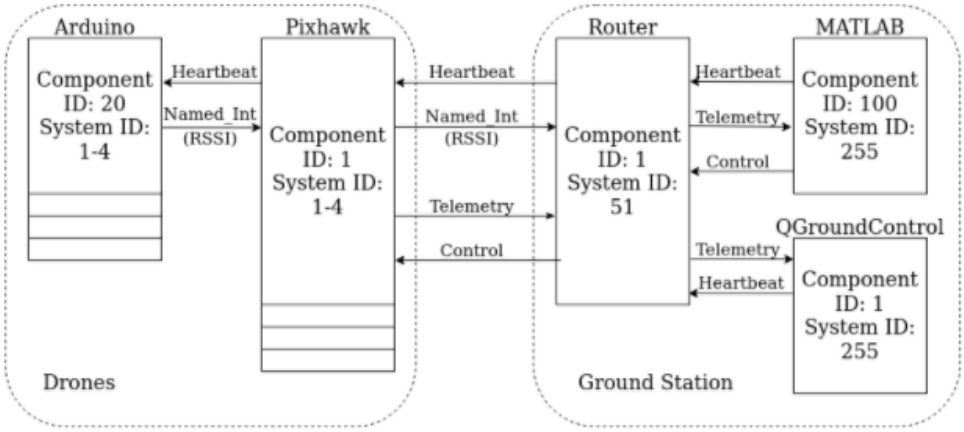


Figure 3.7: Packet Flow Within Communication Network

Currently, there are two versions of the MAVLink protocol available that can be used interchangeably. The blocks within a MAVLink packet are shown in Figure 3.8, where the highlighted blocks represent data specific to the version 2 protocol [14]. Updates from version 1 to version 2 include the addition of a version flag and a signature attached to the end of the packet. The shared ordering of these sections allow for any device using the MAVLink 2 library to accept and parse MAVLink 1 packets. MAVLink 2 packets follow the same structure as version 1, with an added field for version number and a signature attached to the end of the packet for additional authentication. MAVLink 2 packets also have a larger payload length to allow for more unique types of messages within the dialect. The sections of each packet that is relevant to our project are the system ID and component ID, payload, and signature.

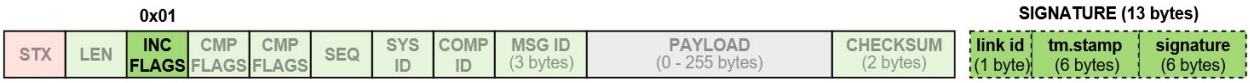


Figure 3.8: MAVLink Packet Contents

Every device in our network has a unique system and component ID pair which is used to identify each device. Our ground station subsystem is assigned system ID 255 and each drone is assigned system ID 1 through 4. Within our ground station, MATLAB signs its packets with component ID 100 and QGroundControl uses component ID 1. Each drone contains two MAVLink enabled devices. The Pixhawk runs at component ID 1, which is reserved for autopilot modules. The scalar field sensor uses component ID 20 so that the Pixhawk can route packets to it. By using a shared system ID for each physical subsystem and separate component IDs for each device within that subsystem helps shift some of the routing processes from the MAVLink-router to each of the Pixhawk devices.

The payload section is filled with the fields specific to the message being sent, where all messages contain different fields within the 255-byte block related to the type of message being used. Most telemetry messages being used will identify the type of sensor by the message ID, then include the mode of operation, sensor readings, and destination system and component ID in the payload block. The payload can be set to either be a message or a command. Messages deal with data transfer and most do not require a return acknowledgment.

Commands are a subset of messages that are part of the expanded version 2 functions. To use commands, the message ID is first set to COMMAND_LONG or COMMAND_INT based on the size of the command. Within the message fields for a command message, the fields specific to the type of command are filled similar to the payload for a message. A command ID is also set within the payload to specify which type of command is being used. In our project commands are directly used for all states of drone control, setting the version number, requesting scalar data, and subscribing to data transmissions. Commands are also used by the autopilot module and QGroundControl to ensure all physical components of the drone are functioning properly. Two of these functions that are carefully monitored are battery status and GPS link status. If the battery drops below a set threshold, the drone will begin a landing sequence. The GPS fail safe was disabled for stationary testing, but is configured to set the drone to a hold position mode until connection is regained.

The signature is specific to MAVLink 2 packets and accounts for most of the size difference between version 1 and 2 packets. This part of the packet is usually parsed by the receiving device before any of our programs receive the packet. The timestamp on packets is always compared against the value of the previous packet received from the same sending device before the receiving device accepts the packet. Instead of attempting to optimize receiving buffers to minimize packet loss, speed of processing packets was prioritized. In a real-time system such as ours, constant data is flowing and receiving the next update is more important than organizing and processing the complete data flow. Packet loss is still a very important factor to be considered in our design and issues related to it are addressed within the MAVLink-Router section.

Before each device can begin sending and receiving packets related to its specific function, it must first register with all other MAVLink enabled devices. This process involves two steps: broadcasting a heartbeat and identifying the version of other devices. Upon powering up, every device utilizing MAVLink begins broadcasting heartbeat packets. This packet contains basic information about the device including the system and component ID, the type of device, and the system status. Once a device receives the first heartbeat from another device, that device's information is registered in the receiving device. This allows each device to remember information about other devices for more efficient authentication of packets and the immediate connection routing involved in sending to each device. After registering other devices as MAVLink enabled endpoints, the version of the packets must be determined.

3.2.1 Ground Station

The ground station consists of three MAVLink enabled devices. QGroundControl is running as a visual display of the current state of each drone in the cluster as well as performing safety checks. Our control program is running in MATLAB to provide control and receive feedback from each drone. MAVLink-router is also running on the ground station to provide separate communication channels to each drone.

MATLAB Program

All packets sent from MATLAB are indicated by system ID: 255 and component ID: 100. As a MAVLink device, the control program's function is to receive data from each drone and reply with commands for each drone's next movement. The program in place to process this data and produce the appropriate control is detailed in the Control section below. After MATLAB has sent out heartbeats to detect each MAVLink device, all devices with sequential system IDs starting at 1 are recognized as drones. A command is then sent to each drone to subscribe to telemetry data. This request causes each Pixhawk to respond with orientation and position readings at a specified interval. For control of the drones, three separate commands are implemented. For direct operator control of individual drones or the cluster, MANUAL_CONTROL commands are populated with joystick values and sent to each drone. The Pixhawk then parses these commands to determine how to drive the motors to achieve the desired motion. Any drone put in hold mode during direct control is sent the command to hold current position. When Adaptive Navigation is controlling the cluster, commands are sent to control the velocity and direction of each drone.

MAVLink-Router

To connect each drone to the control program, a line of communication between these subsystems must be established. For communication between the drones and ground station, the Pixhawk relies on a serial connection to the telemetry radio. A UART connection to the telemetry radio on the ground station allows a program to exchange packets with this Pixhawk. During early testing, it was determined that a single radio link would not be sufficient for stable connection

to multiple drones. After implementing a radio pair per drone, a router was needed to connect all radio channels to the control program and QGroundControl. For this connection interface, the open-source MAVLink-router [15] was used. This program is a simple router that connects endpoints and routes MAVLink packets between these endpoints. Each USB port that the telemetry radios are plugged in to is connected to a UART port within the router configuration. Separate UDP ports are opened to connect to QGroundControl and MATLAB to the router as well. The baud rate for all ports is set to 115200 to mitigate packet loss. While the MAVLink-router offers a compact router capable of connecting the required devices, its routing algorithm is not complex. Each of the six connections is identified as an endpoint to MAVLink-router, so packets are forwarded to other endpoints without much use of the system ID and component ID.

QGroundControl

During drone operation, a visual display of the status of the drones is very useful for the pilot. QGroundControl is a ground control and mission planner software available to interface with autonomous drones using the MAVLink protocol. By feeding the telemetry data to QGroundControl and minimizing the control output, the graphical interface of this program is used as our display. Many of the parameters related to sending commands to the drones were disabled, while safety features capable of overriding the drones were kept in place.

3.2.2 Drones

There are two separate MAVLink devices within each drone. These are the Pixhawk unit that handles direct control of the drone's motors and sensor readings and the Arduino Mega which collects scalar values. Once the drone is powered up, the Pixhawk begins broadcasting heartbeats to identify itself in the network. Once the control program is running, the Pixhawk will begin its response of telemetry data and awaits commands. The scalar field sensor handles its data transmission with a different exchange process. To be able to compile the MAVLink library on an Arduino, the version 1 protocol had to be implemented due to memory constraints. This did not cause any issues within our network as all other devices are operating at version 2 and the exchange between the Arduino and ground station relied on version 1 messages. Each time a heartbeat is received from the control program on the ground station, the most recent scalar data is averaged and transmitted in a Named_Int message.

3.3 Control

The control architecture used in this project is multilayered and configurable. The multilayered nature of the control architecture is split into three main layers: an on board drone control layer that handles platform stability and executes position-hold and position-drive commands, a formation controller which controls the geometry and aggregate motion of a group of drones, and the Adaptive Navigation layer which performs source-seeking and other "primitive"

functions. This multilayered approach to the control architecture allows for the architecture to also be configurable. Configurability of the architecture refers to the pilot or operator being able to specify which drones are controlled by which control layers and the input to any control layer can be either a pilot command, i.e. the pilot flies a drone or the cluster, or an automatically generated command from the next layer up.

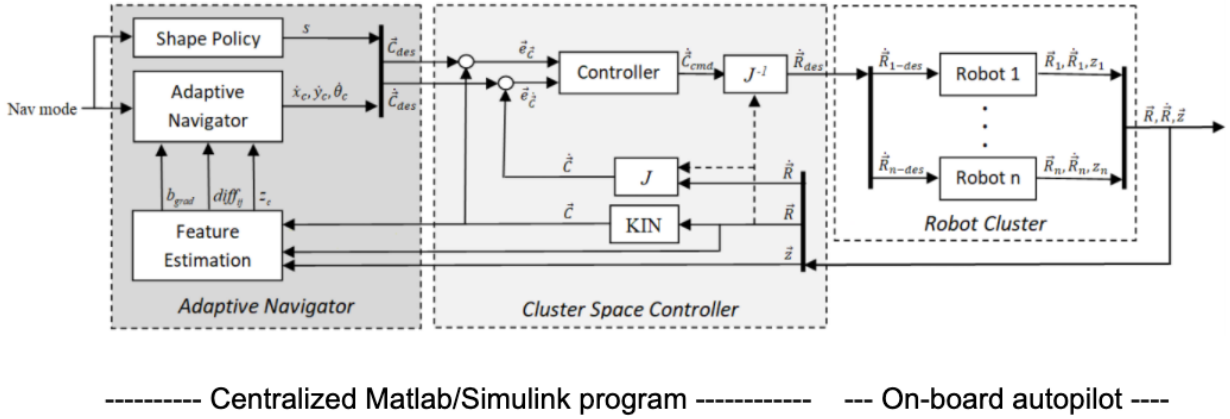


Figure 3.9: Control Architecture Hierarchy [4]

The diagram in Figure 3.9 shows the control architecture with the three control layers [4]. It is important to remember that the operator can mix/match which of these apply to which drones. On the right there are the individual drones with an interface of drone level motion commands. In the middle, the cluster space controller is depicted to control the cluster so that a single pilot can fly the cluster given the use of that layer. This layer's primary purpose is to keep the formation in the same geometry by utilizing kinematic equations to translate between robot and cluster geometries and velocity commands. On the left, there is the Adaptive Navigation layer that may ask for a specific formation geometry or for the geometry to change which is carried out by the cluster layer in the middle. The Adaptive Navigation layer also analyzes position and sensor data from each drone and then issues the drive commands to the cluster to, for example, follow a gradient. Drone control is done by the on board Pixhawk autopilot for pitch/roll stability and to execute velocity or hold position commands. MATLAB/Simulink exists on a centralized workstation and provides cluster formation and/or Adaptive Navigation control.

The diagram in Figure 3.10 depicts the state diagram for a 2-drone system. Pilots can specify what control layers apply to which drones and a state machine formally manages this control configuration. In particular, state machine diagrams depicting the resulting system states and transitions for 2-drone, 3-drone, and 4-drone systems were implemented in MATLAB's Simulink. This includes calculating and transitioning to the cluster space going from individual robot states. This also involves performing detailed calculations to transfer from the robot space to the cluster space by placing a single drone in the set of drones as the center of the cluster frame.

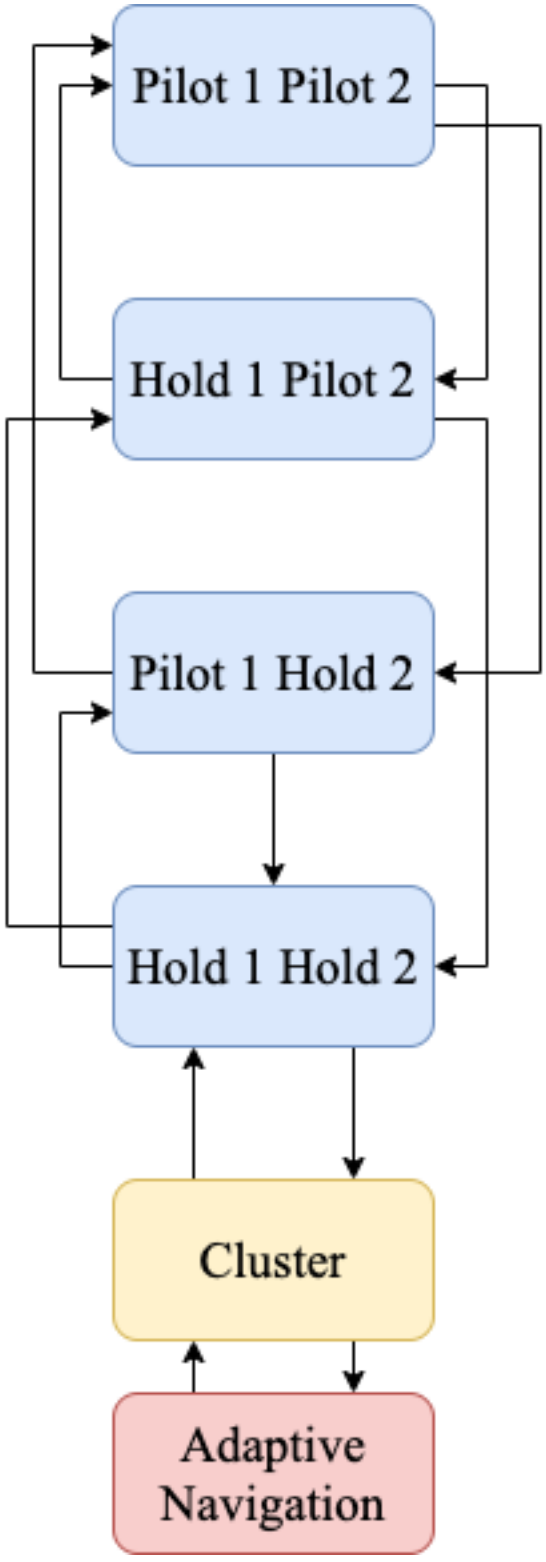


Figure 3.10: State Diagram for 2-Drone System

For a particular n-drone system, the number of system states that exist within the state machine is equivalent to $2^n + 2$ states, when considering the system states corresponding to individual control and the cluster control state. The initial controller contains 2^n states to encapsulate all permutations of vehicle states. A system state refers to the state of the system of all drones at an instance of time. A vehicle state corresponds to the state of a single drone within that system state. This vehicle state in the case of the individual control alone is split between either a hold state or a pilot state. The hold state corresponds to the case when the controller, i.e. the joystick, toggles the autopilot on a drone to hold the current position of the drone. The autopilot on the drone will make corrections to the velocity commands sent to the drone automatically to ensure that the drone holds the current position of the drone with little uncertainty. The pilot state allows the joystick to individually control a drone in whichever way the pilot wishes to control the drone. This includes x, y, z, and yaw commands sent from the joystick through MATLAB to the Pixhawk autopilot on the drone. The extra two states arise from a cluster state to build the cluster among the drones and the adaptive navigation state to calculate gradients and send commands through the cluster space controller to control the drones in the cluster.

One of our main design goals for this project were to be able to form and maintain a cluster formation, while also being able to control the cluster and move it with a joystick. To implement this design goal a cluster controller was built in MATLAB and Simulink. This is done through formal kinematic transformations relating drone positions to cluster positions and the drone velocities to cluster velocities. Each drone has 4 controllable degrees of freedom (DOF) containing the position (x, y, and z) values and a heading value (a θ) for the drones with respect to the global reference frame that is set. The cluster frame origin corresponds to the position and heading for the cluster origin, which is set to those values for the first drone in the cluster in the case of a leader/follower relationship. The cluster frame also depicts the distance values in the x-y plane (named dx), the z-axis (named dz), and the θ-axis (named ϕ) of the n-th drone in an n-drone cluster from the first drone. The exact equations used to translate position values for a single robot to its cluster geometry position and from its cluster position to its robot frame position are shown below. The equations translating robot position values to cluster position values are named the forward kinematics, and the opposite going from the cluster frame to the robot frame are named the inverse kinematics.

For a two cluster formation, as an example, we formally define R, the Robot Space position vector, as being a vector of the drone positions, as shown in Equation 3.1. Similarly, we define C, the Cluster Space position vector, as being the vector of the cluster parameters that define the location and geometry of the cluster, as shown in Equation 3.2.

$$R = [x_1, y_2, z_2, \theta_1, x_2, y_2, z_2, \theta_2] \tag{3.1}$$

$$C = [x_c, y_c, z_c, \theta_c, dx, dz, \phi, \alpha] \tag{3.2}$$

The forward kinematic equations are provided by Equations 3.3- 3.10.

Forward Kinematics:

$$x_c = x_1 \tag{3.3}$$

$$y_c = y_1 \tag{3.4}$$

$$z_c = z_1 \tag{3.5}$$

$$\theta_c = \theta_1 \tag{3.6}$$

$$dx = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.7}$$

$$dz = z_2 - z_1 \tag{3.8}$$

$$\phi = \frac{\pi}{2} - \theta_1 - \tan^{-1}(\frac{y_2 - y_1}{x_2 - x_1}) \tag{3.9}$$

$$\alpha = \theta_2 \tag{3.10}$$

The inverse kinematic equations are provided by Equations 3.11- 3.18.

Inverse Kinematics:

$$x_1 = x_c \tag{3.11}$$

$$y_1 = y_c \tag{3.12}$$

$$z_1 = z_c \tag{3.13}$$

$$\theta_1 = \theta_c \tag{3.14}$$

$$x_2 = x_c + dx\cos(\frac{\pi}{2} - \theta_c - \phi) \tag{3.15}$$

$$y_2 = y_c + dx\sin(\frac{\pi}{2} - \theta_c - \phi) \tag{3.16}$$

$$z_2 = dz + z_c \tag{3.17}$$

$$\theta_2 = \alpha \tag{3.18}$$

Given these position relationships, a set of velocity relationships between robot-specific velocities and cluster-specific velocities can be obtained. This is done by taking the derivatives of the sets of Equations 3.3- 3.10 and 3.11- 3.18. This will lead to the relationships shown in Equations 3.24 and 3.25. As can be seen the velocity relationships can be factored into a matrix-based equation where the matrix J is known as the system's Jacobian matrix. Given that the cluster controller computes new cluster space velocity set-points, Equation 3.25 can be used to convert these to robot velocity commands that are ultimately relayed to each individual drone.

Cluster Velocity:

$$\dot{C} = [\dot{x}_c, \dot{y}_c, \dot{z}_c, \dot{\theta}_c, \dot{dx}, \dot{dz}, \dot{\phi}, \dot{\alpha}] \tag{3.19}$$

$$\dot{dx} = K_{dx}(dx_{desired} - dx_{actual}) \tag{3.20}$$

$$\dot{dz} = K_{dz}(dz_{desired} - dz_{actual}) \tag{3.21}$$

$$\dot{\phi} = K_{\phi}(\phi_{desired} - \phi_{actual}) \tag{3.22}$$

$$\dot{\alpha} = K_{\alpha}(\alpha_{desired} - \alpha_{actual}) \tag{3.23}$$

where $\dot{x}_c, \dot{y}_c, \dot{z}_c, \dot{\theta}_c$ are set by the commands received from the joystick.

The cluster velocity and robot velocity commands are calculated using the following equations:

$$\dot{C} = J\dot{R} \tag{3.24}$$

$$\dot{R} = J^{-1}\dot{C} \tag{3.25}$$

where J and J^{-1} are the Jacobian and inverse Jacobian, respectively, based on the forward and inverse kinematics equations detailed in Equations 3.3 to 3.18.

