Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Final Report

REVISION – 2.0
28 April 2019

Table of Contents

Zach Drone

Elias Duque
Grayson Vansickle
Justin Armenta
Jonah Morris

Concept of Operations

REVISION – 2.0
28 April 2019

CONCEPT OF OPERATIONS
FOR
Zach Drone

PREPARED BY:
TEAM <49>

APPROVED BY:

_____
Project Leader Date

_____
Prof. S. Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/21/18	Justin Armenta		Added Introduction and Scenario(s) sections
0.4	9/24/18	Jonah Morris		Added Executive Summary and Operating Concept sections
0.6	9/24/18	Elias Duque		Added Impacts section
0.8	9/26/18	Grayson Vansickle		Added Analysis section
1.0	10/3/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information
2.0	4/28/19	Grayson Vansickle		Formatted
2.2	5/1/19	Jonah Morris		Updated information

Table of Contents

List of Tables

No tables at this time.

List of Figures

1. Executive Summary

Navigating a new building as large as Zachry by yourself can be difficult. Traditional tours with a tour guide do not always work around an individual's schedule. With the Zach Drone, tourists can take a tour anytime to predefined locations inside of Zachry. Using a phone application, tourists will be able to tell the drone to start, stop and pause. While on the tour the drone will narrate important information about the building such as where it is going next and details of different art pieces in the building. After twenty minutes of use, the drone will return to the tour starting point where it can be recharged for the next set of tourists.

2. Introduction

Zachry is the newest and most innovative of the Texas A&M engineering buildings. As the flagship of all the engineering buildings it is presented as a mixture of technology, innovation, and convenience. Since the building exemplifies these values, the tours given in the facility should convey these same values. This bold and innovative style is what inspired us to make a drone tour inside of the Zachry building. Guests will not have to wait for scheduled tours by people, but instead can ask for a quick and easy drone tour where the drone will tell customers key information about the building while also guiding its prospective tourists around the complex.

2.1. Background

Currently, tours are given by tour guides provided by the Texas A&M Engineering department’s communication team. It usually takes around 30 minutes for these employees to give a tour, and they have to give these tours at least once a day. These are hours wasted every day for them where they are giving tours instead o bf doing other helpful jobs for the department. It also can be an inconsistent tour if the tour guide forgets about any of the key locations or information about the building. Also, the tour times would not be flexible since a human has to run on a scheduled time to give the tour every day with no room for flexibility. A drone tour of Zachry would solve these problems by offering a more customizable and innovative experience. The drone would free up employees to do other more important jobs that the department is working on. It would also allow tourists to choose the destinations they would like to see and have all the information about that destination told to them by the drone. Lastly, it would allow customers to choose a more convenient time for them to get their tour instead of being told by Texas A&M when they can come see the tour.

2.2. Overview

The drone will consist of a custom built drone made by a previous engineering team, Raspberry Pi 3 B microcontroller, a speaker, three ultrasonic sensors, and a camera. The Raspberry Pi will be used to process information received by the ultrasonic sensors and the WiFi connection from the tourist’s device. The microcontroller will connect to the camera to make sure the tourist’s stay close while also guiding the tourist towards the next destination in the tour. The ultrasonic sensors will monitor how close the drone gets to obstacles while the microcontroller will give commands to the drone to either stop or change its course of

direction if an obstacle gets too close. A speaker will be used to give information to the customer about different destination points while also explaining where the drone is going next.