For this study, we have used a leader-follower approach in order to define the cluster. Given this, the pilot effectively controls the leader drone with all other vehicles following the leader at commanded distance and angle offsets. Those offsets may be changed if desired. Figure 4.8 on Page 45 shows the two cluster geometry used to define this leader-follower policy. It is noted that other formation definitions may be defined, in which case a different set of kinematic transforms would be in effect.

Movement of the cluster involves using the first joystick to control the cluster as a unit in any direction dictated by the pilot/operator. After having controlled the cluster as a unit, the pilot/operator is given the option to move into the Adaptive Navigation control layer. This allows for a transition into source-seeking or similar ”primitive” functions.

Adaptive Navigation involves sensing the terrain that the drones are within and using those sensed values to make intelligent and informed decisions toward a source, in the case of this project’s implementation. Such a test bed requires that the drones in the cluster have sensors related to the values that the drones aim to sense, attached to the drones. In the case of our original real-world field test, the objective was to be able to sense RF signals emitted from an RF beacon as described in the concept of operations model in Figure 2.1. However, these sensors can adapt depending on what type of values exist in the application for which such a framework is used.

With a valid set of sensors attached to the drones, the Adaptive Navigation phase involves obtaining sensor values and studying their changing behavior in relation to those values obtained by other drones in the cluster. This allows for a gradient based on the scalar field values, in the case of this project, to be calculated. This gradient is used to transfer to a control command that can be calculated using the same cluster kinematics equations as explained in the above

equations. This control command is sent to each of the drones, allowing for an automated control command generation and distribution service. Overall, this allows for the drones to move in the direction designated by the gradient which will move the cluster towards the source progressively until the source is found [4]. These equations are included below.

The first equation shows how each matrix for a drone is formatted, where x and y represent the drones x and y position within the global frame, while w represents the value retrieved by the drones scalar sensor.

$$R = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \tag{3.26}$$

The next equations calculate the vectors from the leader drone to the follower drones.
Vector From Drone 1 to Drone 2

$$R12 = R2 - R1 \tag{3.27}$$

Vector From Drone 1 to Drone 3

$$R13 = R3 - R1 \tag{3.28}$$

These vectors are then used to calculate the gradient using the cross product of negative R12 and R13.

$$N = -R12 \times R13 \tag{3.29}$$

Because the gradient is only for two dimensions, the value for z is replaced with zero to get the final calculation.

$$G = \begin{bmatrix} N(1) \\ N(2) \\ 0 \end{bmatrix} \tag{3.30}$$

Such a framework is applicable to many scenarios as mentioned before and our project is a stepping stone for future developments in this area. Our test bed can be used to implement Adaptive Navigation in two-dimensional space and can be further developed for three-dimensional space with limited modifications. With the future work later detailed in the conclusion, we hope to set a course for future developments and research that can utilize our test bed to produce ground-breaking results ultimately bettering our ecosystem and all who reside in it.

Chapter 4

System Integration, Testing, and Results

4.1 System Integration

4.1.1 Octo-copter Integration and Evaluation

Once each of the drones had been fully assembled their were many steps taken to ensure that each of the drones were functioning. To configure each of the drones to operate as an octo-quad-copter the Pixhawk firmware needed to be updated to the latest version of PX4, after which each of the drones parameters would be changed to our specific drone configuration. To do this QGroundControl was used. Through a USB connection from the Pixhawk to our workstation, we were able to flash the latest version of PX4. The version used throughout the project was version 1.9 [5]. QGroundControl also gives users the ability to list all of a drones parameters and change them through a simple GUI. Also when a new drone is connected through USB it takes you through a list of configuration steps before a drone can become operational.

The first and most important parameter is setting the drones air frame. In our case we used the pre-configured octo-quad-copter orientation. By doing this each of the ESCs motors are set to a specific pin on the flight management unit's(FMU) pulse width modulation(PWM) output port. These PWM ports are connected to their respective ESCs motors outlined by the octo-quad-air-frame shown in Figure 4.1. Without the FMU PWM pins being set to the appropriate orientation and connected to the correct motors a drones behavior would be unmanageable. Another advantage to using the pre-configured air-frame is the auto pilot can handle any control commands whilst also maintaining stable flight. Additionally while using this setup, QGroundControl can be used to easily set the correct trims on all the control axes. The next step to ensure stable flight is properly calibrating all of the on-board instruments.

The Pixhawk 4 and GPS module comes equipped with multiple internal instruments. To provide the most stable and accurate orientation the drones have redundant instruments. The Pixhawk 4's internal sensors include two accelerometers and gyroscopes, one magnetometer, and one barometer. The GPS module also includes its own magnetometer as well as a GPS accuracy of +/- 3 meters. Each of these sensor readings are automatically handled by the on-board

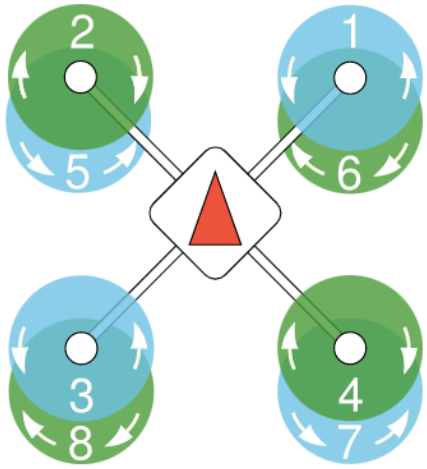


Figure 4.1: Octo-quad air-frame motor configuration [5]

autopilot to maintain position, speed, and orientation. Although this is handled automatically, each of these sensors need to be calibrated before each flight. If there is a mismatch in any of these instruments the drone will fail to arm. To properly calibrate, QGroundControl will instruct the user through multiple specified motions of the drones. Once all sensor have been calibrated within a couple degrees of error, the drones will successfully arm if all other pre-arm checks are met.

Another process that needs to be completed before flight is ensuring the ESCs motors have been calibrated with the proper battery setup. The standard parameters for the octo-quad-copter air frame assumes a 4S LiPo battery, however this can be changed to a NiMH battery but requires specifying the new max and min voltage per cell. Each battery will have different charge characteristics so it is important to calibrate the ESCs motors whenever using a different battery. To do so a voltage reading will need to be taken with the battery connected to the drone. In addition to a voltage reading, a current reading can be inputted to ensure the most accurate battery life estimates. Once these parameters have been specified QGroundControl allows a user to automatically calibrate each ESC motor. Doing so will make sure each motor is receiving the desired current to maintain controllable flight. If the motors and batteries are not properly calibrated the autopilot may force the drone to fail to arm or might not be able to provide low battery readings, causing the drone to potentially shut-down mid-flight.

PX4 offers hundreds of parameter that can be changed within all systems of the Pixhawk. This enables the user to make almost any change he wants to the autopilot's performance. Many of these parameters were useful when testing indoors without a GPS lock and throughout our stationary testing when drones would need to be armed but couldn't take off.

4.1.2 Communication Integration and Evaluation

Once our drones were flight ready and our communication subsystem was developed we needed to integrate them together. Many of these processes were also handled by configuring Pixhawk parameters in QGroundControl. The first parameter changed was to give each of our drones a different system ID so each of the drone could be addressed individually by our ground station. Once the SiK telemetry radios and the Arduino Megas were connected to the Pixhawk, their individual ports needed to be configured to enable communication to our ground station. Because the Arduino Mega was connected to the Pixhawk it was recognized as a component of the drone. This meant that the Arduino shared the same system ID as the drone. The Arduino was connected to a spare telemetry port that was enabled to use the Mavlink protocol and was also enabled to allow for packet forwarding on the port. Through doing so we gave each Arduino Mega the component ID of 20 for each drone. Once these parameters were changed and the appropriate baud rate was set the Arduino was able to send and receive packets.

For a one drone system the primary telemetry port came pre-configured to be used with the SiK telemetry radios, however, even though this worked for each drone we wanted to set up separate frequency channels for each of our radios. We discovered this was possible with our telemetry radios, but QGroundControl wasn't capable of installing the custom firmware to do so. Our solution for this was to use another popular flight control software, Mission Planner. Using Mission Planner we flashed the new firmware onto each pair of Sik radio which enabled separate frequency channels for all the drones to communicate with the ground station. Once the new firmware was installed, the primary pre-configured telemetry port still worked for each of the drones.

For most drone systems there would be two radios, one for commands and the other for telemetry. The common method is using a stick and throttle RC controller that would be responsible for sending commands. However, for our system, we choose to have telemetry and commands being sent from one central controller that existed in our ground station software. To enabled this functionality we had to disable the standard RC channels for commands, as well as disabling the RC checks required to arm any drone

Once all the drones were properly calibrated and each of the drones parameters configured, all of the drones could be armed through our ground station.

4.2 Test Plan

A summary of all the tests run in relation to the project's requirements are listed below, after which each of the tests are elaborated on.

Meeting Requirements	
Requirements	Task Carried Out
Communication Range: Up to 100m	End-to-End Verification Testing
Human operator or automated control of 1-4 drones	Simulation Testing of control of drones to form cluster
Control positioning within +/- 3m	Simulation Testing to move cluster
Simulate Drone Cluster	Simulation Testing
Maximum-seeking Adaptive Navigation in three dimensions	Simulation Testing of Adaptive Navigation
Implement critical safety features	Flight Testing through simulation and physical system testing

Table 4.1: Requirements and Accomplishments

4.2.1 Incremental Testing

At the beginning of the project, the original underlying idea was to have an incremental test plan. In particular, this would involve starting with a single drone. For this single drone, it would be important to test every possible aspect of the drone. For example, each of the motors were tested, other device-related aspects of the drone were tested, and communication with that drone through the other components in the overall system allowed for a detailed and thorough understanding of the successes and limitations of the system. In addition to such device-related tests, the drone would be connected through the MATLAB simulation along with a joystick connected to the work station. Movement of the joystick was registered in MATLAB and the commands sent to the drone would be tested to ensure that the motors on the drone would move accordingly.

From here we could move on to testing two drones together to see if the same communication links and joystick control work as desired. On top of this, another test was to set a global reference frame through the Simulink Stateflow chart and roll the drones on carts to see how their local position varied in correlation to the global reference frame. The main reason for such a test is for verification that local position values sent from the drones relatively match the physical state of the drones with respect to the given global reference frame. In addition to stationary tests like what was just mentioned, testing with two drones for ensuring takeoff, piloting with a joystick, and hold commands can be sent correctly to the drones in an efficient manner, while the drones respond accurately to given commands. Another aspect of hold commands is testing the autopilot software on board the drones by tampering with the drone in hold to see if corrective measures are taken by the drone to reset itself to the desired position set by the pilot/operator.

Once a solid foundation for individual control was established, our plan was to test the cluster controller to see if planar movements of the two drone cluster would result in a successful leader/follower relationship. Such a test would allow for stress-testing of the cluster controller on a smaller scale prior to expanding to a higher-dimensional cluster of drones. An important test in this regime that we planned to also carry out was being able to test the safety of the system by ensuring low latency in traversal between control modes. One way would be to test how quickly movement from cluster control to individual control happens when the cluster is in motion toward a possible obstacle.

Tests Performed		
Test Name	Description	Result
Beacon Range Test	Tests signal strength between two Xbee modules for beacon testing	Produced Nearly Linear Result for 0-70m
End-to-End Physical Test	Ensures all devices on the network can properly receive and transmit	Successful for 4 Drone System
Stationary Test	All drones (without propellers) on our network can be properly controlled	Successful for 2 Drone System
Simulated Manual Control	Tested individual control of SITL from multiple joysticks	Successful for 4 Drone System
Simulated 2 Drone Cluster Control	Tested control of a 2 drone cluster from one joystick	Successful
Simulated 3 Drone Cluster Control	Tested control of a 3 drone cluster from one joystick	Successful
Simulated Two Dimensional Adaptive Navigation	Tested that a 3 drone cluster could find a maxima within a plane	Successful

Table 4.2: List of Tests Performed Throughout the Project

From here the test plan progresses similarly for higher-dimensional clusters going to three- and four-drone systems while testing in a similar manner to a two-drone system. One main addition to the test plan once the formation includes three or more drones is the testing of the adaptive navigation suite. This involves carefully testing various scalar field distributions to see if two-dimensional adaptive navigation works to a satisfactory standard and later if three-dimensional adaptive navigation also works well.

Incremental testing not only includes progressively scaling our system's test bed to include more complicated systems, but also involves testing each component within our system to ensure no one component can possibly be a possible single point-of-failure. This is described in greater detail in the following end-to-end verification testing section.
A complete list of our performed tests can be found in Table 4.2

4.2.2 End-to-End Verification

Verifying that all packets are successfully transmitted across all nodes within our network is important to safe operation of the cluster. Continuous testing was performed to ensure the packets were traveling between the correct nodes as well as being received properly at the destination.

As each connection was established, the link was thoroughly tested to record and reduce any major packet losses. The first tested connection was the radio link between the Pixhawk modules and the ground station.

This test was performed prior to assembling the drones with just the Pixhawk and radio modules. To test this connection, the radios were connected to each Pixhawk device and the ground station ports. The ground station connection was then tested by running MAVLink-Router and QGroundControl to view the connection status of each Pixhawk.

Once it was confirmed that two, three, and four Pixhawk devices could be simultaneously connected and controlled

within QGroundControl, the link between the Pixhawk and scalar field sensor was tested. To test this connection, all MAVLink packets received by the Arduino were printed out to the Serial Monitor to confirm that both devices had completed the handshake process.

A final test in the end-to-end verification was to ensure that the XBee-retrieved sensor values are able to be sent to MATLAB. Mavlink commands implemented in MATLAB were used to listen for the NAMED_VALUE_INT corresponding to the RSSI value that is sent from the Arduino to MATLAB. Then MATLAB was able to display the received messages to confirm that MATLAB received the correct values to be used for later calculations internal to the Stateflow diagram.

4.2.3 Stationary Testing

Stationary testing was conducted to ensure that the motors equipped on the drones could be controlled through movements made through the joysticks connected through MATLAB. The process is as follows:

1. Plug in radios and joysticks. The number of radios should match the number of endpoints in the config file. The number of joysticks should be four.

2. Run MAVLink-router by typing mavlink-routerd into the terminal. Figure 4.2 shows the terminal output once mavlink-routerd has been run correctly. If any radio is not connected, the line "Could not open /dev/ttyUSB (No such file or directory)" will be printed out.

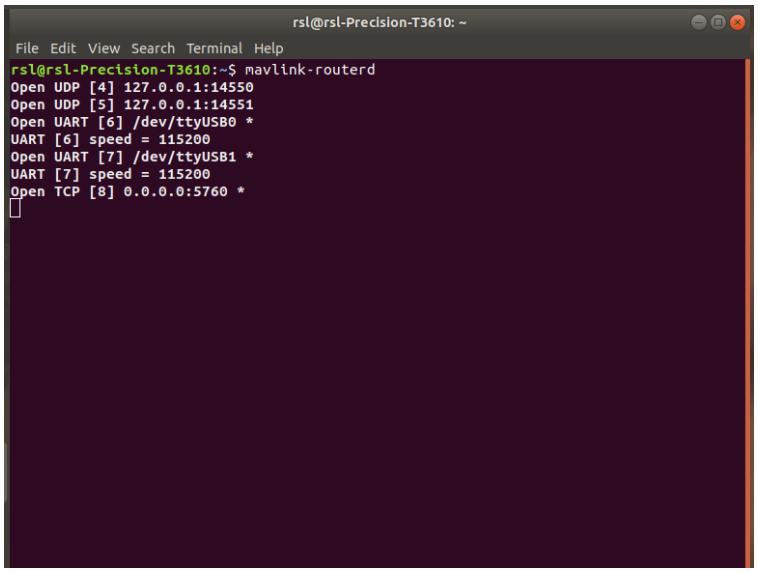


Figure 4.2: Correct Output for MAVLink-Router

3. Start QGroundControl and wait for drone connection.

4. Start MATLAB, but not the state diagram.

5. Power up Drones with battery. Upon power up, the Pixhawk will automatically begin communicating through the available radio pair. If no radio pair is available, the Pixhawk will wait for a connection to be made. Each Pixhawk will signal with a tone on power up and GPS lock.

6. Wait for connection to QGroundControl to be established. Each drone will connect and exchange its initial parameter list. For full parameter listings, plug each drone in and wait for full connection one at a time. For a faster connection without the full parameter access connect all drones simultaneously. Figure 4.3 shows the central multi-drone display of QGroundControl. The GPS position of each drone is displayed on the map. Each drone is listed in the right column, where its mode can be set for the mission.

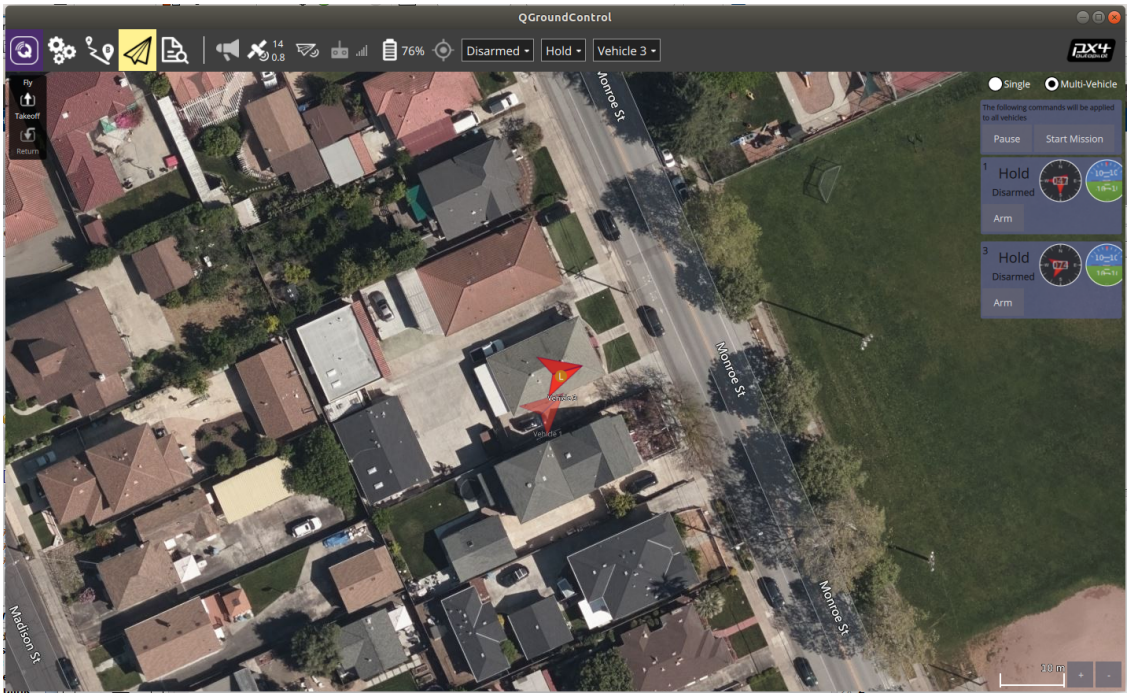


Figure 4.3: QGroundControl Multi-Drone Display

7. Arm drones through QGroundControl. The right column in Figure 4.3 is where each drone is armed prior to switching to the control program. Once armed, the motors will begin spinning at low power and the drone is ready for control commands.

8. Run the Stateflow diagram in MATLAB to connect to and control the drones. When running, the initial prompt asks for the number of drones being controlled. Once this is filled out, the Stateflow diagram is running. This is shown in Figure 4.4 .

Figure 4.5 shows a picture of all the fully integrated and configured drones being tested during our stationary test. The drones were placed outside to ensure a stable GPS lock and the data flow and control abilities were tested.