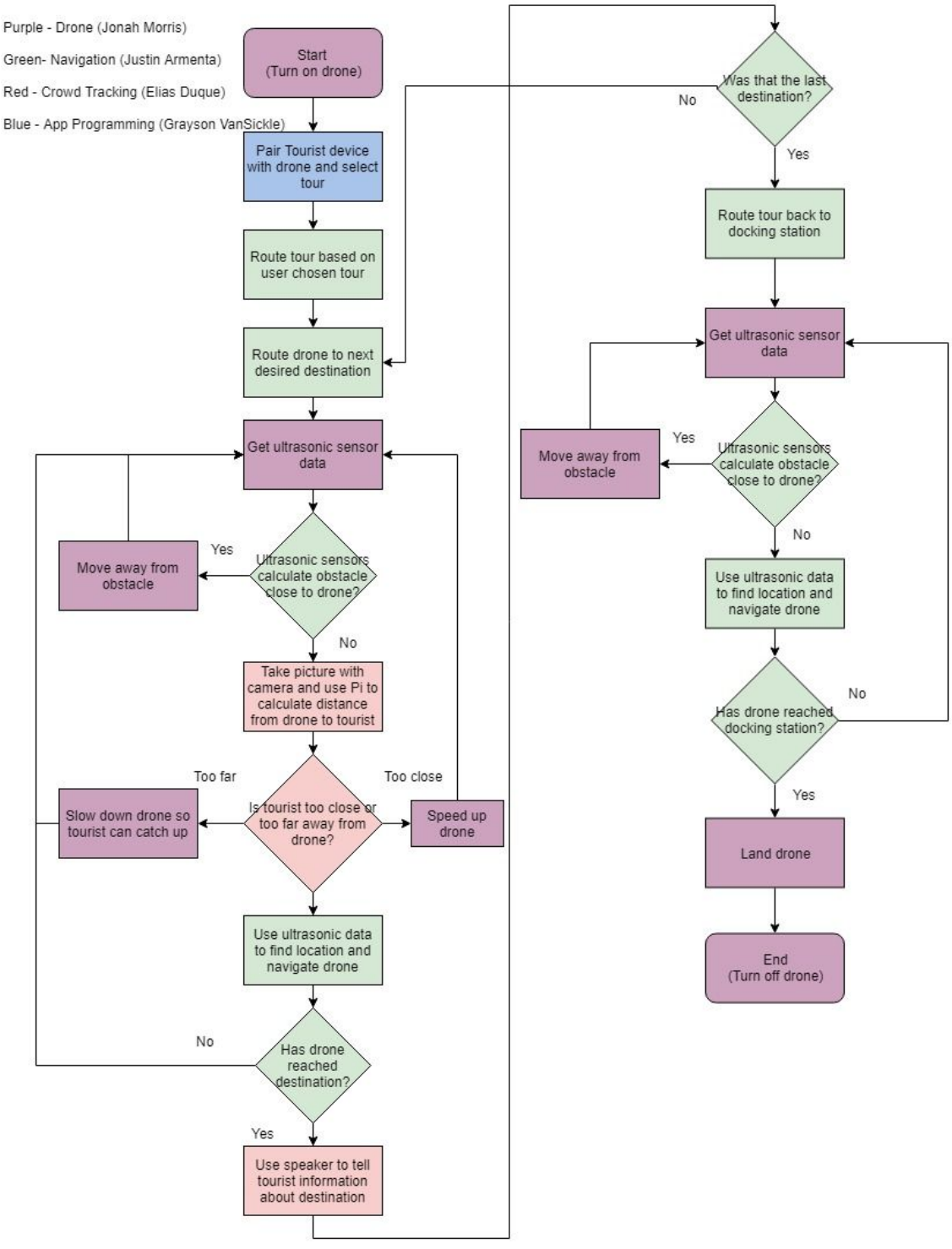


Figure 1: Drone Logic Overview

Figure 1 shown above displays the logic that the drone will follow throughout the process from the beginning of the tour to the end. Every function block is color coded based on who will be in charge of that particular function. As shown above Jonah (Purple) will be in charge of the placement and function of the components on the drone, Grayson (Blue) will be in charge of creating an app for the WiFi connection between the drone and the tourist, Elias (Red) will be in charge of making sure the drone stays within a certain distance of the tourists, and Justin (Green) will be in charge of making sure the drone knows how to get from one destination to the next.

2.3. Referenced Documents and Standards

https://federaldroneregistration.com/?gclid=EAIaIQobChMIq9mDtIjN3QIVIIlpCh0-KAneEAAYASAAEgLrHPD_BwE
https://federaldroneregistration.com/know-before-you-fly
http://knowbeforeyoufly.org/
https://www.scienceabc.com/innovation/what-is-the-range-of-bluetooth-and-how-can-it-be-extended.html
https://zachry.tamu.edu/tours/
https://zachry.tamu.edu/

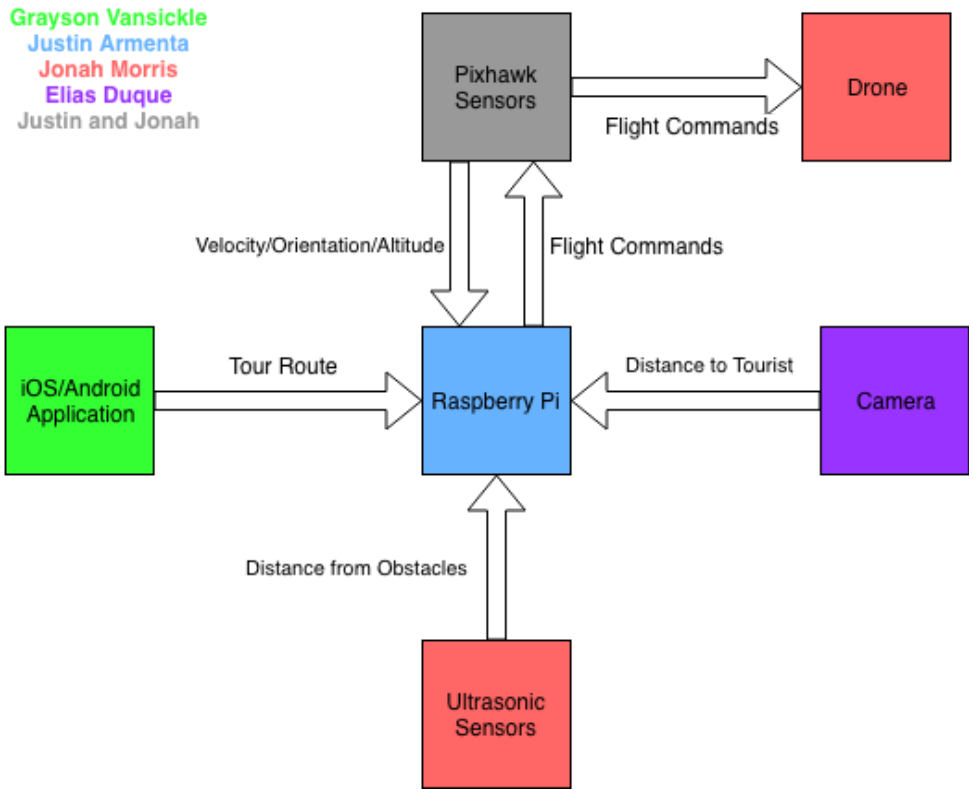


Figure 2: Overall System Diagram

3. Operating Concept

3.1. Scope

The Zach Drone is intended to fly inside Zachry where the drone will use ultrasonic sensors on the drone to navigate. The drone assumes that the tourists will not try to hit the drone or alter it's path while it is flying. It will be controlled using a microcontroller that can be connected to phones through WiFi using an app interface.

3.2. Operational Description and Constraints

The Zach Drone will be used by small or large groups to tour Zachry. The drone will start from the Anadarko Welcome Center where the tourists will connect to the drone with their phone through the phone app. The user will then pick what type of tour they would like to take. The drone will use the ultrasonic sensors on the drone and compare those values to the known dimensions of the building to navigate through the tour while using a speaker to playback important information about the building. A microcontroller will monitor sensors, use the camera, communicate with the flight controller and take input from the phone application.

The constraints for the Zach Drone are as follows:

- The drone must start at the Anadarko Welcome Center.
- Tourist must have a smart-phone with the app downloaded.
- The drone must not be intentionally hit by the user or others in the building.
- The tour can only last up to twenty minutes before needing to recharge the drone.
- The speed of the drone cannot exceed that of walking speeds.
- Zach Drone will not be functional in an overcrowded building because camera may not be able to detect the group leader
- The drone must be given two hours in between tours to recharge the battery.
- Must take open stairs to travel between floors and cannot pass through doorways.

3.3. System Description

The Zach Drone is made up of four main subsystems detailed below:

Tourist Tracking & Speaker System: This system is used to track the tourists to make sure that they do not fall too far behind while taking the tour and to give the commentary about the various points of interest in Zachry. The system will consist of a camera to analyze and track a crowd leader to gauge the distance between the crowd and the drone. Additionally this system includes a speaker to give the commentary. The tracking system will communicate with the microcontroller to determine if the drone should slow down to allow the tourists to catch up or speed up if they get too close.

Drone & Sensors: This subsystem consists of the drone itself and the sensors mounted on it. Sensors will be used to avoid collisions with people and objects inside Zachry. It will accomplish this by having the microcontroller telling the flight controller to move in the opposite direction of whichever ultrasonic sensor is outputting a close reading.