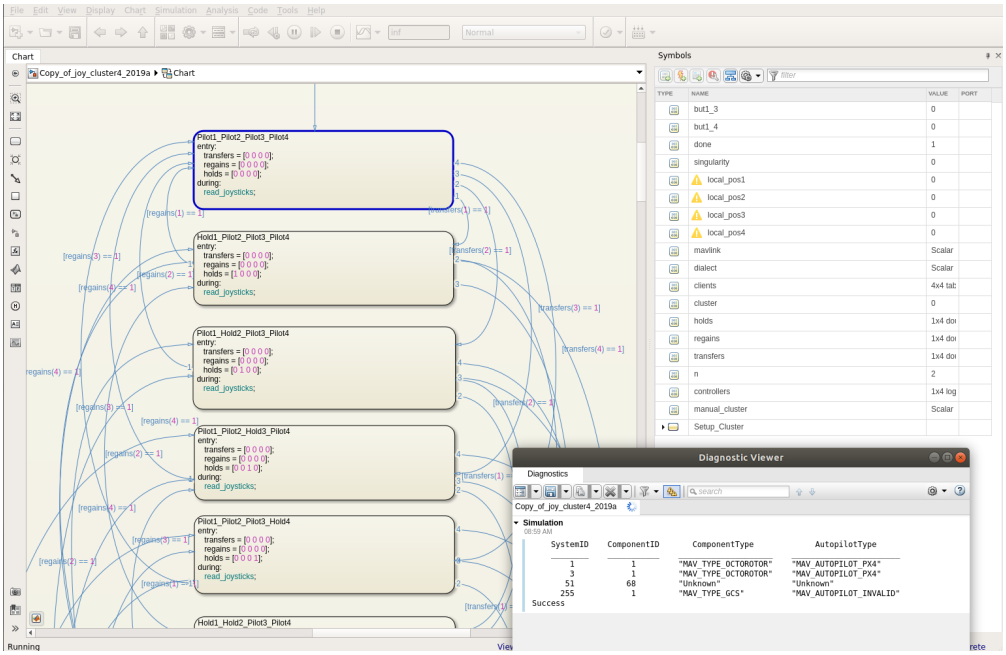


Figure 4.4: QGroundControl Multi-Drone Display

Figure 4.5: Stationary tests of four drones

4.2.4 Control Testing

The control subsystem testing was done in two main parts, the second of which was a product of transitioning to a simulation framework from a physical drone test framework due to COVID-19. The first part was a physical test to ensure that the joysticks connected through MATLAB/Simulink and the implemented state machine was able to

control the motors equipped on the drones. The second part was a test of control of each individual drone, of a cluster of drones, and of Adaptive Navigation with that cluster of drones to find a source of scalar field all through a simulated drone framework.

The first physical test involved arming the physical drones through QGroundControl. The joysticks were connected to the workstation. This was followed by running the Simulink Stateflow chart simulation to test each joystick's movements to test if their movements were registered as commands that were sent to the drones resulting in varied motion of the motors corresponding to the movement made through the joysticks. The motors on the drones had tape attached to verify visually that the rotation of the motors were modified by movements made through the joysticks and that such modified rotation seemed valid in correspondence with each joystick's movements.

The simulated field test involved first starting the simulated environment, the Gazebo simulator, and consecutively starting the Stateflow chart simulation to progress through the state diagram while controlling the drones in the simulation with joysticks. With separate joysticks for each drone, the drones were progressively moved to their corresponding position in the cluster formation before activating the cluster controller. Once the cluster controller is activated, the first joystick is used to control the cluster of drones as a unit. This allows the operator to move the cluster anywhere in the environment set up in the simulation. Once the pilot/operator has moved the cluster to some initial position away from the source of scalar field values evident throughout the environment's plane, the Adaptive Navigation control layer is activated by the pilot. The Adaptive Navigation control layer involves computations of local scalar field values corresponding to gradients computed at the ground station. These computed gradients are then analyzed to be converted into corresponding velocity commands to be sent to the drones using the cluster controller's calculations to move all drones in the cluster as a unit toward the source. If the source is found, the drones in the cluster begin to fly in circles notifying the successful finding of the source at which point the drones are brought to a stop by the pilot and can be safely landed. All of these protocols are explained in more detail in the results section below.

4.3 Procedures and Checklists

In accordance with SCU UAV flight policy, SCU faculty, students and/or staff may only fly UAV vehicles under the following conditions:

1. These procedures are to be executed only by students who have been properly trained and approved to fly the X8.

2. The Remote Pilot in Command holds a valid FAA UAS Remote Pilot Certificate/License.

3. The UAV is registered with the FAA under CFR Part 107.

4. The SCU UAV flight review board must approve flight privileges based on the platform, the Remote Pilot in Command and purpose. The Remote Pilot in command must be an Academy of Model Aeronautics (AMA) member in good standing and with AMA liability coverage in order to conduct any flights/testing other than bench tests; any flight objectives that do not meet AMA constraints/guidelines must be explicitly reviewed and approved by the SCU UAV flight review board.

Flights must be conducted in an area with adequate space given the objectives of the flight.

- Indoor flight on University-owned property requires approval from the University prior to flight. Flights must be conducted in the pre-approved flight locations or with special approval from the University and the RSL Director.
- Tethered flights may be conducted under the conditions that the work space is clear of hazards, personnel, etc.
- Constrained bench testing is permitted given the use of approved safety barriers, emergency off switches, etc.

4.3.1 Flight Test Checklist:

The flight test checklist was broken into a checklist corresponding to safety and the safety of equipment, briefing of all members prior to flight testing in the field, a controller pre-flight check, and a vehicle pre-flight and startup procedure checklist. All of the checklists are colluded and presented in Appendix E. These procedures and checklists were followed strictly in the testing and implementation done that followed the updates to our objectives due to COVID-19. These checklists were used primarily to ensure the safety of all equipment, members, and bystanders involved in testing and observing during the testing operations.

4.4 Results

4.4.1 RF Beacon

Although a fully realized real world Adaptive Navigation system test was never preformed, preliminary results of our scalar field sensor and beacon were tested to show its feasibility for future Adaptive Navigation tests. In our scalar field test we were able to gather results that showed what our RF scalar field would look like. Because our system used a moving average of the last three RSSI values this was also done in our scalar field test. As shown in Figure 4.6 we can see that from 5 meters up to 70 meters the graph is nearly linear. After this section levels off we can then see the curve flatten. This behavior was unexpected and the test wasn't performed with extreme accuracy but for simply proving our system could preform AN was still possible. Our original requirement for our system was to perform AN within a 100 meters of the beacon would still be possible if the correct cluster sizing was implemented. These results helped inform us on how large our formation of drones should be. Its difficult to say what the best formation

size would be without running a few tests, but with sides of a formation roughly between 10-25 meters we believe our system would have been able to navigate its way to the RF source.

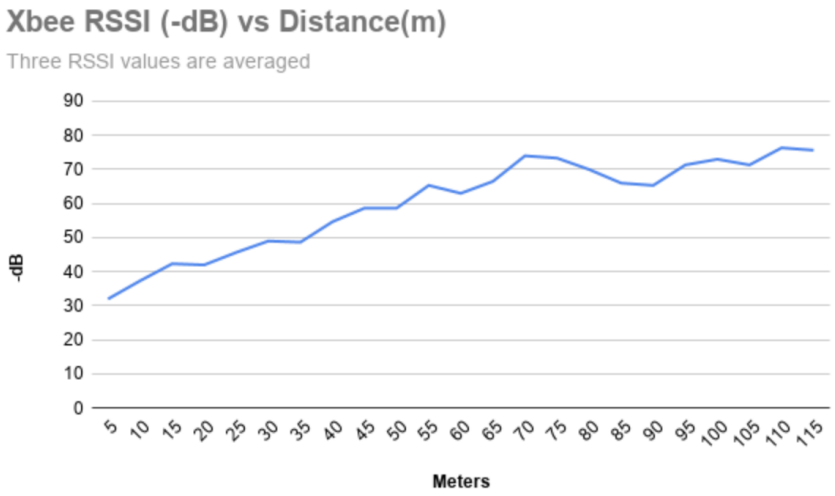


Figure 4.6: Signal Strength(RSSI) vs Distance in an open field

4.4.2 Physical System

In carrying out the stationary test to probe the physical system, it was discovered that connecting two drones through MATLAB was viable and worked efficiently. By following the steps outlined in our stationary testing procedure, the ground station was able to send manual throttle control to both drones as well as set the hold mode of each drone. Packet loss of incoming telemetry data was minimal and all drone status was continuously updated in QGroundControl. An operator error that was quickly caught was the order in which the drones were armed and the control program was run. If MATLAB began sending commands to the drones before QGroundControl sent the ARM command, the connection to all drones become too unstable for continued operation and required power cycling. Due to the stationary aspect of this testing, control latency could only be observed and recorded through throttle response and not positional movement. For a 2-drone system, motor speed response to throttle was observed to be around two seconds and to rarely exceed three seconds.

However, as more drones are connected along with joysticks, the latency goes up and more communication packets are lost as a result of higher communication bandwidth. Reasons as to why the increase in communication packet loss and dropping was attributed to the MAVLink-Router. By connecting four radio channels to a single UDP port, the router acts as the central bottleneck of our network. Initial testing of the router did not involve the full drone subsystem, resulting in lightened packet load of available telemetry data. Once the full system had been connected and data began

to transfer, packets involved with querying the autopilot state dropped at a higher rate than QGroundControl allowed for. The baud rate for all connection within the router was increased, but a 3-drone system could not be consistently armed before losing connection to one drone.

4.4.3 Simulated System

Due to the COVID-19 pandemic, much of the physical testing that would have been occurring in Spring quarter of 2020 was moved to a simulation environment. While much of the core structure remained the same, there were a few alterations to allow for this to happen. The first change was all physical hardware of the drones was not a viable option for testing due to the distributed nature of the team. In order to perform our tests, we relied heavily on the PX4 development team's GitHub repository, which included the necessary firmware to create Software In the Loop (SITL) instances on the computer [5]. Additionally, it also included plugins and scripts for the SITL instances to interface with simulation environments, allowing for a continuation of formation and control testing without relying on external hardware.

For this simulation, Gazebo was chosen as it provided the most realistic environment for a simulation while remaining relatively lightweight. The operating system for the environment was Ubuntu 18.04, which was the same operating system used for our control software before and meant there were less dependencies to check. The only major difference that had to be accounted for was the communication channels for the drones were slightly different as the SITL instances did not communicate through telemetry radio or serial ports. Instead, they communicated through UDP ports, but the only change that was necessary for the system was to switch for serial port outputs for each drone to their specialized UDP port for each. Once the software was properly installed, we were then able to continue testing of our control and communication architecture

Communication with Software In the Loop Instances

For the simulated system, the approach for developing a test plan was to transition much of the original test plan of flight into a simulated environment. The test procedure begins by initializing a single instance of the PX4 autopilot on a workstation. This instance of the autopilot was not connected to any simulation environment and instead ran headless with only a command line interface. QGroundControl was then launched on the same machine to determine that the SITL could be recognized as a PX4 instance. Telemetry was being constantly monitored to ensure that the PX4 SITL was utilizing the same telemetry packets and commands as the PX4 hardware instance on the Pixhawk.

Once this was properly verified, a new instance of the autopilot was launched with the a connected Gazebo object and simulation environment. Again, the system was verified to be recognized as a PX4 instance in QGroundControl. Once the proper telemetry was confirmed to be received, QGroundControl was used to arm the drones. This action was confirmed within QGroundControl through audio and written confirmation, the PX4 SITL command line through

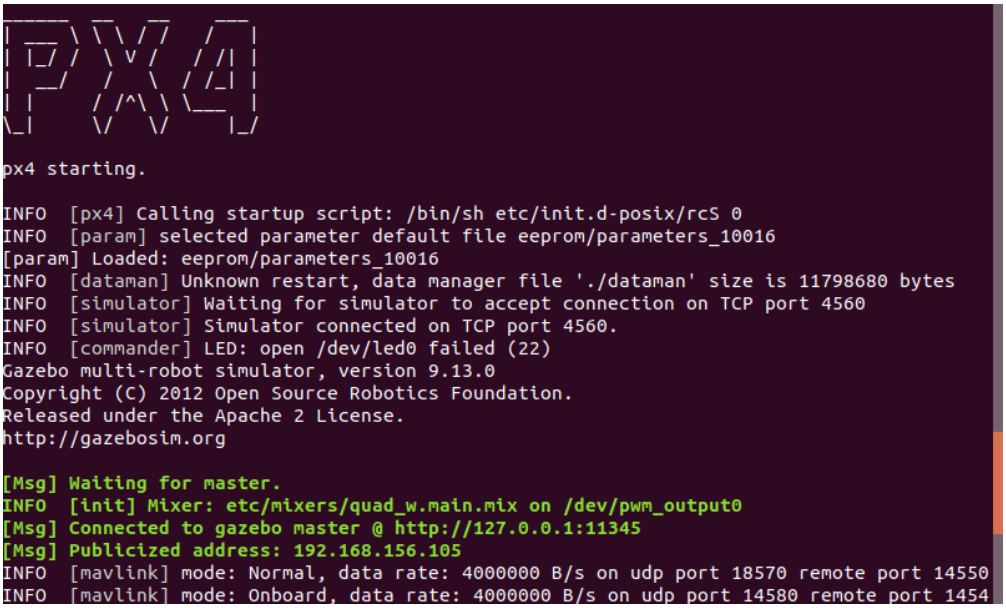


Figure 4.7: PX4 Command Line Start

written confirmation, and the Gazebo simulation through visual confirmation.

At this point, the simulation could then be tested to respond to physical commands sent from the ground control software. In order to facilitate this test, QGroundControl was utilized with an attacked joystick to allow for preliminary joystick control and verification. After the joystick was calibrated, and the PX4 SITL was armed, QGroundControl was used to send a takeoff command to the PX4. At this point, the simulation visually verified that the SITL responded to the Takeoff command. From there, the PX4 held position until motion was initiated with the joystick, again confirming that the PX4 SITL was acting as expected compared to a physical hardware instance of PX4.

Afterwards, more instances of PX4 were launched using the provided Gazebo script for PX4 instances. It could be specified exactly how many instances were required for the simulation, so the test included four instances of the SITL to represent four physical drones. The script spawned four of these drone objects within the gazebo world, with PX4 autopilots attached to every single instance. From there, it was confirmed in QGroundControl that each of the drones could be armed, taken off, and then individually piloted through QGroundControl as in the individual drone test.

After it was confirmed that the system was operational with multiple instances of drones, our own software was then tested. The main goal was to ensure that each drone functions similarly to all others through our software. As predicted, it was possible for our MATLAB program to find all of the PX4 instances and it then displayed these devices on our console. This ensured that all proper communication channels were effectively established and the drones would behave appropriately if commands were sent.

Manual Control Testing

To perform control testing of the drones within our system, testing began with a one drone simulation. From there, it was tested whether individual control of the drone was possible. This was verified through a program that sent heartbeat messages and MANUAL CONTROL messages, which is the basic functions needed for a joystick control when QGroundControl is operating in parallel. This program sent MAVLink messages over UDP to the communication port of the drones while QGroundControl also operated in parallel. To test full flight functionality, the drone was armed and taken off through QGroundControl. After the drone had successfully taken off, the program was run to determine that there was successful control of a single drone using our MATLAB program.

After this was verified, a synchronized control formation was developed, which was where every drone would receive the same message. This was done by using the same program as before, except modifying the program to include a for loop to increment the target system field of the message packet so there would be a packet destined for every single drone within the system. To perform this test, the multiple drone gazebo script was loaded with four copies of a quad copter. Each drone was armed and taken off with the assistance of QGroundControl. After every drone was in the air, the program was initiated which allowed the drones to be flown as one group all receiving the same command. This ensured that every drone was capable of being controlled in parallel, which ensured that the drones were capable as a test bed for checking the capability of cluster control functions of all of the individual drones.

Two Drone Cluster Testing

The main purpose of this test was to make sure that all of the cluster functions for a leader-follower drone system worked as anticipated. Additionally, it verified that each portion of the Stateflow diagram within Simulink was working in the correct order and allowed for full control of each drone and transition through all of the states. To perform this test, the drones were armed and taken off using QGroundControl. After all the drones were in the air, the 2-drone Stateflow diagram was run. As the program was running, each of the states were transitioned through to ensure that each controller was able to communicate with the drone it was responsible for. Additionally, if the state called for the controller to have no manual control within that state, it was also ensured that all joystick direction commands were ignored appropriately After each state was was working properly, the operator transitioned to the cluster control state.

Within the cluster control state, the operator transitioned to the command window within MATLAB. From there, the desired values for dx, dz, ϕ, and α were all entered, as well as all of the corresponding values were entered. Once the appropriate values were entered, the operator attempted different maneuvers utilizing the handheld controller originally purposed for the leader drone of the system. Maneuvers in the x, y, and z directions were all tried to see if the follower drone correctly positioned itself within the cluster after each command was sent. After all of the maneuvers had been finalized, the cluster state was exited by the operator and the operator transitioned to the initial state of the

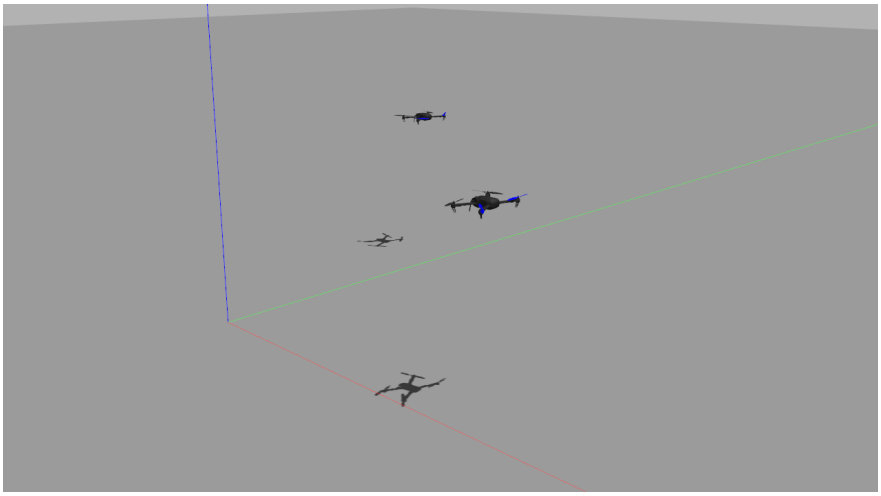


Figure 4.8: Two Drones being held in Cluster Formation

Stateflow diagram. The Stateflow simulation environment was stopped and all of the drones were landed using the QGroundControl ground station.

Three Drone Cluster Testing

Building upon the two drone simulation test, the three drone initial test was to ensure that all of the three drone cluster functions performed the same was as all of the two drone cluster function. Additionally, it facilitated as a way to test a more complex Stateflow diagram than the two drone Stateflow diagram. The test began in very much the same way as the two drone test, except the Gazebo simulation had three drone instances rather than two. After the initial arm and takeoff of each drone, the drones hover in position until the three drone Stateflow diagram has been initialized to run. At this point, all of the states are entered and exited by the operator to conclude that there are no errors in any of the entry or exit logic of the states. Then, once all of the logic has been affirmed, the operator transitions into the cluster control state. Once there, the operator enters the desired values for dx, dz, phi, and alpha for each of the two follower drones to establish the cluster. Once the cluster has been established, the first joystick is then used again to pilot the entire cluster. Each direction is checked to make sure the cluster behaves appropriately. Once this has been confirmed, the operator then descends the Stateflow diagram to the base state. The Stateflow diagram is then terminated and all of the drones are then landed through the QGroundControl interface.

Adaptive Navigation in a Two-Dimensional Plane

The final test that was performed in simulation was a test of the two dimension Adaptive Navigation engine. Because there was no physical hardware and no sensor technology on the simulated drones themselves, the goal of the adaptive navigation engine was to find the center of the map, which is where the first drone's initial position is within the simulation. The test is performed exactly the same as in the three drone cluster space test, however, after the cluster

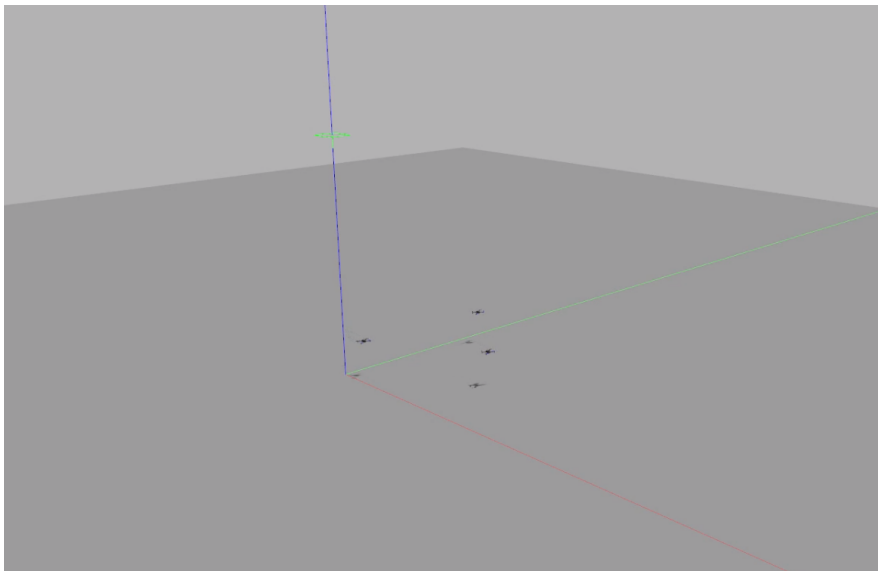


Figure 4.9: Three Drones being held in Cluster Formation

of three drones has been established, the operator flies to the corner of the map within the Gazebo environment. From there, the pilot then transitions to the Adaptive Navigation engine to test whether the gradient to the desired destination is properly being calculated. The drones then drive along that gradient until they reach the desired location, at which point the cluster begins to rotate around the point of interest. Once this happens, it means the test was performed successfully and the operator can disable the adaptive navigation engine to put all drones into a hold formation. The cluster is then degraded in the same manner as the three drone cluster test, where the simulation is stopped and all the drones are landed through the assistance of QGroundControl. Figure 4.10 shows a successful run of this test with the cluster hovering around the blue line after running adaptive navigation to determine its location.