This system tells the microcontroller how close the drone is to objects on the left, right and front of the drone. The microcontroller will determine if the drone will need to change paths and then communicate with the flight controller on the drone to tell it which direction to fly.

Zach Drone App: The drone will use a phone application to communicate with the tourists. This subsystem includes the user interface as well as the ability to talk to the microcontroller through a WiFi connection. The app will contain a connect-to-drone feature as well as options for what type of tour that the user wants to take. Currently, only the art tour is functional but other tours can easily be added. The user will also be able to stop or pause the tour from the app.

Tour Navigation: The Zach Drone will use ultrasonic sensors and the dimensions of the inside of Zachry to determine where the drone is at any given time. This system will receive signals from the ultrasonic sensors and then determine where the drone is within the building and in which direction the drone should fly based on this location.

3.4. Modes of Operations

There are three modes of operation for the drone. The first mode is the tour mode. During this mode, the drone will fly from destination to destination while guiding the tourist. The second mode of operation is the wait mode. The drone will enter this mode if the users are too far away for too long, the user presses the wait button on the app, or if connection is lost with the phone. In this mode, the drone will land at its current position and wait for the condition to start again to be satisfied. The conditions for the drone to exit the wait mode and enter back into tour mode will be: the customer's device has connection and is in range of the drone after losing connection or that the start button has been clicked again to resume the tour after a user-initiated pause. The drone will only wait until the maximum length of the tour is reached. Once the maximum tour length of twenty minutes is reached, the drone has successfully completed the tour, or the user cancels the tour the drone will enter the third mode of operation which is the return to start mode. In this mode the drone will navigate itself back to the Anadarko Welcome Center where it will be picked up to charge.

3.5. Users

The Zach Drone can be used by anyone who would like to take a tour of the new Zachry building without having to follow the tour schedule. The drone provides more flexibility to those who are taking a short visit to the Texas A&M campus. Anyone with a smartphone will be able to download the app and take a tour.

3.6. Support

Commonly asked questions will be listed in the app. Although, this functionality has not been implemented at this time.

4. Scenario(s)

4.1. Small Personalized Groups

For tours with a small group of guests, the Zach Drone tour would be an ideal option. This would allow tourists to choose a personalized tour that meets their needs, whether it be looking at and listening about all the different art installations around Zachry or seeing what different learning and advising opportunities are available around the building. The drone will be able to easily guide small groups by having them follow behind the drone at whatever pace they would like since the drone will wait on you.

4.2. Large School Groups

For large groups such as a school field trip the Zach Drone would be a great option. Since all large groups on school field trips or general tours of Texas A&M will have group leaders such as teachers or actual guides it will be very easy to control the pace of the group by making the drone track the leader. That way the leader of the group can control at what pace the drone gives the tour. This will also add a bonus incentive for people to stay with the group because of the interest that the drone tour will bring. Instead of having a boring tour guide that many people will ignore, the drone could be as big of an attention getter as the building itself.

5. Analysis

5.1. Summary of Proposed Improvements

The use of the Zach Drone will allow tourists to have a more immersive experience while visiting the new Zachry Engineering Education Complex. The Zach Drone will give tourists the opportunity to custom fit a tour to whatever they preference using the Zach Drone App's multiple tour feature. Using the Zach Drone will allow tourists to tour Zachry whenever they feel instead of having to wait for a tour guide. The Zach Drone will show the tourists some of the innovative ways Texas A&M University is utilizing the new building.

5.2. Disadvantages and Limitations

The Zach Drone will be limited to the time of the tours since it will be running on a battery. The tour given by the Zach Drone will only be able to be a maximum of twenty minutes since that is the longest the battery can run before needing to be charged. The Zach Drone App is the main interface to communicate with the Zach Drone, and without a smartphone the tourists won't be able to select a tour.

5.3. Alternatives

- Camera vs. Bluetooth for crowd tracking
    - Using Bluetooth would be more unreliable due to the nature of Bluetooth connections.
    - Using Bluetooth signals would also lead to greater inaccuracies when calculating the distance between the drone and the tourists

    - Using Bluetooth signals would take less processing power than image processing.
- Ultrasonics vs ARUBA beacons for navigation
    - ARUBA signals would produce fuzzy data when used that is only up to 3m of accuracy which would cause the location of the drone to be incorrect.
    - Ultrasonics would only have a small degree of inaccuracy but could be difficult to use if the drone gets lost.
    - The ultrasonics would need the dimensions of all locations near the tour route to be used.

6. Impact(s)

6.1. Economic & Time Impacts

Electricity is the only upkeep cost the drone will have; more specifically the cost of the amount of electricity it takes to recharge the drone battery. Charging a battery for an hour or two has a negligible cost.

Currently though the true cost of the Zachry tours is time. Tour givers are departmental members that volunteer to have time taken out of their day to give these tours. A drone tour would return that time back to them; allowing them to be put to better use. In other settings this can save a company the cost of an entire position.

6.2. Social & Business Impacts

In the case of the Zachry building the Zach Drone will act as an engineering feat to entice potential students. Its secondary purpose is to “wow” students & show them what engineering is capable of. For other business settings it will allow companies to give tours where they otherwise could not spare time or money to have someone else do it. It could even be used in an accessibility sense if we were to switch the speaker out for a screen or set of LEDs, the drone could be used to direct deaf people. Applicable settings could be museums, airports, & many others.

6.3. Safety Impacts

The propellers on a drone can spin at thousands of RPM & can pose a hazard. The drone will be set to fly as high as it can leaving enough room to not hinder its performance. Sensors will detect people & the drone will attempt to maneuver away should someone attempt to touch it by flying in the opposite direction of whichever ultrasonic had a reading that was too close in proximity.

7. Citations

https://federaldroneregistration.com/?gclid=EAIaIQobChMIq9mDtIjN3QIVIIlpCh0-KAneEAAYASAAEgLrHPD_BwE
https://federaldroneregistration.com/know-before-you-fly
http://knowbeforeyoufly.org/
https://www.scienceabc.com/innovation/what-is-the-range-of-bluetooth-and-how-can-it-be-extended.html
https://zachry.tamu.edu/tours/
https://zachry.tamu.edu/

Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Functional System Requirements

REVISION – 2.0
28 April 2019

FUNCTIONAL SYSTEM REQUIREMENTS
FOR
Zach Drone

PREPARED BY:
TEAM <49>

_____
Author Date

APPROVED BY:

_____
Project Leader Date

_____
Prof. S. Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/24/18	Elias Duque		Added Introduction section
0.4	10/1/18	Grayson Vansickle		Added Applicable and Reference Documents section
0.6	10/1/18	Jonah Morris		Added System Definition subsection of Requirements section
0.8	10/1/18	Justin Armenta		Added Characteristics subsection of Requirements section
1.0	10/3/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information
2.0	4/28/19	Grayson Vansickle		Formatted
2.2	5/1/19	Jonah Morris		Updated Information

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Purpose and Scope

The purpose of the Zach Drone is to give guided tours to a crowd of people that have signed up for a viewing of the new Zachry Engineering building. It shall communicate with an app that will sync up with the phones of the tourists so that it shall keep track of them. The drone must maneuver the hallways of Zachry and must give a spoken tour. It should give information on where labs and classrooms are, as well as draw the crowd's attention to the installed art pieces. The drone shall detect via a camera that will track the crowd leader's location. When the tour is over the drone must return to the Anadarko Welcome Center so that it shall recharge before the next tour.

The drone must be able to fly using its own power for at least 15 minutes to give a fully-fledged tour. It must have a speaker to give commentary on the surroundings and shall provide relevant information. The ultrasonic sensors on the drone shall be used along with known dimensions of the building to find the drone's location and plot its course. The drone shall have ultrasonic sensors to prevent collisions with ceiling fixtures, walls, and pedestrians.