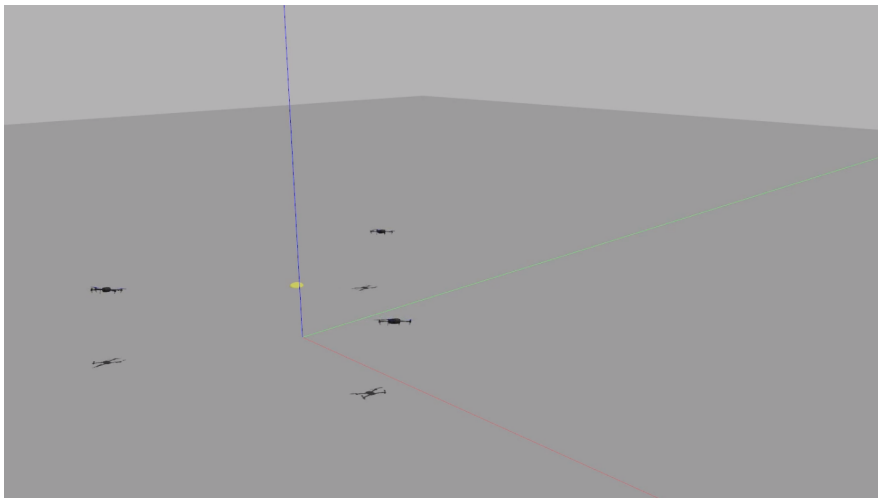


Figure 4.10: A Three Drone Cluster Finding The Point of Interest using Adaptive Navigation

Chapter 5

Engineering Standards and Realistic Constraints

5.1 Standards and Regulations

Standards that involve proper documentation and a distributed code base to support distributed teamwork, especially during the COVID-19 pandemic, were utilized over the course of this project. For documentation, a service called Trac was used to ensure that all aspects of our project are described in detail through this platform that allows other security-permitting individuals to view the documentation and is helpful for future senior design or research groups who might endeavor in a topic related to that of our project. A distributed code base was established through the use of SVN which allows for code sharing and version control. This allowed individual team members to make updates to the code base and push their updates to a shared SVN repository that holds all our code and is accessible to future groups that might find such code useful.

As per legal regulations, it was important to obtain an FAA small UAS drone pilot license to safely and legally fly drones for a real-world field test. Also with the usage of LiPo batteries equipped on the drones, it was important that each of us participated in LiPo battery training to learn how to safely handle LiPo batteries for usage and charging. In addition, FAA regulations for multi-drones were to be satisfied by filling out a multi-drone waiver. However, due to the shift from a real-world field test to a simulated field test stemming from the COVID-19 pandemic and its repercussions, there was no need to fill and submit the form. In addition, regulations for flight testing were satisfied by ensuring that field tests were to be run in RC Parks, a local University-owned warehouse, and similar locations where legal flight tests are allowed to be run.

MAVLink 2 was implemented as the standard communication protocol between all devices [14]. MAVLink 2 offers lightweight, 14 byte packets with a common dialect containing messages specific to drone control. This protocol is also open-source, making it easy to build libraries for various devices and alter the dialect if custom messages need to be added. Multiple dialects are available to allow for the contents of the MAVLink packets to be tailored towards

specific devices and drones.

Other regulations followed throughout this project were as listed in the checklist above in the System Integration chapter. All of these regulations were followed to ensure utmost safety of our system for our system for any individuals involved in the testing whether directly or indirectly.

5.2 Ethical Considerations

5.2.1 Intentions/Justification

For our project, we intend on taking a utilitarian approach to our standard of ethics. By doing this we are maximizing the good of our project by weighing it against the potential harms. We define goodness within engineering as something that helps better society and the environment whilst advancing technology in a manner that causes the least amount of harm. For our project, we are trying to bring forward a technology that has the ability to greatly benefit society and the environment. This technology is Adaptive Navigation. The goodness of our project is affected by how well our project functions as a test bed for the Robotics Systems Labs' (RSL) Adaptive Navigation software.

5.2.2 Benefits

Being successful in building a drone cluster that runs Adaptive Navigation software means the RSL can further test and develop their software in a 3-dimensional space. The benefits of this project are fully realized when the software reaches a point for commercial and practical use. Adaptive Navigation is used for intelligently mapping and quickly finding points of interest. These points of interest could be search and rescue beacons, pollution sources, thermal sources, etc. Being able to find and map these points allows for cheaper and faster navigation. The improved efficiency realized through using Adaptive Navigation also allows for better results in being able to bring about satisfactory results in responding to urgent scenarios as described above. Using Adaptive Navigation can also allow for a more effective use of technology that minimizes the risks and harm incurred by humans trying to help in search and rescue, pollution-seeking, or radiation-seeking scenarios. With drones taking charge of identifying sources or other meaningful points can minimize the overall exposure of toxic or harmful objects like radiation, pollution, or fire to humans.

5.2.3 Ethical Consideration and our Approach

By taking a utilitarian approach it is important to us that we lay out our ethical considerations and how we deal with them. Our most important consideration is safety for ourselves, others, and our environment. Building this project includes inherent safety risks but we will not make any decision that would intentionally jeopardize safety in order to meet any other goal. In this manner, serving the community and protecting the ecosystem exists as an end to our means and not a means to our end and, in this manner, exists as our utmost priority in carrying out the goals of this project. We are also taking many safety precautions to mitigate safety risks. Other considerations include user-simplicity,

functionality, cost, technological competence, and integrity. Because our project is within and for the RSL we need our system to be easily operated and understandable to others in the lab. After completing the core functionality of being able to test Adaptive Navigation software our priority lies with user-simplicity. We also believe the simpler the system, the safer it is to operate. Because a significant portion of the lab's research is in Adaptive Navigation we find functionality to be more important than cost. This helps us not cut corners and get our design to the testing phase faster. Nonetheless, we are still minimizing costs by doing thorough research and building incrementally. Due to the safety risk of our project, it is crucial for us to understand the systems that we are using and be able to spot any faults in our system before testing. We also will make operational procedures and document all issues and solutions. Lastly, the lab's Cluster Control and Adaptive Navigation software project spans many years and is the lab's property; therefore, we will not share any information without the lab's consent. Integrity within our design, the lab, and our approach enables us to complete an ethical project.

5.3 Sustainability

In regards to social sustainability, this project has applications in search and rescue operations that can embolden support for firefighters and other rescue teams in times of distress by locating sources of danger or finding beacons to help people in distress. The Adaptive Navigation focus of this project seeks to integrate efficiency and accuracy in the ability of the cluster to locate a source or target in question. This motivation arises from the goal to ensure safety for citizens affected by natural disasters but also for firefighters or other rescue teams who risk their lives to protect others who are victims. Using drones and Adaptive Navigation, search and rescue operations can be completed efficiently, effectively, and safely without the need to endanger any individual's life.

On the side of environmental sustainability, this project seeks to apply to the prevention of forest fires and pollution. With the methodology of our project enabling drones to locate sources or targets and to map regions of particular significance, applications in forest fire prevention and pollution localization are effects of this project. Adaptive Navigation enables the efficient and pervasive necessities to ensure natural disasters are less likely to occur and for wildlife to be protected by the negative effects of man-made actions. In this manner, the protection and prospering of wildlife, nature, and humans are focuses of this project and sustainable living practices can be complemented by using the outcomes of this project.

Within our project, we have focused on sustainable design through three main principles. These principles are eliminating the concept of waste, designing for reuse and recycling, and seeking constant improvement by the sharing of knowledge. Through working with the RSL, we have been able to make use of all their resources within the lab. These resources have enabled us to use past projects and spare parts to help prototype and build our project. By doing this, we have minimized the materials wasted and have given new purpose to older systems. The drones that we are using

are being re-purposed from a past project to bring new life to the drones. We are also using the same batteries used to previously fly the drones to avoid wastefulness. As well as reusing the drones and batteries, we have also given our drones upgrades through integrating new hardware to ensure that they can continually be used for future projects and research within the lab.

Through the addition of an Arduino and coding it to be able to talk to our flight computer, we have allowed for further scalability and use of our drones. Lastly, by working with the RSL lab we have been given access to their expansive knowledge of robotics and the design process. This has allowed us to understand early on within our design process of what will work and what won't. We were able to look at all the documentation of past projects and work alongside current projects to identify certain technologies that would be helpful. We have also documented and shared information on our project to continue the tradition of sharing knowledge for the further advancement of sustainable design within the RSL.

5.4 Risk Assessment

As mentioned above our greatest risk was safety for ourselves, others, and our environment. We mitigated this risk in many ways but can not say for certain that our drones won't ever fail. To ensure we were operating as safely as possible, and legally, we completed all the necessary FAA drone pilot certifications and applied for a waiver to allow for single operator control of multiple drones. We have also received training from the RSL for using and handling lithium polymer batteries. Furthermore, we incrementally built and tested the design at each stage to verify our design. We then completed an Environmental Health and Safety form to ensure we were taking all the proper safety precautions before any drones take flight.

Another concern regarding our project is in the event that we were only partly successful with our project. Because our project is within the RSL and there is a demand for our project, there would be interest in the lab completing it with another senior design group or separate from senior design. We don't believe this could cause anyone immediate harm, but we would want to confirm that whoever continues progress on this project does so using the same approach to safety and security.

One of the most frightening risk for our project, although unlikely, was that our project would fall in the wrong hands outside of the lab. We believe this is unlikely as the lab has control of the software and we would only be using the software on SCU/RSL computers. However, if the software and our system were fully developed and used by a wrongdoer, it is possible that our project could be used for harm if it was further expanded to do so. Although this is frightening, we believe it is better to take the risk and build this system as it can provide more benefits for society than harm.

5.5 Safety

For our Senior Design project, we have made it one of our objectives within our project to ensure safety for everyone involved as well as the environment. We have gone through a variety of training and have been constantly working on receiving proper safety approvals before each of our tests. Working with drones, especially multiple drones, comes with multiple safety and environmental hazards. Outlined within this safety report are all of our hazards and how we are working to mitigate the associated risks.

For our project, we will be using Lithium Polymer (LiPo) batteries to power each of our drones. We will be required to store, transport, and charge and discharge each battery. We have chosen to use a four-cell LiPo battery due to its energy density characteristics, popular use within drones, and their availability within the RSL. LiPo batteries, however, are prone to catching fire if they are not handled properly. With this hazard, we are unable to eliminate or substitute, because each drone needs a battery source to fly and the battery has to be power-dense enough to support flight. To minimize this risk we have each taken a LiPo training course. Through this course, we have learned how to safely handle the batteries. We now know, if a fire does occur we will smother the batteries with sand to prevent it from spreading. We will also not fly over a brush to prevent a brush fire and have easy access to water if a fire were to start. There is a low probability of this hazard occurring but it is an important hazard and the necessary training has been completed to mitigate this risk.

Another potential risk is a low battery charge for the drones while the flight mission is in progress. This is mitigated by carefully planning the flight and using the drones autopilot battery monitor as a backup system to return to base when battery levels become too low.

Our drones will be set up in an octo-quad-copter configuration. This means that each drone will have eight propellers and two motors attached to each of the four arms of our drones. The hazard associated with the motors and propeller blades is that they can spin very rapidly and if the control was lost they could cause damage to persons, property, or the environment. Due to our whole project being based around drones we cannot eliminate or substitute this hazard. The next best approach we have chosen is to isolate and use protective barriers and equipment to keep us and others away from our drones. We also are working on procedures and checklists that will be followed prior to and during any test. These checklists will include multiple checks of propellers to ensure safety. To help mitigate the risk of propellers causing harm we will only work on the drones with the propeller blades detached and the battery disconnected. The drones will also have an advanced autopilot to prevent erroneous behavior during flight. While testing, we will all stand behind a wooden barrier to reduce the risk of contact with the propellers.

To test our drone swarm we will have to fly our drones. Flying drones can lead to risks of crashing/falling or flying away of drones. All of these situations could cause harm to persons, property, or the environment. To minimize the

risk of injury or damage we will follow all FAA flight rules. Brendan and Aditya have received FAA small UAS licenses to ensure we are operating legally and safely. We will also fly at a remote offsite location where there is little to no possibility of danger to others. We will have at least one person at all times looking out for people or other environmental hazards. Each person included in the test will wear a high visibility vest and a hard hat. At any point during the flight, each drone will be able to switch back to individual manual control.

Another potential risk associated with drones is the loss of communication packets about any of the communication links between each of the components in our project setup. Safety precautions of landing drones in critical battery health or ensuring drones hold their positions if communication link packets are dropped or corrupted will be taken care of by our autopilot. Although we acknowledge the hazard of using RF for communication, we will not use power levels anywhere close to being dangerous.

5.6 COVID-19 Constraints

Due to the COVID-19 pandemic, many of the the core functionalities that we wanted to test within our physical system had to be put on hold as we transitioned to a system that could properly demo our progress.

5.6.1 Physical System Testing

The biggest hindrance to our project was the transition from a physical hardware systems to being able to demonstrate our project entirely in software. As discussed in our testing section, much of our formation testing and Adaptive Navigation testing had to be completed entirely in simulation rather than testing in a open area location like what was originally proposed. Additionally, any testing that was completed with the physical system had to be grounded due to social distancing policies that prevented our team from meeting in close proximity. Finally, the pivot of our system from a physical one to a software solution took a significant portion of our time for the project, and resulted in less testing opportunities overall for any physical systems that we had purchased throughout the year.

5.6.2 Communication

Due to lack of in person meeting and our team being distributed across the country for the entirety of Spring Quarter, much of our communication and interaction had to occur entirely online. Therefore, much of the fast, in-person integration and collaboration that was enjoyed the previous quarters was relegated to services like Zoom, Trac, SVN, GroupMe, and email. As a result, much of the communication and troubleshooting happened much slower than normal as team members were working alone across multiple different time zones It was then important to our team that we each made a active effort to stay present in the project and aware of what each team member was working on as to keep the collabrative atmosphere that exist prior to the COVID-19 outbreak.

5.6.3 Adaptive Navigation Shortcomings

The last major constraint that was placed on our team due to the COVID-19 outbreak was a changing of goals regarding Adaptive Navigation The original goal for the project was to complete a four drone cluster controller and Adaptive Navigation engine that would have allowed point of interest location in three dimensions. However due to a loss of time of having to transition our project into a simulation environment, there was no feasible way to complete the three dimensional Adaptive Navigation engine. However, we were still able to perform Adaptive Navigation with a drone cluster, which our team believes is still a very large accomplishment and we hope that our work can be utilized to eventually reach the goal of three dimensional Adaptive Navigation.

Chapter 6

Conclusion

6.1 Overall Evaluation of Design

The project's objectives were to first implement an end-to-end experimental simulation test bed for adaptive navigation in three-dimensions for a three-drone cluster. Secondly, it was important for the project to be able to model and implement a multi-layered control hierarchy that was also configurable. This meant that the pilot or operator was able to switch between an individual control mode, the cluster control mode, and the Adaptive Navigation control mode. In addition, the pilot or operator should be able to decide which drones or group of drones were under which control layer in the hierarchy and that the drones would be able to be controlled by either pilot commands or automated generated commands sent from the next layer up. Thirdly, our objectives included implementing critical safety measures for all aspects of the project and flight.

The resulting system produced by this project was able to meet all the revised objectives and requirements set out after changes made due to the COVID-19 pandemic. In particular, through this project a multi-drone Adaptive Navigation experimental test bed was constructed for end-to-end testing in a simulation environment. With further improvements and testing of the implementation this test bed testing with a physical framework can be implemented with real drones in a field test. In addition, this Adaptive Navigation framework is able to identify sources by calculating gradients of the scalar fields and efficiently calculating and commanding a cluster of drones corresponding to gradient descent/ascent.

In addition to an Adaptive Navigation test bed, we were able to retrofit four drones with RF sensors, GPS modules, and telemetry radios such that all four drones are flight-ready. A multi-point communication system was also developed through the course of this project and was implemented to efficiently communicate packets with minimal latency. With more time, issues found in communication packet loss can be addressed and fixed to improve the efficiency and overall safety of our system. With a two-drone system, however, the communication network works effectively while ensuring the overall system works to satisfactory standards. This was confirmed through a hardware test of end-to-end communication with two drones.

By the finale of this project, the successful implementation of three drone cluster control and Adaptive Navigation control was achieved. This was tested through the use of a simulated environment with three drones aiming to seek a global maxima signal. Overall, the system in this context worked as required with ease of control for each drone when forming the cluster, for the cluster when moving it as a unit, and for Adaptive Navigation for the cluster when seeking the source. The general formation was held in position in relation to each drone's relative cluster position and for the cluster geometry using a proportional controller. In addition, the pilot/operator was able to easily transition between each control layer in determining which drones were under which control layers. Critical safety features were also enabled through the controller implementation through the state-based controller in Simulink by ensuring that the pilot/operator has the flexibility to transition between control layers with low latency, especially in situations where a transition from the Adaptive Navigation control layer to the cluster control layer is necessary to avoid external obstacles, for example.

In light of the COVID-19 pandemic, this project's implementation was still able to meet all general and main requirements in a timely manner. In addition, the ability to implement and test the system in a simulation setting can allow for safer and more conditioned real flight tests, reducing the overall risks that arise from real-world flight tests. We were able to create a new test bed for drone formation control and Adaptive Navigation. The test bed allows for the immersive and critical feature of allowing controllers to be switched and evaluated between a simulation and the hardware. This project and its results are a critical step in the pursuit of future Adaptive Navigation technology done in the RSL and outside int he general engineering community. Overall, the Adaptive Navigation test bed is useful for many future applications, and with future work done to expand the Adaptive Navigation capabilities of this framework, such a system can have a pervasive impact on the way in which drones and multi-drone systems operate.

6.2 Lessons Learned

One main lesson learned was that using services like Trac for documentation and SVN for version-control and code sharing was extremely helpful in ensuring that each team member had access to the most recent code base and that updates made to code spread throughout our implementation was accessible. The other importance of documentation allowed us to ensure we were keeping on track in completing the project in a timely manner while also following the requirements set out at the beginning of the project. These services, in addition to Zoom and other social media applications, allowed for immersive collaboration especially when in need of working as a team while being distributed about the US and sheltered-in-place due to COVID-19. Under this branch, we learned that it is important to collaborate to ensure our teamwork was supportive in ensuring the work done by each team member was truly progressing toward the accomplishment of a successful project.

Another lesson learned was that it was important to ensure testing was incremental and thorough in order to ensure

that the end-to-end system would not be prone to any error. This was also important because it ensured that no single component or device can possibly act as a single point-of-failure. Another reason for which such testing is important is that it allows for a clear and rigorous identification and removal of errors that may arise in the system. Also, incremental testing in this manner is important for ensuring the overall safety of our system especially prior to a real-world field test.

6.3 Future Work

Future work includes being able to expand to a four-drone cluster controller and Adaptive Navigation controller. Due to the constraints set out by COVID-19 it was difficult to implement a 4-drone cluster controller and Adaptive Navigation controller that could be tested through the simulation. If given more time, the first tasks would be to fully implement these control layers for a 4-drone cluster and test these layers through the simulation in Gazebo.

Such future work also involves implementing an incremental test plan to test and verify the communication links for a 3-drone and 4-drone system in the physical system layer. This would involve connecting three joysticks and four joysticks to the workstation and confirming that all the drones are reliably able to communicate with the ground station. The main goal in this scenario is ensuring that communication packets have a low drop and loss rate. This would ensure that deploying the cluster controller and Adaptive Navigation framework for a three drone and four drone cluster in the real-world setting would ensure that all safety requirements are met while also ensuring critical design goals are met concurrently.

Future work also involves utilizing more nuanced cluster formations and control mechanisms. In the three drone system a triangular cluster formation is used. However, expanding the cluster formations used in the two dimensional plane and the three dimensional plane, in the case of a four drone cluster can enable the testing of different cluster formations in correlation to the outcome of the project. This could allow us to determine certain optimal formations that work better for certain scenarios in comparison to other scenarios.

More nuanced control mechanisms would involve using control mechanisms more complex than a simple leader/-follower relationship. The reason for using a leader/follower relationship is due to the simplicity it brings to the mathematics used to carry out transformations between the robot and cluster frames, which was important in the lens of the overall timeline of this project and the need to produce tangible results for our different control layers. More complex control mechanisms could range from different placement of the origin of the cluster frame to an adaptive leader set up, where the leader changes dependent on where each drone is relative to the source being sought through Adaptive Navigation.

This could also allow for expanded Adaptive Navigation capabilities. Instead of finding a global maxima or source, the

locating of local maxima/minima, or local/global saddle points are other possible targets for the Adaptive Navigation control layer. Being able to support such different capabilities allows for a more robust and pervasive impact of Adaptive Navigation to a range of applications.

Currently, probing for user input is done through the command line in MATLAB. For future senior design teams that may take up this project, enabling a Graphical User Interface (GUI) could be useful, especially in the case of groups with limited computer engineering backgrounds. Such an interface could also reduce latency in communication between MATLAB and other system components and in the internal calculations done in MATLAB. Currently using the command window through MATLAB pauses the other processes within the Simulink simulation which could result in major issues with effectively and safely controlling the drones while in flight.

Finally, with a more robust and stress-tested system, application to the real-world of this system could be ascertained. Applications in environmental monitoring, pollution-seeking, radiation source-seeking, search and rescue, and other applications could allow for a more real-world test-bed of the system allowing for continuous improvement and updating of the system and also a tangible realization of the positive real-world impacts our project can be a part of.

Bibliography

[1] Dji agras t16. [Online]. Available: https://www.dji.com/t16

[2] (2018, Feb) Intel drone light show breaks guinness world records title at olympic winter games pyeongchang 2018. [Online]. Available: https://newsroom.intel.com/news-releases/intel-drone-light-show-breaks-guinness-world-records-title-olympic-winter-games-pyeongchang-2018

[3] C. A. Kitts and I. Mas, "Cluster space specification and control of mobile multirobot systems," *IEEE/ASME Transactions on Mechatronics*, Aug 2009.

[4] C. A. Kitts, R. T. Mcdonald, and M. A. Neumann, "Adaptive navigation control primitives for multirobot clusters: Extrema finding, contour following, ridge/trench following, and saddle point station keeping," *IEEE Access*, vol. 6, p. 17625–17642, 2018.

[5] Px4 autopilot user guide (v1.9.0). [Online]. Available: https://docs.px4.io/v1.9.0/en/

[6] (2019, Apr) Drone service market application (aerial photography, data acquisition, analytics), industry (infrastructure, media entertainment, agriculture), type (drone platform service, drone mro, drone training, solution, region - global forecast to 2025. [Online]. Available: https://www.researchandmarkets.com/reports/4763830/drone-service-market-application-aerial?

[7] (2020, Apr) Uav formations: Use and examples. [Online]. Available: https://www.embention.com/news/uav-formations-use-and-examples/

[8] F. E. Schneider and D. Wildermuth, "A potential field based approach to multi robot formation navigation," in *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings. 2003*, vol. 1, 2003, pp. 680–685 vol.1.

[9] I. Mas and C. A. Kitts, "Dynamic control of mobile multirobot systems: The cluster space formulation," *IEEE Access*, vol. 2, p. 558–570, May 2014.

[10] S. Tomer, C. Kitts, M. Neumann, R. McDonald, S. Bertram, R. Cooper, M. Demeter, E. Guthrie, E. Head, A. Kashikar, J. Kitts, M. London, A. Mahacek, R. Rasay, D. Rajabhandharaks, and T. Yeh, "A low-cost indoor testbed for multirobot adaptive navigation research," in *2018 IEEE Aerospace Conference*, 2018, pp. 1–12.

[11] C. Kitts, T. Adamek, M. Vlahos, A. Mahacek, K. Poore, J. Guerra, M. Neumann, M. Chin, and M. Rasay, "An underwater robotic testbed for multi-vehicle control," in *2014 IEEE/OES Autonomous Underwater Vehicles (AUV)*, 2014, pp. 1–8.

[12] DroneCode. (2019) Mavsdk. [Online]. Available: https://mavsdk.mavlink.io/develop/en/index.html

[13] D. Gagne. (2019) Qgroundcontrol user guide. [Online]. Available: https://docs.qgroundcontrol.com/en/

[14] MAVLink. (2018) Mavlink developer guide. [Online]. Available: https://mavlink.io/en/

[15] Intel. (2019, Nov) intel/mavlink-router. [Online]. Available: https://github.com/intel/mavlink-router

[16] MathWorks. (2019) Get matlab. [Online]. Available: https://www.mathworks.com/products/get-matlab.html?s_tid=gn_getml

Appendix A

Software Setup

A.1 Mavlink Router

A.1.1 Installation

To use the mavlink router, the first step is to install the prerequisite software from their GitHub page [15]. This can be done in the following steps.

```
git clone https://github.com/intel/mavlink-router.git
cd mavlink-router
git submodule update --init --recursive
./autogen.sh && ./configure CFLAGS='-g -O2' \
        --sysconfdir=/etc --localstatedir=/var --libdir=/usr/lib64 \
    --prefix=/usr
make
make install
```

This will install the MAVLink router on your system.

A.1.2 Run Program

To start the mavlink router, enter the following command to use the configuration file located at /etc/mavlink-router/main.conf. To see configuration file examples, check Appendix C

```
mavlink-routerd
```

A.2 MATLAB

A.2.1 Installation

To install MATLAB, follow the instructions on MathWorks website to install MATLAB version 2019A [16]. Use all default setting to prevent variance from machine to machine.

A.2.2 Run Program

To start the MATLAB, enter the following command to launch the MATLAB environment.

```
matlab
```

A.3 QGroundControl

A.3.1 Installation

To install QGroundControl, follow the instructions on their website to download the most recent version with all firmware updates [13].

A.3.2 Run Program

To start the QGroundControl, click the included app image in the location of download. This will launch the program.

Appendix B

Additional Setup for Simulation

In order to use a simulation environment for testing, it is important to install additional software and change some facets of the setup to work with the simulation

B.1 Installing Additional Software Components

Within the Linux Terminal on Ubuntu 18.04, type the following commands

```
git clone https://github.com/PX4/Firmware.git --recursive
cd Firmware
bash ./Tools/setup/ubuntu.sh
reboot
```

After the system reboots, all of the necessary Firmware should be located in the Firmware folder in the directory you called the first instruction. This should include all of the PX4 software, including all necessary packages for the PX4 SITL and the Gazebo Simulation

B.2 Building the SITL

In order to build the SITL for gazebo, run the following command from the Firmware folder.

```
make px4_sitl gazebo
```

B.3 Multiple Instances of the SITL

To use multiple SITL instances with Gazebo, run the following command from the Firmware Folder after you have run the Building the SITL command once before.

```
Tools/gazebo_sitl_multiple_run.sh [-m <model>] [-n <number_of_vehicles>] [-w <world>]
```

You can choose to do any combination of options, but the default scenario will be an empty world with 3 quad-copter drones.

Appendix C

Mavlink Router

The following code segments set up the router for a two-drone, three-drone, and four-drone system, respectively. The code particularly enables the connection to telemetry radios connected to the workstation through USB and ensures that the system IDs and component IDs are set such that messages can be forwarded to the appropriate devices within the drone subsystem.

C.1 Two Drone System

```
# mavlink-router configuration file is composed of sections,
# each section has some keys and values. They
# are case insensitive, so ‘Key=Value‘ is the same as ‘key=value‘.
#
# [This-is-a-section name-of-section]
# ThisIsAKey=ThisIsAValuye
#
# Following specifications of expected sessions and key/values.
#
# Section [General]: This section doesn’t have a name.
#
# Keys:
#   TcpServerPort
#       A numeric value defining in which port mavlink-router will
#       listen for TCP connections. A zero value disables TCP listening.
#       Default: 5760
#
#   ReportStats
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if traffic statistics should be reported.
#       Default: false
#
#   MavlinkDialect
#       One of <auto>, <common> or <ardupilotmega>. Defines which MAVLink
#       dialect is being used by flight stack, so mavlink-router can log
#       appropiately. If <auto>, mavlink-router will try to decide based
#       on heartbeat received from flight stack.
#       Default: auto
#
#   Log
#       Path to directory where to store flightstack log.
#       No default value. If absent, no flightstack log will be stored.
#
#   LogMode
#       One of <always>, <while-armed>
#       Default: always, log from start until mavlink-router is stopped.
#       If set to while-armed, a new log file is created whenever the vehicle is
#       armed, and closed when disarmed.
#
```

```
#   MinFreeSpace
#       The Log Endpoint will delete old log files until there are MinFreeSpace bytes
#       available on the storage device of the logs. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   MaxLogFiles
#       Maximum number of log files to keep. The Log Endpoint will delete old
#       log files to keep the total below this number. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   DebugLogLevel
#       One of <error>, <warning>, <info> or <debug>. Which debug log
#       level is being used by mavlink-router, with <debug> being the
#       most verbose.
#       Default:<info>
#
# Section [UartEndpoint]: This section must have a name
#
# Keys:
#   Device
#       Path to UART device, like `/dev/ttyS0`
#       No default value. Must be defined.
#
#   Baud
#       Numeric value stating baudrate of UART device
#       Default: 115200
#
#   FlowControl
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if flow control should be enabled
#       Default: false
#
#
# Section [UdpEndpoint]: This section must have a name
#
# Keys:
#   Address
#       If on `Normal` mode, IP to which mavlink-router will
#       route messages to (and from). If on `Eavesdropping` mode,
#       IP of interface to which mavlink-router will listen for
#       incoming packets. In this case, `0.0.0.0` means that
#       mavlink-router will listen on all interfaces.
#       No dafault value. Must be defined.
#
#   Mode
#       One of <normal> or <eavesdropping>. See `Address` for more
#       information.
#       No default value. Must be defined
#
#   Port
#       Numeric value defining in which port mavlink-router will send
#       packets (or listen for them).
#       Default value: Increasing value, starting from 14550, when
#       mode is `Normal`. Must be defined if on `Eavesdropping` mode.
#
# Section [TcpEndpoint]: This section must have a name
#
# Keys:
#   Address:
#       IP to which mavlink-router will connect to.
#       No default value. Must be defined.
#
#   Port:
#       Numeric value with port to which mavlink-router will connect to.
#       No dafault value. Must be defined.
#
#   RetryTimeout:
#       Numeric value defining how many seconds mavlink-router should wait
```

```
#       to reconnect to IP in case of disconnection. A value of 0 disables
#       reconnection.
#       Default value: 5
#
# Following, an example of configuration file:
[General]
#Mavlink-router serves on this TCP port
TcpServerPort=5790
ReportStats=false
MavlinkDialect=auto

[UdpEndpoint alfa]
Mode = Normal
Address = 127.0.0.1
Port = 14550

[UartEndpoint bravo]
Device = /dev/ttyUSB0
Baud = 115200

[UartEndpoint charlie]
Device = /dev/ttyUSB1
Baud = 115200
```

C.2 Three Drone System

```
# mavlink-router configuration file is composed of sections,
# each section has some keys and values. They
# are case insensitive, so ‘Key=Value‘ is the same as ‘key=value‘.
#
# [This-is-a-section name-of-section]
# ThisIsAKey=ThisIsAValuye
#
# Following specifications of expected sessions and key/values.
#
# Section [General]: This section doesn’t have a name.
#
# Keys:
#   TcpServerPort
#       A numeric value defining in which port mavlink-router will
#       listen for TCP connections. A zero value disables TCP listening.
#       Default: 5760
#
#   ReportStats
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if traffic statistics should be reported.
#       Default: false
#
#   MavlinkDialect
#       One of <auto>, <common> or <ardupilotmega>. Defines which MAVLink
#       dialect is being used by flight stack, so mavlink-router can log
#       appropiately. If <auto>, mavlink-router will try to decide based
#       on heartbeat received from flight stack.
#       Default: auto
#
#   Log
#       Path to directory where to store flightstack log.
#       No default value. If absent, no flightstack log will be stored.
#
#   LogMode
#       One of <always>, <while-armed>
#       Default: always, log from start until mavlink-router is stopped.
#       If set to while-armed, a new log file is created whenever the vehicle is
#       armed, and closed when disarmed.
#
#   MinFreeSpace
```

```
#       The Log Endpoint will delete old log files until there are MinFreeSpace bytes
#       available on the storage device of the logs. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   MaxLogFiles
#       Maximum number of log files to keep. The Log Endpoint will delete old
#       log files to keep the total below this number. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   DebugLogLevel
#       One of <error>, <warning>, <info> or <debug>. Which debug log
#       level is being used by mavlink-router, with <debug> being the
#       most verbose.
#       Default:<info>
#
# Section [UartEndpoint]: This section must have a name
#
# Keys:
#   Device
#       Path to UART device, like `/dev/ttyS0`
#       No default value. Must be defined.
#
#   Baud
#       Numeric value stating baudrate of UART device
#       Default: 115200
#
#   FlowControl
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if flow control should be enabled
#       Default: false
#
#
# Section [UdpEndpoint]: This section must have a name
#
# Keys:
#   Address
#       If on `Normal` mode, IP to which mavlink-router will
#       route messages to (and from). If on `Eavesdropping` mode,
#       IP of interface to which mavlink-router will listen for
#       incoming packets. In this case, `0.0.0.0` means that
#       mavlink-router will listen on all interfaces.
#       No dafault value. Must be defined.
#
#   Mode
#       One of <normal> or <eavesdropping>. See `Address` for more
#       information.
#       No default value. Must be defined
#
#   Port
#       Numeric value defining in which port mavlink-router will send
#       packets (or listen for them).
#       Default value: Increasing value, starting from 14550, when
#       mode is `Normal`. Must be defined if on `Eavesdropping` mode.
#
# Section [TcpEndpoint]: This section must have a name
#
# Keys:
#   Address:
#       IP to which mavlink-router will connect to.
#       No default value. Must be defined.
#
#   Port:
#       Numeric value with port to which mavlink-router will connect to.
#       No dafault value. Must be defined.
#
#   RetryTimeout:
#       Numeric value defining how many seconds mavlink-router should wait
#       to reconnect to IP in case of disconnection. A value of 0 disables
```

```
#       reconnection.
#       Default value: 5
#
# Following, an example of configuration file:
[General]
#Mavlink-router serves on this TCP port
TcpServerPort=5790
ReportStats=false
MavlinkDialect=auto

[UdpEndpoint alfa]
Mode = Normal
Address = 127.0.0.1
Port = 14550

[UartEndpoint bravo]
Device = /dev/ttyUSB0
Baud = 115200

[UartEndpoint charlie]
Device = /dev/ttyUSB1
Baud = 115200

[UartEndpoint delta]
Device = /dev/ttyUSB2
Baud = 115200
```

C.3 Four Drone System

```
# mavlink-router configuration file is composed of sections,
# each section has some keys and values. They
# are case insensitive, so ‘Key=Value‘ is the same as ‘key=value‘.
#
# [This-is-a-section name-of-section]
# ThisIsAKey=ThisIsAValuye
#
# Following specifications of expected sessions and key/values.
#
# Section [General]: This section doesn't have a name.
#
# Keys:
#   TcpServerPort
#       A numeric value defining in which port mavlink-router will
#       listen for TCP connections. A zero value disables TCP listening.
#       Default: 5760
#
#   ReportStats
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if traffic statistics should be reported.
#       Default: false
#
#   MavlinkDialect
#       One of <auto>, <common> or <ardupilotmega>. Defines which MAVLink
#       dialect is being used by flight stack, so mavlink-router can log
#       appropiately. If <auto>, mavlink-router will try to decide based
#       on heartbeat received from flight stack.
#       Default: auto
#
#   Log
#       Path to directory where to store flightstack log.
#       No default value. If absent, no flightstack log will be stored.
#
#   LogMode
#       One of <always>, <while-armed>
#       Default: always, log from start until mavlink-router is stopped.
#       If set to while-armed, a new log file is created whenever the vehicle is
```

```
#       armed, and closed when disarmed.
#
#   MinFreeSpace
#       The Log Endpoint will delete old log files until there are MinFreeSpace bytes
#       available on the storage device of the logs. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   MaxLogFiles
#       Maximum number of log files to keep. The Log Endpoint will delete old
#       log files to keep the total below this number. Set to 0 to ignore this limit.
#       Default: 0 (disabled)
#
#   DebugLogLevel
#       One of <error>, <warning>, <info> or <debug>. Which debug log
#       level is being used by mavlink-router, with <debug> being the
#       most verbose.
#       Default:<info>
#
# Section [UartEndpoint]: This section must have a name
#
# Keys:
#   Device
#       Path to UART device, like `/dev/ttyS0`
#       No default value. Must be defined.
#
#   Baud
#       Numeric value stating baudrate of UART device
#       Default: 115200
#
#   FlowControl
#       Boolean value <true> or <false> case insensitive, or <0> or <1>
#       defining if flow control should be enabled
#       Default: false
#
#
# Section [UdpEndpoint]: This section must have a name
#
# Keys:
#   Address
#       If on `Normal` mode, IP to which mavlink-router will
#       route messages to (and from). If on `Eavesdropping` mode,
#       IP of interface to which mavlink-router will listen for
#       incoming packets. In this case, `0.0.0.0` means that
#       mavlink-router will listen on all interfaces.
#       No dafault value. Must be defined.
#
#   Mode
#       One of <normal> or <eavesdropping>. See `Address` for more
#       information.
#       No default value. Must be defined
#
#   Port
#       Numeric value defining in which port mavlink-router will send
#       packets (or listen for them).
#       Default value: Increasing value, starting from 14550, when
#       mode is `Normal`. Must be defined if on `Eavesdropping` mode.
#
# Section [TcpEndpoint]: This section must have a name
#
# Keys:
#   Address:
#       IP to which mavlink-router will connect to.
#       No default value. Must be defined.
#
#   Port:
#       Numeric value with port to which mavlink-router will connect to.
#       No dafault value. Must be defined.
#
```

```
#   RetryTimeout:
#       Numeric value defining how many seconds mavlink-router should wait
#       to reconnect to IP in case of disconnection. A value of 0 disables
#       reconnection.
#       Default value: 5
#
# Following, an example of configuration file:
[General]
#Mavlink-router serves on this TCP port
TcpServerPort=5790
ReportStats=false
MavlinkDialect=auto

[UdpEndpoint alfa]
Mode = Normal
Address = 127.0.0.1
Port = 14550

[UartEndpoint bravo]
Device = /dev/ttyUSB0
Baud = 115200

[UartEndpoint charlie]
Device = /dev/ttyUSB1
Baud = 115200

[UartEndpoint delta]
Device = /dev/ttyUSB2
Baud = 115200

[UartEndpoint echo]
Device = /dev/ttyUSB3
Baud = 115200
```

Appendix D

Controller Source Code

D.1 Setup

The following code segment depicts what was run to initiate the system in Simulink's state diagram. In particular, the mavlink object and mavlink dialects are created to enable usage of mavlinkio commands later on in the state diagram. The mavlinkio commands are used to send joystick commands to move the drones. Other commands are also used to send a hold command to the drones, send heartbeats to QGroundControl, to set a global reference frame, to retrieve messages from the drones about status, command retrieval acknowledgement, and to retrieve the local positions for each of the drones relative to the global reference frame.

```
function setup
    coder.extrinsic('createsubs','mySub','createmsg', 'pause', 'input', 'mavlinkclient', 'empty',
     'Controllers', 'xboxController', 'mavlinkdialect', 'mavlinkio', 'connect', 'listClients', '
    height', 'disp', 'mavlinksub');
    n = 3;
    gps = double(49);
    local = double(32);
    target = double(85);
    dialect = mavlinkdialect("~/common.xml");
    mavlink = mavlinkio(dialect);
    connect(mavlink,"UDP", 'LocalPort',14551);
    pause(2);
    clients = listClients(mavlink);
    controllers = [xboxController(1,clients(1,1)) xboxController(2,clients(1,2)) xboxController
    (3,clients(1,3))];
    manual_cluster = xboxController(1,clients(1,1:3));
    [gps_origins, local_positions, local_targets] = createsubs(mavlink, n, gps, local, target);
    Cdot = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
    pause(1);
    disp('Success');
end
```