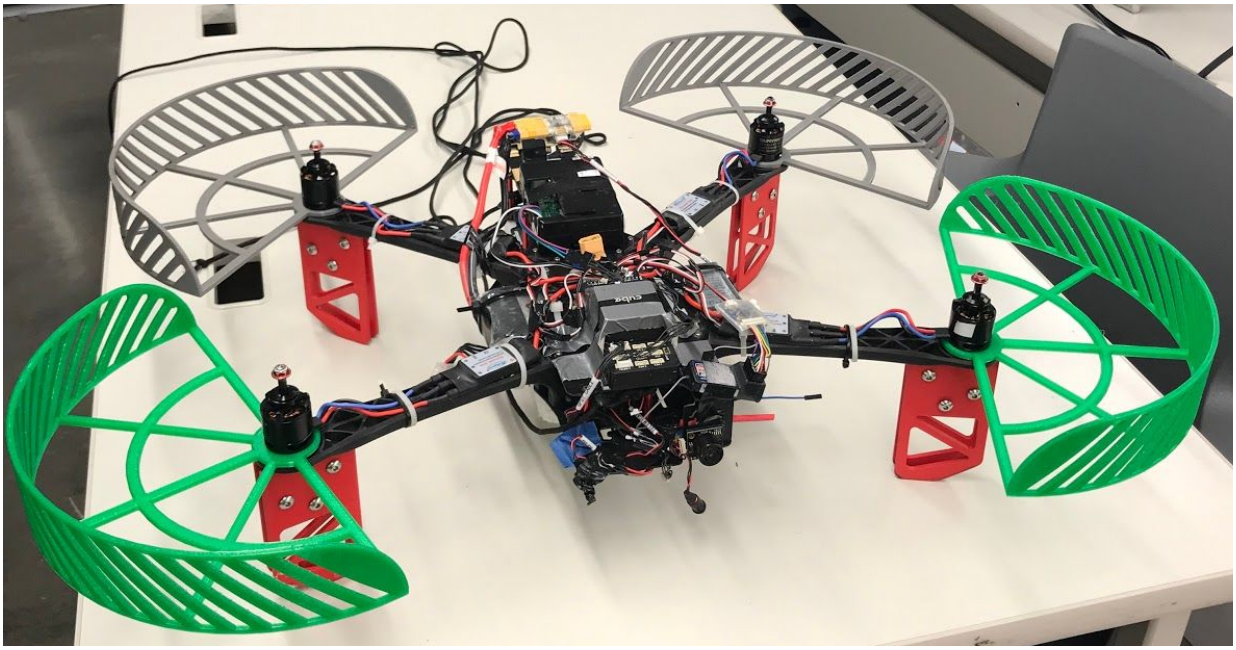


Figure 1 - Final Image of Drone

1.2. Responsibility and Change Authority

Jonah Morris is the team leader & shall be responsible for the oversight of the subsystems & ensuring that requirements are met according to the validation plan. Any changes to be made to the deliverable requirements must be approved by both Jonah Morris & Stavros Kalafatis.

Subsystem	Responsibility
Drone & Sensors	Jonah Morris
Navigation	Justin Armenta
Crowd Tracking	Elias Duque
Mobile Application	Grayson Vansickle

Table 1 - Subsystem Responsibility

2. Applicable and Reference Documents

2.1. Applicable Documents

The following documents, of the exact issue and revision shown, form a part of this specification to the extent specified herein:

Document Number	Revision/Release Date	Document Title
NFPA 70	2014	National Electrical Code
IEEE 802.15.4	2013	Standard for Low Rate WPAN
FAA Part 107	2018	Small Unmanned Aircraft Regulations

Table 2 - Applicable Documents

2.2. Reference Documents

The following documents are reference documents utilized in the development of this specification. These documents do not form a part of this specification, and are not controlled by their reference herein.

Document Title	Publisher
Raspberry Pi 3 B Data Sheet	Raspberry Pi
Pixhawk 2.1 Flight Controller Data Sheet	Pixhawk
Ultrasonic Rangefinder HRLV-EZ0 Data Sheet	Maxbotix
Python Standard Library Reference	Python
Swift Language Reference	Apple
Java SE Technical Documentation	Oracle

Table 3 - Reference Documents

2.3. Order of Precedence

In the event of a conflict between the text of this specification and an applicable document cited herein, the text of this specification takes precedence without any exceptions.

All specifications, standards, exhibits, drawings or other documents that are invoked as “applicable” in this specification are incorporated as cited. All documents that are referred to within an applicable document are considered to be for guidance and information only, with the exception of ICDs that have their applicable documents considered to be incorporated as cited.

3. Requirements

The drone will give guided tours to tourists who have set up a tour of Zachry. The drone must be able to connect with the tourist's phone using a WiFi connection from an app, so that the user can start, stop and pause the tour. The drone must be able to navigate through Zachry from one checkpoint to the next without colliding with objects.