D.2 Read Joysticks

The following code segment obtains joystick inputs directly from the joysticks and makes the appropriate transformation to place it in the range necessitated by the MANUAL_CONTROL mavlinkio message to send to the drones. Here we read the joysticks' inputs with respect to the axes orientation of the joystick and the button states on the joysticks. The corresponding button states are used to update two variables transfers and regains to dictate whether or not a state transition is needed in the state diagram. The axes information is used to set up a MANUAL_CONTROL packet that is then sent to the drone corresponding to which joystick is being read at that instance of time.

```
function read_joysticks
  coder.extrinsic('controlValues', 'send_command');
  i = 0;
```

```
  if cluster == 1
    [inputs, transfer, regain]=controlValues(manual_cluster);
    for i = 1:n
        send_command(mavlink,dialect,inputs,i)
    end
    %but1_4 = double(buttons(4));
    return;
  end
  for i = 1:n
    if holds(i) == 1
        [inputs, transfer, regain]=controlValues(controllers(1,i));
        regains(i) = regain;
    else
        [inputs, transfer, regain]=controlValues(controllers(1,i));
        send_command(mavlink,dialect,inputs, i);
        transfers(i) = transfer;
    end
  end
end
```

D.3 Send Heartbeat

The code segment that follows sets up a heartbeat packet to be sent from MATLAB to QGroundControl. This is necessary to maintain communication between MATLAB and QGroundControl ultimately ensuring that the drones are able to receive messages from MATLAB and respond correspondingly. Here the message is simply set up based on the rules of the HEARTBEAT message itself and then the message is sent out.

```
function send_heartbeat
    coder.extrinsic('createmsg', 'sendmsg');
    Payload = struct('custom_mode', uint32(0), 'type', uint8(0), 'autopilot', uint8(0), '
    base_mode', uint8(0),'system_status', uint8(0), 'mavlink_version', uint8(0));
    heartbeat = struct('MsgID', uint32(0), 'Payload', Payload);
    heartbeat = createmsg(dialect,0);
    sendmsg(mavlink,heartbeat);
end
```

D.4 Set Global Reference Frame

The following code segment creates a mavlink message to set the global reference frame for all the drones in the cluster to be controlled. Then a return message is awaited upon from each of the drones confirming that the GPS global origin that the drones sync to is the same as that which was set for the global reference frame.

```
function set_ref_frame(sids,params)
   for i = 1:length(sids)
       Payload = struct('target_system', uint8(0), 'latitude', int32(0), 'longitude', int32(0), '
    altitude', int32(0), 'time_usec', uint64(0));
       msg = struct('MsgID', uint32(0), 'Payload', Payload);

       % Create message to set the global origin
       msg = createmsg(dialect, 48);
       msg.Payload.target_system = uint8(sids(i));
       msg.Payload.latitude = int32(params(1));
       msg.Payload.longitude = int32(params(2));
       msg.Payload.altitude = int32(params(3));

       while(1)
           sendmsg(mavlink, msg)

           pause(1);

           % Listen for Global GPS Origin message from drone
```

```
            sub = mavlinksub(mavlink, mavlinkclient(mavlink,2,1), "GPS_GLOBAL_ORIGIN");

            while(1)
                gpsmsg = latestmsgs(sub,1);
                if (~isempty(gpsmsg))
                    break;
                end
            end

            % Check to see if received GPS global origin message from drone
            % matches gps values sent when setting global origin
            if (gpsmsg.param1 == params(1) & gpsmsg.param2 == params(2) & gpsmsg.param3 == params
    (3))
                break;
            end
        end
    end
end
```

D.5 Receive Drone Positions

The code segment that follows is used to receive the drone's local positions in relation to the global reference frame that was set. The received positions are saved into variables that contain the position for each drone in the cluster. These individual positions are then later concatenated to create the robot position vector to be used in the forward kinematic equations for the cluster controller as described in Equations 3.3 to 3.10.

```
function position = receive_drone_pos(sid, local_positions, local_targets)

    x = double(0);
    y = double(0);
    z = double(0);
    head = double(0);

    pos = [];
    heading = [];

    % Listen for drone's local position with respect to global frame
    pos = currentPayload(local_positions(sid));
    heading = currentPayload(local_targets(sid));


    x = pos(3);
    y = pos(2); %+3*(sid-1); If drone set ref frame soes not work, %an offset would need to be
    provided. This issue was %noticed when working in simulation
    z = -pos(4);
    head = heading(11);

    position = [double(x) double(y) double(z) double(head)];
end
```

D.6 Initialize Cluster

The following code segment simply initializes the cluster variables that will be used throughout the state diagram in all cluster and Adaptive Navigation-related calculations. This function exists primarily because of Stateflow as a software requiring the initialization of variables that are used in the Stateflow chart to be done prior to setting them from the output of a function or in some other similar method of assignment. Here the maximum speeds in each of the possible directions are set to convert joystick values to values with tangible units to be used in the cluster space equations.

```
function initialize_cluster
    R = [0 0 0 0 0 0 0 0 0 0 0 0];
    C = [0 0 0 0 0 0 0 0 0 0 0 0];
```

```
    Rt = [0 0 0 0 0 0 0 0 0 0 0 0];
    Cdot = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
    Rdot = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
    Cdes2 = [0; 0; 0; 0];
    Cdes3 = [0; 0; 0; 0];
    max_speedx = 1.9;
    max_speedy = 1.9;
    max_speedzup = 2.9;
    max_speedzdown = 1.0;
    max_yawrate = 2.618;
end
```

D.7 Read Cluster

The code segment that follows reads the main joystick and sets up the values of the cluster velocity command that correspond to a proportional control law as described in Equations 3.20 to 3.23 in the Subsystem Design chapter. Here the user is probed through the command window in MATLAB to enter the desired values for dx, dz, ϕ, and α, as well as the K values for each of these values as well. Once these are set up, these values are used in conjunction with the inputs from the main joystick controlling the cluster to calculate the corresponding robot velocity commands that are then sent to the drones to commence corresponding motion.

```
function read_cluster
    coder.extrinsic('controlCluster', 'jac_vel', 'input', 'send_command');
    if cluster == 1
        transitions = [double(0) double(0) double(0) double(0) double(0) double(0) double(0)
    double(0) ...
            double(0) double(0) double(0)];
        [vels, transitions]=controlCluster(controllers(1,1), max_speedx, max_speedy, max_speedzup
    , max_speedzdown, max_yawrate);
        but1_4 = double(transitions(4));
        transitions = logical(transitions);
        K2 = eye(4);
        K3 = eye(4);
        L2 = eye(4);
        L3 = eye(4);

        % To be changed depending on testing info corresponds to dx dz phi alpha, respectively
        A = [0.2 0.2 pi/16 pi/12];
        Cdes2(1) = input("Please enter a desired dx position value for drone 2...\n");
        K2(1, 1) = input("Please enter a desired K value for the dx term for drone 2...\n");
        Cdes3(1) = input("Please enter a desired dx position value for drone 3...\n");
        K3(1, 1) = input("Please enter a desired K value for the dx term for drone 3...\n");
        %Cdot(5) = K(1,1)*(Cdes(1) - C(5));
        L2(1, 1) = 0;
        L3(1, 1) = 0;

        Cdes2(2) = input("Please enter a desired dz position value for drone 2...\n");
        K2(2, 2) = input("Please enter a desired K value for the dz term for drone 2...\n");
        Cdes3(2) = input("Please enter a desired dz position value for drone 3...\n");
        K3(2, 2) = input("Please enter a desired K value for the dz term for drone 3...\n");
        %Cdot(6) = K(2,2)*(Cdes(2) - C(6));
        L2(2, 2) = 0;
        L3(2, 2) = 0;

        Cdes2(3) = input("Please enter a desired phi heading value for drone 2...\n");
        K2(3, 3) = input("Please enter a desired K value for the phi term for drone 2...\n");
        Cdes3(3) = input("Please enter a desired phi heading value for drone 3...\n");
        K3(3, 3) = input("Please enter a desired K value for the phi term for drone 3...\n");
        %Cdot(7) = K(3,3)*(Cdes(3) - C(7));
        L2(3, 3) = 0;
        L3(3, 3) = 0;

        Cdes2(4) = input("Please enter a desired alpha heading value for drone 2...\n");
        K2(4, 4) = input("Please enter a desired K value for the alpha term for drone 2...\n");
```

```
        Cdes3(4) = input("Please enter a desired alpha heading value for drone 3...\n");
        K3(4, 4) = input("Please enter a desired K value for the alpha term for drone 3...\n");
        %Cdot(8) = K(4,4)*(Cdes(4) - C(8));
        L2(4, 4) = 0;
        L3(4, 4) = 0;

    end
end
```

D.8 Robot Velocity Transform

This function calculates the Jac^{-1} used to calculate the robot velocity commands as described in Equation 3.25 and calculates the robot velocity commands using this Jac^{-1} and the $\dot{C}$ cluster velocity command. In the following code segment, transforms to convert the robot velocity commands to valid commands that can be used in the MANUAL_CONTROL mavlink message are also done.

```
function Rdot = jac_vel3(n,R,C,Cdot, max_speedx, max_speedy, max_speedzup, max_speedzdown,
    max_yawrate)
    J=[1 0 0 0 0 0 0 0;
    0 1 0 0 0 0 0 0;
    0 0 1 0 0 0 0 0;
    0 0 0 1 0 0 0 0;
    (R(1)-R(5))/C(5) (R(2)-R(6))/C(5) 0 0 (R(5)-R(1))/C(5) (R(6)-R(2))/C(5) 0 0;
    0 0 -1 0 0 0 1 0;
    (R(2)-R(6))/C(5)^2 (R(5)-R(1))/C(5)^2 0 -1 (R(6)-R(2))/C(5)^2 (R(1)-R(5))/C(5)^2 0 0;
    0 0 0 0 0 0 0 1];

    invJ21=[1 0 0 0 0 0 0 0;
    0 1 0 0 0 0 0 0;
    0 0 1 0 0 0 0 0;
    0 0 0 1 0 0 0 0;
    1 0 0 -C(5)*sin(pi/2-C(4)-C(7)) -cos(pi/2-C(4)-C(7)) 0 -C(5)*sin(pi/2-C(4)-C(7)) 0;
    0 1 0 C(5)*cos(pi/2-C(4)-C(7)) -sin(pi/2-C(4)-C(7)) 0 C(5)*cos(pi/2-C(4)-C(7)) 0;
    0 0 1 0 0 1 0 0;
    0 0 0 0 0 0 0 1];

    invJ31=[1 0 0 0 0 0 0 0;
    0 1 0 0 0 0 0 0;
    0 0 1 0 0 0 0 0;
    0 0 0 1 0 0 0 0;
    1 0 0 -C(9)*sin(pi/2-C(4)-C(11)) -cos(pi/2-C(4)-C(11)) 0 -C(9)*sin(pi/2-C(4)-C(11)) 0;
    0 1 0 C(9)*cos(pi/2-C(4)-C(11)) -sin(pi/2-C(4)-C(11)) 0 C(9)*cos(pi/2-C(4)-C(11)) 0;
    0 0 1 0 0 1 0 0;
    0 0 0 0 0 0 0 1];

%     Cdot(5) = 1000 * Cdot(5);
%     Cdot(6) = 500 * Cdot(6);
%     Cdot(7) = 1000*Cdot(7);
%     Cdot(8) = 1000*Cdot(8);
    Rdot(1:8) = invJ21*Cdot(1:8);
    Rnew = [0; 0; 0; 0; 0; 0; 0; 0];
    Rnew = invJ31*(cat(1,Cdot(1:4), Cdot(9:12)));
    Rdot(9:12) = Rnew(5:8);
%     if(abs(Rdot(5)) > 1)
%         Rdot(1) = Rdot(1)/abs(Rdot(5));
%         Rdot(5) = Rdot(5)/abs(Rdot(5));
%     end
%     if(abs(Rdot(6)) > 1)
%         Rdot(2) = Rdot(2)/abs(Rdot(6));
%         Rdot(6) = Rdot(6)/abs(Rdot(6));
%     end
%     if(abs(Rdot(7)) > 1)
%         Rdot(3) = Rdot(3)/abs(Rdot(7));
%         Rdot(7) = Rdot(7)/abs(Rdot(7));
```

```
%     end
%     if(abs(Rdot(8)) > 1)
%         Rdot(4) = Rdot(4)/abs(Rdot(8));
%         Rdot(8) = Rdot(8)/abs(Rdot(8));
%     end

    % Normalize values
    Rdot(1) = Rdot(1)/max_speedx;
    Rdot(5) = Rdot(5)/max_speedx;
    Rdot(9) = Rdot(9)/max_speedx;
    Rdot(2) = Rdot(2)/max_speedy;
    Rdot(6) = Rdot(6)/max_speedy;
    Rdot(10) = Rdot(10)/max_speedy;
    if(Rdot(3) >= 0)
        Rdot(3) = Rdot(3)/max_speedzdown;
    else
        Rdot(3) = Rdot(3)/max_speedzup;
    end
    if(Rdot(7) >= 0)
        Rdot(7) = Rdot(7)/max_speedzdown;
    else
        Rdot(7) = Rdot(7)/max_speedzup;
    end
    if(Rdot(11) >= 0)
        Rdot(11) = Rdot(11)/max_speedzdown;
    else
        Rdot(11) = Rdot(11)/max_speedzup;
    end
    Rdot(4) = Rdot(4)/(max_yawrate);
    Rdot(8) = Rdot(8)/(max_yawrate);
    Rdot(12) = Rdot(12)/max_yawrate;

    % Transform values to manual_control ranges
    for i = 1:(4*n)
        if (mod(i,4) == 1)
            Rdot(i) = -1000 * Rdot(i);
        elseif (mod(i,4) == 0 | mod(i,4) == 2)
            Rdot(i) = 1000 * Rdot(i);
        else
            Rdot(i) = ((-1000 * Rdot(i))+1000)/2;
        end
    end

    Rdot;
    %Rdot(5) = 1000 * Rdot(5);
    %Rdot(6) = 1000 * Rdot(6);
    %Rdot(7) = ((1000 * abs(Rdot(7)-500))+1000)/2;
    %Rdot(8) = 1000 * Rdot(8);
end
```

D.9 Send Cluster

The following code segment starts with the congregation and final setting of the cluster velocity command. After the robot position values, cluster position values, and cluster velocity commands are used to calculate the robot velocity commands. Once the robot velocity commands are set up using the jac_vel function, as detailed above, the commands can be sent to the corresponding drones.

```
function send_clustercom3(Cdot, Rdot, n, mavlink, dialect, controllers, K2, K3, Cdes2, Cdes3, C,
    L2, L3, A, R, max_speedx, max_speedy, max_speedzup, max_speedzdown, max_yawrate)
    coder.extrinsic('send_clusters','jac_vel3', 'send_command', 'controlCluster');
    [vels, transitions]=controlCluster(controllers(1,1), max_speedx, max_speedy, max_speedzup,
    max_speedzdown, max_yawrate);
    Cdot(1:4) = vels;
    Cdot(5:8) = (K2*(Cdes2 - C(5:8)')) + (L2*(A'));
    Cdot(9:12) = (K3*(Cdes3 - C(9:12)')) + (L3*(A'));
```

```
    Rdot = jac_vel3(n,R,C,Cdot, max_speedx, max_speedy, max_speedzup, max_speedzdown, max_yawrate
    );

    send_clusters(mavlink, dialect, Rdot, n);
end
```

D.10 Calculate Scalar Field Value

The code segment that follows calculates the scalar field values through the simulated sensor that reads these values from the scalar field set up in the terrain. These scalar field values are later used to calculate gradients.

```
function value = calculate_scalar_2d(x,y)
    magnitude = sqrt(x^2 + y^2);
    value = 1/(magnitude^2);
end
```

D.11 Calculate Gradient

The following code segment calculates the gradient for each drone in the cluster to act as the corresponding cluster velocity commands to be used later in transitioning to the robot velocity command. The gradient is used to efficiently move the drone cluster toward the source in the terrain.

```
function unit_vector = grad_calc( R1, R2, R3 )
 %GRAD_CALC Calculates a gradient based on robot data
 %   Using equations straight from the Thomas tmech paper

R12 = R2 - R1;
R13 = R3 - R1;

N = cross(-R12, R13);
grad = [N(1); N(2); 0];

unit_vector = grad/norm(grad);

end
```

D.12 Send Adaptive Navigation

The following code segment is highly similar to the prior code segment that sends commands to the cluster. The only difference in this function is that the initial values of the cluster velocity commands are not set by the values read from the first joystick controller but instead comes from the calculated gradients in the Adaptive Navigation control layer.

```
function send_AN3(Cdot, Rdot, n, mavlink, dialect, unit_vector, K2, K3, Cdes2, Cdes3, C, L2, L3,
    A, R, max_speedx, max_speedy, max_speedzup, max_speedzdown, max_yawrate)
    coder.extrinsic('send_clusters','jac_vel3', 'send_command', 'controlCluster');
    vels = [-unit_vector(1)*max_speedx -unit_vector(2)*max_speedy 0 0];
    Cdot(1:4) = vels;
    Cdot(5:8) = (K2*(Cdes2 - C(5:8)')) + (L2*(A'));
    Cdot(9:12) = (K3*(Cdes3 - C(9:12)')) + (L3*(A'));
    Rdot = jac_vel3(n,R,C,Cdot, max_speedx, max_speedy, max_speedzup, max_speedzdown, max_yawrate
    );

    send_clusters(mavlink, dialect, Rdot, n);
end
```

D.13 Controller Class

The following code segment is the basis for all controller code within our system. It allows for any controller, regardless of axes and buttons, to inherit from the Controller class as a base class and then specify which axes and buttons are used for control of the individual drones in our system.

```
classdef Controller
    properties
        Joy
        Drones
       % Current
    end
    methods
        function obj = Controller(jid, drone_list)
            coder.extrinsic('vrjoystick', 'read', 'button');
            obj.Joy = vrjoystick(jid);
            obj.Drones = drone_list;
            %obj.Current = 0;
        end
        function obj = addDrone(obj, drone)
            obj.Drones = [obj.Drones, drone];
        end

        function obj = removeDrone(obj, drone)
            for c = 1:length(obj.Drones)
                if( obj.Drones(c) == drone)
                   obj.Drones(c) = [];
                   return;
                end
            end
        end

        function obj = newList(obj, newDrones)
            obj.Drones = newDrones;
        end

        function obj = clearDrones(obj)
            obj.Drones = [];
        end

%         function obj = changeAccess(obj)
%             if obj.Current == -1
%                 disp("Controller " + obj.Joy.jid + " has no drone access")
%                 return;
%             elseif current == length(obj.Drones)
%                 obj.Current = 0;
%                 disp("Controller " + obj.Joy.jid + " has access to all of its drones")
%             else
%                 obj.Current = obj.Current + 1;
%                 disp("Controller " + obj.Joy.jid + " is controlling drone number" + obj.Drones(
    obj.Current).SystemID)
%             end
%         end
%
%         function obj = changeControl(obj)
%             if obj.Current == -1
%                 obj.Current = 0;
%                 disp("Controller " + obj.Joy.jid + " regained control of its drones")
%             else
%                 obj.Current = -1;
%                 disp("Controller " + obj.Joy.jid + " forfeited control of its drones")
%             end
%         end
%
        function [output, transfer, regain] = controlValues(obj)
           disp("Function has not be overloaded");
        end
```

```
%
%         function droneList = dronesControlled (obj)
%            if obj.Current == -1
%                droneList = [];
%            elseif obj.Current == 0
%                droneList = obj.Drones;
%            else
%                droneList = obj.Drones(obj.Current);
%            end
%         end
    end
end
```

D.14 xboxController Class

The xboxController class inherits from the Controller class to overload the necessary functions to control either an individual drone or a cluster of drones. Additionally it inverses any controls to properly account for the builtin signs of the controller.

```
classdef xboxController < Controller
    properties
    end
    methods
        function obj = xboxController(jid, drone_list)
            obj = obj@Controller(jid, drone_list);
        end

        function [output, transfer, regain] = controlValues(obj)
            transfer = 0;
            regain = 0;
            [axes, button] = read(obj.Joy);
            for c = 1:length(axes)
                if(axes(c) < 0.1 && axes(c) > -0.1)
                    axes(c) = 0.0;
                end
            end

            if(button(1) == 1)
                transfer = 1;
            end
            if(button(2) == 1)
                regain = 1;
            end
            output = rand(1,4);
            output(1) = -1000 * axes(2);
            output(2) = 1000 * axes(1);
            output(3) = ((-1000 * axes(5))+1000)/2;
            output(4) = 1000 * axes(4);
            output = int16(output);
        end

        function [vel, transition] = controlCluster(obj, max_speedx, max_speedy, max_speedzup,
    max_speedzdown, max_yawrate)
            [axes, button] = read(obj.Joy);
            for c = 1:length(axes)
                if(axes(c) < 0.1 && axes(c) > -0.1)
                    axes(c) = 0.0;
                end
            end

            transition = double(button);
            vel = rand(1,4);
%             vel(1) = axes(2);
%             vel(2) = axes(1);
%             vel(3) = axes(5);
%             vel(4) = axes(4);
```

```
47              vel(1) = axes(2)*max_speedx;
48              vel(2) = axes(1)*max_speedy;
49              if(axes(5)>=0)
50                  vel(3) = axes(5)*max_speedzdown;
51              else
52                  vel(3) = axes(5)*max_speedzup;
53              end
54              vel(4) = axes(4)*max_yawrate;
55              vel = int16(vel);
56          end
57      end
58  end
```

Appendix E

Arduino Source Code

Multiple programs were written to test the full functionality of our scalar field devices. The communication link between the on-board Arduino Mega and the control program in MATLAB was tested by printing out and analyzing all packet transmission. The connection between the drones and the beacon was also tested to ensure that signal strength could be accurately recorded.