This section defines the minimum requirements that the development item(s) must meet. The requirements and constraints that apply to performance, design, interoperability, reliability, etc., of the system are covered.

3.1. System Definition

The Zach Drone is an automated tour delivery system. Users connect to the drone using WiFi through a mobile app interface. The drone will fly a predefined route while using a speaker to playback important information about the building. For this project there are four major subsystems: mobile app, navigation, tour group tracking and the drone.

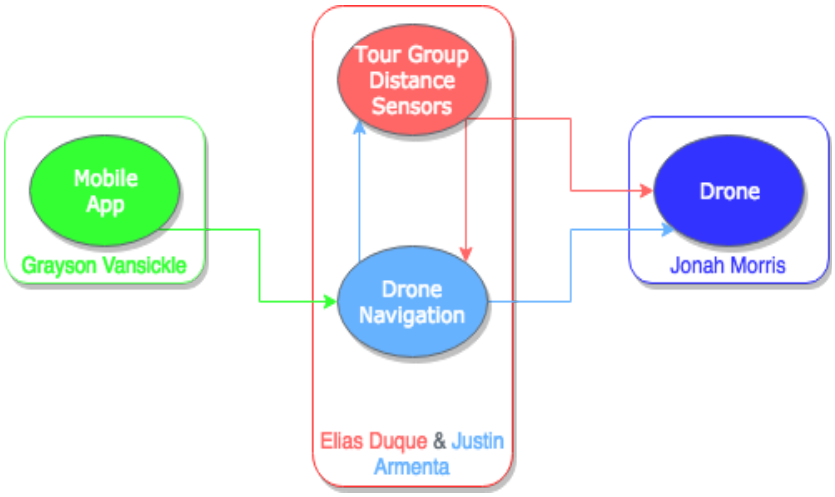


Figure 2 - Drone Subsystems & Responsibilities Block Diagram

The mobile application is the primary interface with the user. Once the user starts the tour, the mobile app will communicate with the navigation system and tell it where to go. The navigation system will use ultrasonic sensors on the drone to determine where it is, if there are obstacles, and what direction the drone should go next. The navigation system will then tell the drone's flight controller which direction to fly in. The drone will fly towards the waypoint while the group tracking system controls the speed of the drone based on the distance from the group.

3.2. Characteristics

3.2.1. Functional / Performance Requirements

3.2.1.1. Smartphone Application

An application shall be created on both Android and Apple platforms where the tourist can use their smartphone to access the app, and use it to select a tour route and when they would like a tour. The drone shall also have a speaker to tell the tourists information about Zachry. The app will continuously communicate with the drone allowing the user to change route or cancel.

3.2.1.2. Drone Functionality

The drone shall be able to fly for the full length of a 15 minute tour, while being able to avoid a collision with anything that comes near the drone. It will also be able to take off and land at the designated location inside of the Anadarko Welcome Center. The maximum time requirement of the tour is necessary since tours need to cover as much of the building as possible while also taking into account that our battery life is about 20 minutes. There also must be a designated take-off and landing location so people know where to go for the tour and a storage spot for the drone.

3.2.1.3. Navigation of Checkpoints

The drone shall be able to navigate, in an efficient manner, from any of the tour destinations to another successfully using the ultrasonic sensors on the drone as a guidance system. Without this capability the drone wouldn't be able to lead a tour group to all the spots that the group would like to see.

3.2.1.4. Crowd Guiding

The drone shall stay within about 16 feet of the crowd using a camera to keep track of a crowd leader and their distance. 16 feet is recommended for the most reliable operation of the tracker. If the crowd becomes closer than 9 feet the drone will speed up. If the crowd exceeds 23 feet away then the drone will slow down. If the drone cannot locate the crowd for longer than 15 seconds then the tour will end assuming the crowd has left or that there is a hardware malfunction.

3.2.2. Physical Characteristics

3.2.2.1. Mass

The maximum total mass of the drone shall be less than 3.06 kg. The maximum combined thrust for four motors produces a total of 4.6 kg. To efficiently take off the motors must be able to produce thrust equal to 150% of the weight of the drone which here is 4.6 kg. 4.6 kg is 150% of 3.06 kg making it the maximum weight we should target.

Rationale: This is the maximum weight that can be carried by the drone such that it will fly at 75% throttle. 75% throttle simply means the motor can run at a higher thrust thanks to the stable conditions inside the building.