Listed below the debugging programs is the source code for the two separate Arduino devices in our system. Mounted on each drone is an Arduino Mega running the scalar field sensor program for sensor data transmission. The field beacon for Adaptive Navigation is running a separate program to broadcast an RF signal at a set interval.

E.1 MAVLink Protocol Test

The following code was written to test the functionality of the MAVLink network within our system. In the setup portion of the program, the identification of the device is set and the Serial ports for debugging and Pixhawk link are enabled at a baud rate of 115200. The main loop of this program continuously checks for new packets to receive and broadcasts a heartbeat every second. If a packet is available in the receiving buffer from the ground station, a packet containing test scalar data is transmitting in response.

```
/*
 * Test heartbeat packets between mavlink systems
 *
 * Broadcast heartbeat every 1000ms
 *   broadcast period defined by WAITING
 * Check for incoming mavlink packets
 *   If packet available, read and parse into packet struct
 *     If packet is a heartbeat, print out the information
 *       If packet is a heartbeat from ground station(sysid=255), respond with RSSI test packet
 */

#include "mavlink.h"

#define WAITING 1000

/*
 * Send/receive mavlink packets from PixHawk over Serial connection
 *
 * pins:
 *  Rx(17): telem2-2(TX)(yellow)
 *  Tx(16): telem2-3(RX)(green)
 *
 * current process: send heartbeat at power up
 *                  broadcast heartbeat every 1sec
 *                  broadcast RSSI every heartbeat received from MATLAB
 */

```

```
// MAVLINK PACKET CONFIGURATION
// - - - -
// system and component id of arduino
// sysid must be unique across entire system
// compid must at least be unique within each drone
uint8_t sysid = 1;    // match to drone system id
uint8_t compid = 20;  // set to first value listed in private network in mavlink documentation

// initialize send message buffer
mavlink_message_t msg;
uint8_t buf[MAVLINK_MAX_PACKET_LEN];
uint16_t len;

// timing global variables
unsigned long curr, prev;


// setup runs at board power up and reset
void setup() {
  // initialize Serial ports
  // Serial for Serial port debugging  (pins: 0/1)
  // Serial2 for PX4 communication     (mega pins: 16/17)
  Serial.begin(115200);
  Serial2.begin(115200);
  while (!Serial);
  while (!Serial2);

  curr = 0;
  prev = 0;

  // broadcast heartbeat msg so that other MavLink systems register Arduino as device on network
  // allows for PX4 to forward msgs between TELEM1 and TELEM2 based on sysid
  // possibly registered by mavlink-routerd as well
  sendHeartbeat();

} // setup()


// send a named int packet
void sendRSSI() {
  uint8_t target_sys_id;  // sysid of MAVsdk
  uint8_t target_comp_id; // compid of MAVsdk (needed?)

  uint32_t sys_time = millis();  // time program has been running in ms
  const char *val_name = "RSSI";
  int32_t rssi_val = 10;  // save value of RSSI here, rssi_val packed into message

  // pack values into mavlink msg and send to buffer
  // write buffer across Serial connection to PixHawk
  mavlink_msg_named_value_int_pack(sysid, compid, &msg, sys_time, val_name, rssi_val);
  len = mavlink_msg_to_send_buffer(buf, &msg);
  Serial2.write(buf, len);
  Serial.println("RSSI sent");

} // sendRSSI()


// broadcast a heartbeat msg to the PX4
void sendHeartbeat() {
  // system config (for heartbeat)
  uint8_t system_type = MAV_TYPE_GENERIC;
  uint8_t autopilot_type = MAV_AUTOPILOT_INVALID;
  uint8_t system_mode = MAV_MODE_PREFLIGHT;  // possible conflict with pixhawk if sysid matches?
  uint8_t system_state = MAV_STATE_BOOT;
  uint32_t custom_mode = 0;

  // package and populate buffer with heartbeat msg
```

```
  mavlink_msg_heartbeat_pack(sysid, compid, &msg, system_type, autopilot_type, system_mode,
    custom_mode, system_state);
  len = mavlink_msg_to_send_buffer(buf, &msg);

  // send msg over Serial2 connection
  Serial2.write(buf, len);
  Serial.println("heartpacket sent");
} // sendHeartbeat()


// read incoming msgs and parse if expected case outlined
void commReceive() {
  // local variables
  mavlink_message_t msg_rec;
  mavlink_status_t status_rec;
  uint8_t c;

  // while data available, parse it
  while(Serial2.available() > 0) {  // possible to get stuck in while if Serial buffer fills
    c = Serial2.read();
    // parse new message if available
    if(mavlink_parse_char(MAVLINK_COMM_0, c, &msg_rec, &status_rec)) {
      // handle message
      switch(msg_rec.msgid) {
        // case for each type of message to be received

        case MAVLINK_MSG_ID_HEARTBEAT:
          //sendHeartbeat();
          // broadcast RSSI value when MATLAB(255) broadcasts heartbeat
          if(msg_rec.sysid == 255) {
            sendRSSI();  // broadcast RSSI value each time heartbeat received
            Serial.print("heartbeat received: ");
            Serial.println(msg_rec.sysid);
          }
          break;

      } // switch
    } // if
  } // while
} // comm_receive()


// main loop
void loop() {
  curr = millis();

  if(Serial2.available() > 0)
    commReceive();
  // check receive buffer for incoming msgs every 1000ms
  if(curr - prev > WAITING) {
    sendHeartbeat();
    //commReceive();
    prev = curr;
  }
} // loop()
```

E.2 Parsing Packets

During our system integration and testing, it was necessary to display the packet transfers in real time. The program used to test the handshake procedures was altered to print out packet information to the Serial monitor. Within the commReceive() function, the system id and message id of all packets is printed out. After this information is displayed, packets of interest are printed out within the state machine based on the message id. This allowed us to view all control input sent to the drone to ensure that it was structured and received correctly.

```
/*
 * Parse and display mavlink messages within drone system
 *
 * Arduino system id matches system id of pixhawk on drone
 * Arduino component id is set to 20
 *   Broadcast heartbeat packet during setup()
 *
 * Broadcast heartbeat packets every 500ms
 *
 * Parse received mavlink packets
 *   Print out system id, message id, and packet contents
 *
 */

// to change dialect: change includes within mavlink.h to point towards dialect folder
#include "mavlink.h"

#define WAITING 500


// MAVLINK PACKET CONFIGURATION
// - - - -
// system and component id of arduino
// sysid must be unique across entire system
// compid must at least be unique within each drone
uint8_t sysid = 2;   // set to pixhawk sysid of drone
uint8_t compid = 20; // set to same value for all field sensors
// initialize send message buffer
mavlink_message_t msg;
uint8_t buf[MAVLINK_MAX_PACKET_LEN];
uint16_t len;

// timing global variables
unsigned long curr, prev;

// setup runs at power up and reset
void setup() {
  // initialize Serial ports
  // Serial for Serial port debugging  (pins: 0/1)
  // Serial2 for PX4 communication     (pins: 16/17)
  Serial.begin(115200);
  Serial2.begin(115200);
  while (!Serial);
  while (!Serial2);
  Serial.println("setup begin");

  curr = 0;
  prev = 0;

  // broadcast heartbeat msg so that other MavLink systems register Arduino as device on network
  // allows for PX4 to forward msgs between TELEM1 and TELEM2 based on sysid
  // possibly registered by mavlink-routerd as well
  sendHeartbeat();

} // setup()


// send RSSI value to ground station
void sendRSSI() {
  uint8_t target_sys_id = 255;    // sysid of MAVsdk
  uint8_t target_comp_id  = 100;  // compid of MAVsdk (needed?)

  uint32_t sys_time = millis();   // time program has been running in ms
  const char *val_name = "RSSI";
  int32_t rssi_val = 10;          // test value for end-to-end verification

  // pack values into mavlink msg and send to buffer
  // write buffer across Serial connection to PixHawk
```

```
  mavlink_msg_named_value_int_pack(sysid, compid, &msg, sys_time, val_name, rssi_val);
  len = mavlink_msg_to_send_buffer(buf, &msg);
  Serial2.write(buf, len);
  Serial.println("RSSI sent");

} // sendRSSI()


// broadcast a heartbeat msg
void sendHeartbeat() {
  // system config (for heartbeat)
  uint8_t system_type = MAV_TYPE_GENERIC;
  uint8_t autopilot_type = MAV_AUTOPILOT_INVALID;
  uint8_t system_mode = MAV_MODE_PREFLIGHT;
  uint8_t system_state = MAV_STATE_BOOT;
  uint32_t custom_mode = 0;

  // package and populate buffer with heartbeat msg
  mavlink_msg_heartbeat_pack(sysid, compid, &msg, system_type, autopilot_type, system_mode,
    custom_mode, system_state);
  len = mavlink_msg_to_send_buffer(buf, &msg);

  // send msg over Serial2 connection
  Serial2.write(buf, len);
  Serial.println("heartpacket sent");
} // sendHeartbeat()


// read incoming msgs and parse if expected case outlined
void commReceive() {
  // local variables
  mavlink_message_t msg_rec;
  mavlink_status_t status_rec;
  uint8_t c;

  // while data available, parse it
  while(Serial2.available() > 0) {
    c = Serial2.read();
    // parse new message if available
    if(mavlink_parse_char(MAVLINK_COMM_0, c, &msg_rec, &status_rec)) {
      // handle message

      // print out sys id and msg id of all packets
      Serial.print("\nsys id: ");
      Serial.print(msg_rec.sysid);
      Serial.print("\tmsg id: ");
      Serial.println(msg_rec.msgid);

      // print out sysid and msgid of packets from matlab
      /*if(msg_rec.sysid > 1 && msg_rec.sysid != 51) {
        Serial.print("\nsys id: ");
        Serial.print(msg_rec.sysid);
        Serial.print("\tmsg id: ");
        Serial.println(msg_rec.msgid);
      }*/

      switch(msg_rec.msgid) {
        // case for each type of message to be received

        case MAVLINK_MSG_ID_HEARTBEAT:
          //Serial.print("\theartbeat packet\n");
          if(msg_rec.sysid == 255 && msg_rec.compid == 1) {
            //sendRSSI();  // broadcast RSSI value each time heartbeat received
            Serial.println("Send RSSI to 255 here");
          }
        break;

        case MAVLINK_MSG_ID_COMMAND_LONG:  // COMMAND_LONG: 76
```

```
        Serial.println(msg_rec.sysid);
        Serial.print("* command received: ");
        Serial.println(mavlink_msg_command_long_get_command(&msg_rec));
          if(mavlink_msg_command_long_get_command(&msg_rec) == 512) {  // request data
            // currently unused
          }
          if(mavlink_msg_command_long_get_command(&msg_rec) == 252) {
            Serial.print("\ttarget system: ");
            Serial.println(mavlink_msg_command_long_get_target_system(&msg_rec));
            Serial.print("\ttarget component: ");
            Serial.println(mavlink_msg_command_long_get_target_component(&msg_rec));
            Serial.print("\tcommand: ");
            Serial.println(mavlink_msg_command_long_get_command(&msg_rec));
            Serial.print("\tparam1: ");
            Serial.println(mavlink_msg_command_long_get_param1(&msg_rec));
            Serial.print("\tparam2: ");
            Serial.println(mavlink_msg_command_long_get_param2(&msg_rec));
          }
        break;

        case MAVLINK_MSG_ID_MANUAL_CONTROL:  // retest with decode func
          Serial.print("\ttarget system: ");
          Serial.println(mavlink_msg_manual_control_get_target(&msg_rec));
          Serial.print("\tx: ");
          Serial.print(mavlink_msg_manual_control_get_x(&msg_rec));
          Serial.print("\ty: ");
          Serial.print(mavlink_msg_manual_control_get_y(&msg_rec));
          Serial.print("\tz: ");
          Serial.print(mavlink_msg_manual_control_get_z(&msg_rec));
          Serial.print("\tr: ");
          Serial.println(mavlink_msg_manual_control_get_r(&msg_rec));
          Serial.print("\tbuttons: ");
          Serial.println(mavlink_msg_manual_control_get_buttons(&msg_rec));
        break;

      } // switch
    } // if
  } // while
} // comm_receive()


// main loop
void loop() {
  curr = millis();

  if(Serial2.available() > 0)
    commReceive();
  // check receive buffer for incoming msgs every 1000ms
  if(curr - prev > WAITING) {
    sendHeartbeat();
    //commReceive();
    prev = curr;
  }
} // loop()
```

E.3 Recording RSSI Values

This program was written to test the functionality of recording signal strength of the beacon. Once the beacon is running, a scalar field sensor running this code will print out its recorded signal strength as each packet is received.

```
/**
 * Record RSSI values
 *
 *   Continuously reads for broadcasted packets
 *   Prints out signal strength of received packets
```

```
 *
 */

#include <XBee.h>
#include <SoftwareSerial.h>
/*
This example is for Series 1 XBee (802.15.4)
Receives either a RX16 or RX64 packet and sets a PWM value based on packet data.
Error led is flashed if an unexpected packet is received
*/

XBee xbee = XBee();
XBeeResponse response = XBeeResponse();
// create reusable response objects for responses we expect to handle
Rx16Response rx16 = Rx16Response();
Rx64Response rx64 = Rx64Response();

SoftwareSerial XBeeS(2,3);

uint8_t option = 0;
uint8_t data = 0;
int rssi;

void setup() {

  // start serial
  Serial.begin(9600);
  XBeeS.begin(9600);
  xbee.setSerial(XBeeS);

}

// continuously reads packets, looking for RX16 or RX64
void loop() {

    xbee.readPacket();

    if (xbee.getResponse().isAvailable()) {
      // got something
      if (xbee.getResponse().getApiId() == RX_16_RESPONSE || xbee.getResponse().getApiId() ==
    RX_64_RESPONSE) {
        // got a rx packet
        if (xbee.getResponse().getApiId() == RX_16_RESPONSE) {
                xbee.getResponse().getRx16Response(rx16);
                Serial.println(rx16.getRssi());
        } else {
                xbee.getResponse().getRx64Response(rx64);
                Serial.println(rx64.getRssi());
        }
      }
    }
}
```

E.4 Scalar Field Sensor

The program below is the current source code for the scalar field sensor of this project. Heartbeats are broadcast every 2.5 seconds to the ground station. Each device has a component ID of 20 and a system ID matching that of the drone it is mounted to. The function getRSSI() handles recording scalar data from the beacon and the function sendRSSI() is called when a heartbeat from the ground station is received. Within sendRSSI(), the scalar values are averaged and packaged within a NAMED_INT message. The function sendHeartbeat() handles populating and transmitting a heartbeat message. The commReceive() function checks the buffer for available messages. If an incoming message is a heartbeat from the ground station the RSSI value is transmitted. All other incoming packets are ignored.

```
/*
 * Send average of last 3 RSSI values to Ground Station
 *
 * Currently runs on Arduino Mega
 *   Uses all 3 of the ahrdware Serial ports availbe on the Mega
 *   MAVLink library takes up majority of used memory
 *
 */

#include "mavlink.h"
#include <XBee.h>

#define WAITING 2500

// MAVLINK PACKET CONFIGURATION
// - - - -
// system and component id of arduino
// sysid must be unique across entire system
// compid must at least be unique within each drone
uint8_t sysid = 1;    // match to drone system id
uint8_t compid = 20;  // set to first value listed in private network in mavlink documentation
// initialize send message buffer
// make local if dynamic memory fills
mavlink_message_t msg;
uint8_t buf[MAVLINK_MAX_PACKET_LEN];
uint16_t len;

// XBEE CONFIGURATION
// - - - -
XBee xbee = XBee();
XBeeResponse repsone = XBeeResponse();
// create reusable response objects for responses we expect to handle
Rx16Response rx16 = Rx16Response();
Rx64Response rx64 = Rx64Response();


// global variables
unsigned long curr, prev;
int rssi;
uint8_t option;
uint8_t data;
int rssi_vals[3];


// setup runs at board power up and reset
void setup() {
  // initialize Serial ports
  // Serial:  Serial port debugging (pins: 0/1)
  // Serial2: PX4 communication     (pins: 16/17)
  // Serial3: XBee port             (pins:14/15)
  Serial.begin(115200);
  while (!Serial);
  Serial2.begin(115200);
  Serial3.begin(115200);
  // initialize XBee
  xbee.setSerial(Serial3);

  curr = 0;
  prev = 0;
  option = 0;
  data = 0;
  rssi = -1;

  // broadcast heartbeat msg so that other MavLink systems register Arduino as device on network
  // allows for PX4 to forward msgs between TELEM1 and TELEM2 based on sysid
  sendHeartbeat();

} // setup()
```

```
69
70
71  // record the current RSSI value from the Xbee module
72  // save the last three values recorded
73  void getRSSI() {
74    xbee.readPacket();
75    // shift recorded values
76    rssi_vals[2] = rssi_vals[1];
77    rssi_vals[1] = rssi_vals[0];
78
79    if (xbee.getResponse().isAvailable()) {
80      // got something
81      if (xbee.getResponse().getApiId() == RX_16_RESPONSE || xbee.getResponse().getApiId() ==
        RX_64_RESPONSE) {
82        // got a rx packet
83        if (xbee.getResponse().getApiId() == RX_16_RESPONSE) {
84          xbee.getResponse().getRx16Response(rx16);
85          rssi_vals[0] = rx16.getRssi();
86        }
87        else {
88          xbee.getResponse().getRx64Response(rx64);
89          rssi_vals[0] = rx64.getRssi();
90        }
91      }
92    }
93  }  // getRSSI()
94
95
96  // send a named int packet
97  void sendRSSI() {
98    uint8_t target_sys_id = 255;    // sysid of MAVsdk
99    uint8_t target_comp_id = 100;   // compid of MAVsdk (needed?)
100
101   uint32_t sys_time = millis();   // time program has been running in ms
102   const char *val_name = "RSSI";
103   int32_t rssi_val = (rssi_vals[0] + rssi_vals[1] + rssi_vals[2]) / 3;  // send the average of
        the last three values recorded
104
105   // pack values into mavlink msg and send to buffer
106   // write buffer across Serial connection to PixHawk
107   mavlink_msg_named_value_int_pack(sysid, compid, &msg, sys_time, val_name, rssi_val);
108   len = mavlink_msg_to_send_buffer(buf, &msg);
109   Serial2.write(buf, len);
110   Serial.print("RSSI sent: ");
111   Serial.println(rssi_val);
112
113 } // sendRSSI()
114
115
116 // broadcast a heartbeat msg to the PX4
117 void sendHeartbeat() {
118   // system config (for heartbeat)
119   uint8_t system_type = MAV_TYPE_GENERIC;
120   uint8_t autopilot_type = MAV_AUTOPILOT_INVALID;
121   uint8_t system_mode = MAV_MODE_PREFLIGHT;
122   uint8_t system_state = MAV_STATE_BOOT;
123   uint32_t custom_mode = 0;
124
125   // package and populate buffer with heartbeat msg
126   mavlink_msg_heartbeat_pack(sysid, compid, &msg, system_type, autopilot_type, system_mode,
        custom_mode, system_state);
127   len = mavlink_msg_to_send_buffer(buf, &msg);
128
129   // send msg over Serial2 connection
130   Serial2.write(buf, len);
131   Serial.println("heartpacket sent");
132 } // sendHeartbeat()
133
```

```
134
135 // read incoming msgs and parse if expected case outlined
136 void commReceive() {
137   // local variables
138   mavlink_message_t msg_rec;
139   mavlink_status_t status_rec;
140   uint8_t c;
141
142   // while data available, parse it
143   while(Serial2.available() > 0) {  // possible to get stuck in while if Serial buffer fills
144     c = Serial2.read();
145
146     // parse new message if available
147     //if(mavlink_parse_char(MAVLINK_COMM_0, c, &msg_rec, &status_rec)) {
148       // handle message
149       int ret = mavlink_parse_char(MAVLINK_COMM_0, c, &msg_rec, &status_rec);
150
151       // print out incoming packets for debugging
152       /*if(msg_rec.sysid) {
153         Serial.print("sysid: ");
154         Serial.print(msg_rec.sysid);
155         Serial.print("\tcompid: ");
156         Serial.print(msg_rec.compid);
157         Serial.print("\tmsgid: ");
158         Serial.println(msg_rec.msgid);
159       }*/
160
161       switch(msg_rec.msgid) {
162         // case for each type of message to be received
163
164         case MAVLINK_MSG_ID_HEARTBEAT:
165           //sendHeartbeat();
166           // broadcast RSSI value when MATLAB(255) broadcasts heartbeat
167           if(msg_rec.sysid == 255) {
168             getRSSI();
169             sendRSSI();  // broadcast RSSI value each time heartbeat received
170             Serial.print("heartbeat received: ");
171             Serial.println(msg_rec.sysid);
172             //Serial.println("heartbeat reply");
173           }
174           break;
175
176       } // switch
177     //} // if
178   } // while
179 } // comm_receive()
180
181
182 // main loop
183 void loop() {
184   curr = millis();
185
186   if(Serial2.available() > 0) {
187     commReceive();
188   }
189   // check receive buffer for incoming msgs every 1000ms
190   if(curr - prev > WAITING) { // add count to shift frequency
191     getRSSI();
192     sendHeartbeat();
193     //commReceive();
194     prev = curr;
195   }
196 } // loop()
```

E.5 Beacon

The code below runs on the beacon placed out in the field for Adaptive Navigation usage. The arduino will broadcast through the attached Xbee module every fifteen seconds. If the packets are received by another Xbee module, a success acknowledgment is printed. This block was kept in the final implementation to be able to verify the Xbee link was functioning properly in the case of sensor reading failure on any of the drones.

```
  /**
 * Copyright (c) 2009 Andrew Rapp. All rights reserved.
 *
 * This file is part of XBee-Arduino.
 *
 * XBee-Arduino is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * XBee-Arduino is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with XBee-Arduino.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <XBee.h>
#include <SoftwareSerial.h>
/*
This example is for Series 1 XBee
Sends a TX16 or TX64 request with the value of analogRead(pin5) and checks the status response for success
Note: In my testing it took about 15 seconds for the XBee to start reporting success, so I've added a startup delay
*/

XBee xbee = XBee();
SoftwareSerial XBeeS(2, 3);

unsigned long start = millis();

// allocate two bytes for to hold a 10-bit analog reading
uint8_t payload[] = { 0, 0, 0, 0, 0, 0 };

// with Series 1 you can use either 16-bit or 64-bit addressing

// 16-bit addressing: Enter address of remote XBee, typically the coordinator
Tx16Request tx = Tx16Request(0xFFFF, payload, sizeof(payload));

// 64-bit addressing: This is the SH + SL address of remote XBee
//XBeeAddress64 addr64 = XBeeAddress64(0x0013a200, 0x4008b490);
// unless you have MY on the receiving radio set to FFFF, this will be received as a RX16 packet
//Tx64Request tx = Tx64Request(addr64, payload, sizeof(payload));

TxStatusResponse txStatus = TxStatusResponse();


void setup() {
  Serial.begin(9600);
  XBeeS.begin(9600);
  xbee.setSerial(XBeeS);

}


void loop() {
```

```
58
59    // start transmitting after a startup delay.  Note: this will rollover to 0 eventually so not
      best way to handle
60    if (millis() - start > 15000) {
61      // break down 10-bit reading into two bytes and place in payload
62      payload[0] = 0x42;
63      payload[1] = 0x65;
64      payload[2] = 0x61;
65      payload[3] = 0x63;
66      payload[4] = 0x6F;
67      payload[5] = 0x6E;
68
69      xbee.send(tx);
70    }
71
72    // after sending a tx request, we expect a status response
73    // wait up to 5 seconds for the status response
74    if (xbee.readPacket(5000)) {
75        // got a response!
76
77        // should be a znet tx status
78      if (xbee.getResponse().getApiId() == TX_STATUS_RESPONSE) {
79         xbee.getResponse().getTxStatusResponse(txStatus);
80         // get the delivery status, the fifth byte
81           if (txStatus.getStatus() == SUCCESS) {
82              Serial.println("Success");
83           }
84        }
85    }
86    delay(5000);
87  }
```

Appendix F

Flight Procedure and Checklists

Safety Equipment:

- ☐ Long pants
- ☐ Closed-toed shoes
- ☐ Safety glasses
- ☐ Long hair must be tied back
- ☐ Remove any loose clothing, such as scarves, that could get caught in spinning parts
- ☐ Visible safety vest

Briefing:

- ☐ Remote Pilot in Command informs the team of the mission
- ☐ Remote Pilot in Command assigns team tasks
  - ☐ The Visual Observer is responsible for remaining by the computer or smartphone/tablet to monitor telemetry connection and vehicle status.
  - ☐ The Visual Observer monitors vehicle and surroundings. For tethered flight, another the Visual Observer is in charge of monitoring the tether and making sure it has enough slack during flight.
  - ☐ **It is everyone and anyone's responsibility to notify the rest of the team of impending hazards**

Controller Pre-Flight Checks:

- ☐ Check QGroundControl configuration
- ☐ Laptop screen dim is off
- ☐ Laptop Sleep-mode off
- ☐ Ensure proper binding of telemetry radios
- ☐ Load flight plan if required

Vehicle Pre-flight / Startup Procedures:

- ☐ Install propellers
  - ☐ Verify that the nuts securing propellers in place are tight
- ☐ Ensure propellers are in good condition

- ☐ If any changes have been made to the motor mounts, ensure the motors are secure, will not vibrate, and have sufficient clearance for all moving parts
- ☐ Check for loose wires
- ☐ Nuts and bolts are tight
- ☐ Frame components are secure
- ☐ For tethered flight:
  - ☐ Install tether using approved line and attachment mechanism
  - ☐ Unwind tether 4 meters
  - ☐ Make sure tether attachment is below the craft
- ☐ The Remote Pilot in Command should notify the team that he/she is plugging in the battery and then plug it in (no one else should be powering the vehicle). Watch for the propellers to twitch.
- ☐ Check QGroundControl:
  - ☐ Ensure drone is recognized
  - ☐ Ensure drone is calibrated and properly positioned
  - ☐ Check Parameters are set as intended
  - ☐ Check battery is within operating limits
  - ☐ For tethered flight, Install tether using approved line and attachment mechanism
- ☐ Set and maintain the throttle at zero
- ☐ Clear away from the vehicle
  - ☐ Remote Pilot in Command should stand 3+ meters/yards away, and the rest of the team should stand behind the Remote Pilot in Command
  - ☐ For tethered flight, the Visual Observer should have clear line of sight of the vehicle at all times and keep 4 meters of slack in the tether
- ☐ Arm the drone
  - ☐ Ensure drone arms properly
  - ☐ Debug any errors and recheck parameters when they arise
- ☐ Remote Pilot in Command notifies the team (loud & audible) that he is powering up & slowly increases the throttle
  - ☐ Increase the throttle to about 20% and wait for all motors to spin up
  - ☐ Once all motors have spun up, increase the throttle as needed for tests
  - ☐ If at any point once throttle is applied anyone sees or smells smoke or hears or sees excess vibration or clearance issues, throttle off or tell the Remote Pilot in Command to throttle off immediately
- ☐ Perform tuning or flight testing procedure
  - ☐ Visual Observer should monitor battery levels to prevent battery damage and avoid an in-flight battery failure. When the X8 reaches 15% battery life remaining, the Remote Pilot in Command should return to base or land depending on location and distance to the home base.
- ☐ When your mission is complete:
  - ☐ Remote Pilot in Command lands the vehicle
  - ☐ Remote Pilot in Command disarms vehicle
  - ☐ Stay at least a meter/yard away from the vehicle until drone is confirmed to be disarmed
  - ☐ Unplug the battery before working with or transporting the vehicle
  - ☐ For tethered flight, rewind and detach the tether

Appendix G

Final Presentation Slides

The next pages of the document show the presentations slides from the 2020 Senior Design Conference. The presentation was part of Interdisciplinary Session 1 and was presented by all the members of our team.

SANTA CLARA UNIVERSITY
School of Engineering

Adaptive Navigation Utilizing a Drone Cluster

Team members: Zach Cameron, Brendan Engh, Thomas Kambe, Aditya Krishnan

Advisors: Dr. Christopher Kitts, Dr. Sally Wood

1

Presentation Outline

- Introduction
- System Overview
- Subsystems
  - Drone Platform and Sensors
  - Communications Protocol
  - Control System Architecture
- Results
- Summary

2

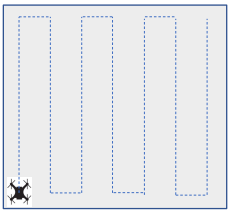
Drones: Uses and Capabilities

- Market Expansion
  - Videography
  - Defense
  - Agriculture
  - Delivery
  - Many more
- Advanced Capabilities
  - Drone Swarms
  - Adaptive Navigation

3

Adaptive Navigation (AN)

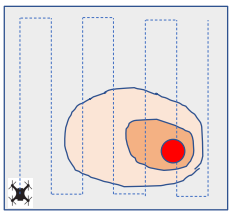
- "Mow the Lawn"
  - Used for mapping and visual searches



4

Adaptive Navigation (AN)

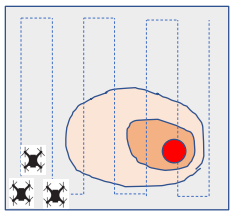
- "Mow the Lawn"
  - Used for mapping and visual searches
- Adaptive Navigation
  - Use sensor data to intelligently move



5

Adaptive Navigation (AN)

- "Mow the Lawn"
  - Used for mapping and visual searches
- Adaptive Navigation
  - Use sensor data to intelligently move
  - Formations can do this most efficiently

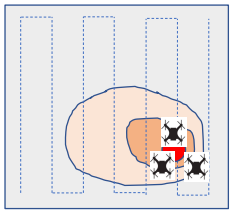


6

Adaptive Navigation (AN)

- "Mow the Lawn"
  - Used for mapping and visual searches
- Adaptive Navigation
  - Use sensor data to intelligently move
  - Formations can do this most efficiently

Challenge: experimentally demonstrate new 3D adaptive navigation techniques



7

Project Objective

- Our original objective was to implement the communication and control systems for a group of four drones to fly in formation and perform 3d adaptive navigation
  - End-to-end experimental testbed
  - Multilayered control system
  - Safety
- COVID Impact: No flight testing, Simulate 3 drone navigation

8

SANTA CLARA UNIVERSITY
School of Engineering

System Overview

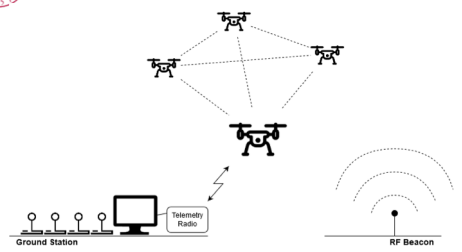
9

System: Requirements

- Communication Range: Up to 100m
- Human operator or automated control of 1-4 drones.
- Control positioning within +/- 3m
- Simulate Drone Cluster
- Maximum-seeking AN in three dimensions
- Implement critical safety features

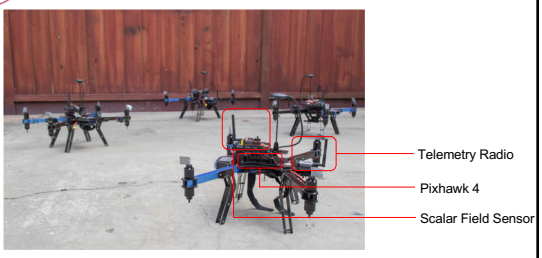
10

System: Concept of Operations



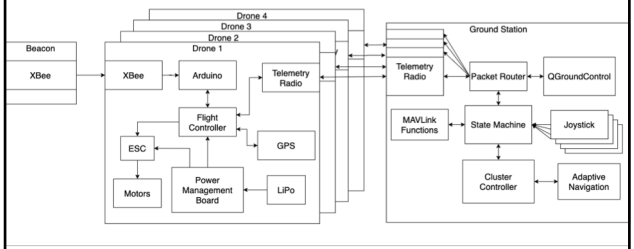
11

System: Mechanical Configuration



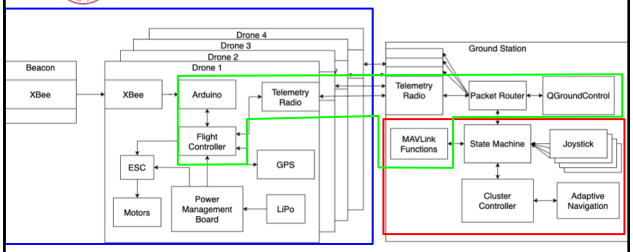
12

System: Component Block Diagram



13

System: Functional Subsystems



14

SANTA CLARA UNIVERSITY
School of Engineering

Subsystem: Drone

15

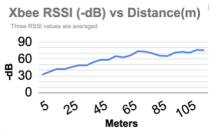
Drone: Flight System

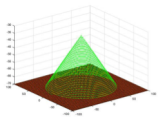
- Rebuilt/upgraded 4 drones
- Integrated new telemetry radios
- Integrated scalar field sensor
- Tested each system

16

Drone: Scalar Field Sensor

- Scalar Field RF Signal Strength
- Battery powered beacon as source
  - Approx linear field slope from 5- 75 meters
- Radio receiver system on each drone
  - XBee3 Pro Radio & Arduino Mega
  - Signal strength relayed to central controller





17

SANTA CLARA UNIVERSITY
School of Engineering

Subsystem: Communications

18

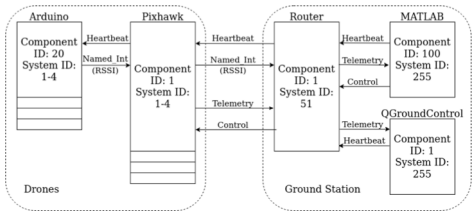
Communication: Objectives

- Drones
  - Send positional information and signal strength
  - Receive control commands
- Ground Station
  - Receive telemetry data
  - Send Control commands
  - Identify individual devices
- Solution: MAVLink Protocol

MAVLINK
MICRO AIR VEHICLE COMMUNICATION PROTOCOL

19

Communication: Packet Flow



20

SANTA CLARA UNIVERSITY
School of Engineering

Subsystem: Control

21

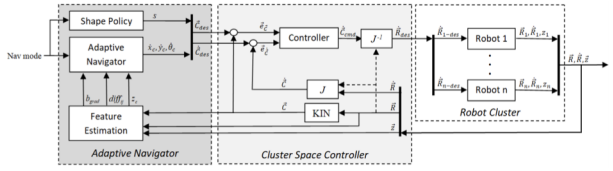
Control: Overview

The control architecture is:

- **Multilayered:**
  - Drone control via on-board autopilot
  - Formation control via centralized Matlab/Simulink program
  - Adaptive navigation control via centralized Matlab/Simulink program
- **Configurable**
  - Layers can be selectively applied to subsets of 1-4 drones
  - Layer input commands can be from a pilot or a higher layer controller

22

Control: Multilayered Architecture



--------- Centralized Matlab/Simulink program ---------- --- On-board autopilot ----

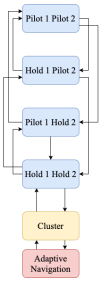
23

Control: Stationary Piloting Test

24

Control: Configurability

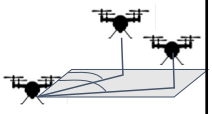
- **Pilot can specify what control layers apply to which drones**
- **A state machine formally manages this control configuration**
- **We have a methodical procedure to:**
  - Take off drones individually and place them in auto position hold
  - Add drones to a formation
  - Command a formation to implement adaptive navigation



25

Control: Formation Motion

- **Fully control geometry and motion of a group of drones**
- **Formal kinematic transforms**
  - Relate cluster positions to drone positions
  - Relate cluster velocities to drone velocities
- **Allows a pilot or controller to command the shape and motion of the cluster**
  - Control law issues cluster commands
  - Cluster commands converted to drone commands



Leader-follower cluster definition
3 robot example

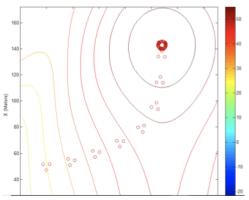
4 DOF/drone
16 DOFs for 4 drone formation

32 kinematic equations computed in the real-time control loop

26

Control: Adaptive Navigation (AN)

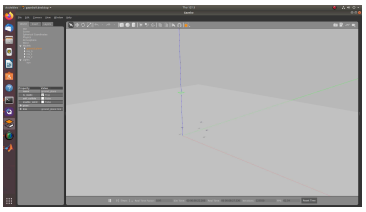
- **Control formation geometry for a high quality gradient computation**
  - 3 drones: Planar triangle
  - 4 drones: Pyramid
- **Collect signal strength values**
- **Calculate the local gradient**
  - 3 drones: Two dimensional
  - 4 drones: Three dimensional
- **Drive cluster along gradient**



27

Control: 3 Drone AN

- **COVID-19 challenge:**
  - Simulation only
  - 3 drones
- **3 drone planar formation control and adaptive navigation**
  - Drone formation travels in plane to location above the source



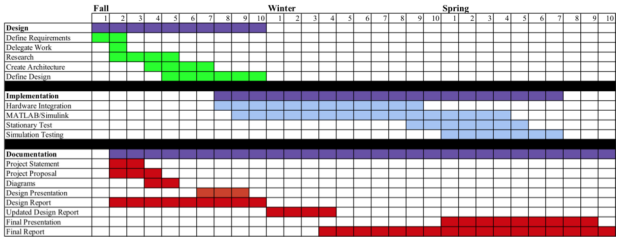
28

Project Management

- **Verification of Requirements**
  - Functional demonstrations
  - Reliability and safety criteria
- **Professional tools/standards**
  - Trac: Project documentation
  - SVN: Software version control and code-sharing
- **Distributed teamwork due to COVID-19**
  - Detailed management of scheduled development tasks
  - Online communications

29

Project Management: Schedule



30

Future Work

- Implement 4-drone cluster control and AN
- Implement incremental test plan
- Utilize other cluster formations and control mechanisms
- Expand AN capabilities
- Enhanced User Interface
  - GUI
- Application of framework to real missions
  - Environmental Monitoring

31

Summary

- **Objectives/Requirements**
  - Multidrone AN experimental testbed
- **Results**
  - Retrofitted 4 drones with RF sensors
  - Developed multipoint communication system
  - Hardware test of end-to-end communications with 2 drones
  - Implemented simulation of 3 drone cluster control and Adaptive Navigation
- **Contribution**
  - Flight ready testbed for future Adaptive Navigation research
  - Research article to be developed

32

Acknowledgments

Thank you to The SCU School of Engineering Undergraduate Programs for providing financial support.

Thank you for the SCU Robotic Systems Lab for providing financial support and a workspace to execute our project.

Thank you to Dr. Kitts and Dr. Wood for providing their expertise and support throughout the project.

33

Thank You!

34

Bibliography

C. A. Kitts, R. T. Mcdonald, and M. A. Neumann, "Adaptive Navigation Control Primitives for Multirobot Clusters: Extrema Finding, Contour Following, Ridge/Trench Following, and Saddle Point Station Keeping," IEEE Access, vol. 6, pp. 17625–17642.

I. Mas and C. A. Kitts, "Dynamic Control of Mobile Multirobot Systems: The Cluster Space Formulation," IEEE Access, vol. 2, pp. 558–570, May 2014.

T. Adamek, C. A. Kitts, and I. Mas, "Gradient-Based Cluster Space Navigation for Autonomous Surface Vessels," IEEE/ASME Transactions on Mechatronics, vol. 20, no. 2, pp. 506–518, Apr. 2015.

35