3.2.2.2. Volume Envelope

The volume envelope of the Zach Drone shall be less than or equal to 150 mm in height, 525.4 mm in width, & 525.4 mm in length.

Rationale: This is the smallest size that can house all components safely. Drone size affects weight so a smaller drone makes meeting our weight requirement easier.

3.2.3. Electrical Characteristics

3.2.3.1. Inputs

No combination of settings input by the user shall damage the Zach Drone, cause any malfunction within the system, nor reduce performance.

Rationale: The chance of user errors should be minimized by the design of the system.

3.2.3.1.1 Power Consumption

The maximum peak power for the system shall not exceed 1050 watts.

Rationale: The maximum power the drone motors can take is 259 watts per motor. The rest of the system requires less than 10 watts to power.

3.2.3.1.2 Voltage Level

The output voltage of the battery will be between 14.8 and 16.5 VDC. The minimum input voltage levels for the sensors, motors, microcontroller and Pixhawk flight controller are 2.8 VDC, 4.5 VDC, 4.75 VDC and 4 VDC respectively.

Rationale: The nominal voltage levels for the sensors, motors, microcontroller and Pixhawk are 3.3 VDC, 5 VDC, 5 VDC and 5 VDC respectively.

3.2.3.1.3 External Commands

The Zach Drone shall document all external commands in the appropriate ICD.

Rationale: The ICD will capture all interface details from the low level electrical to the high level packet format.

3.2.3.1.4 Data Output

The Zach Drone shall include a speaker for playback to the tourist.

Rationale: This requirement was set by the customer.

3.2.3.1.5 Diagnostic Output

The Zach Drone shall include a mobile application interface for control and data logging.

Rationale: Provides the ability to manually control things for debugging.

3.2.3.2. Wiring

The Zach Drone shall follow the connector guidelines set forth in the National Electric Code. Article NEC 300 details general guidelines for wiring methods and materials.

Rationale: Conform to general wiring standards.

3.2.4. Communication Requirements

3.2.4.1. Phone App to Microcontroller Connection

The tourist's phone shall use the app to tell the drone which tour to take, so the drone can navigate to all checkpoints one at a time. The app will also have an option to "Cancel Tour" during the tour if the user becomes uninterested in the tour or has other obligations and has to leave. There will also be a "Pause" option, in case the user wants to look at something for a while or go to the bathroom, where the drone will hover idly until the user clicks the "Resume Tour" option or enough time has passed that the drone will automatically cancel the tour.

3.2.4.2. Microcontroller to Audio Connection

The microcontroller will use a 3.5 mm audio jack connection to send an audio signal to the speaker on the drone to tell the tourists information about destinations on the tour.

3.2.4.3. Microcontroller to Flight Controller Connection

The microcontroller will use a serial connection to connect with the flight controller to command the flight controller where to navigate to, how fast, when to avoid an obstacle, and other important flight commands.

3.2.4.4. Ultrasonic Sensors to Microcontroller Connection

The ultrasonic sensors will be connected to one of the GPIO ports on the Raspberry Pi. These sensors will be used to navigate around Zachry as well as detect if the drone is close to colliding with something as well as for navigation.

3.2.5. Environmental Requirements

The Zach Drone shall be designed to withstand and operate inside of the Zachry Engineering Education Complex.

Rationale: This is a requirement specified by our customer because this is where tours will be held.

3.2.5.1. Thermal Requirement

The Raspberry Pi 3 B on the Zach Drone will not exceed 70℃.

Rationale: The Raspberry Pi 3 B is rated for temperatures of 0℃ - 70℃ by the manufacturer.

3.2.6. Failure Propagation

The Zach Drone shall account for connection failures as well as objects getting near the drone. The drone will have a manual override to land if the user observes a malfunction.

3.2.6.1. Camera Tracking

3.2.6.1.1 Camera Failure

If the camera used to track the crowd should malfunction to where the image can no longer be processed then the camera will shut off and the crowd detection functionality will be disabled. The drone will stop and land at its current location.

3.2.6.1.2 Image Processing Failure

If the image processing detection and tracking should fail such that the leader can no longer be tracked then the drone will attempt to re-initialize the crowd leader. Should this attempt fail then the drone will stop and land at its current location.

3.2.6.2. Object Proximity Detection

3.2.6.2.1 Ultrasonic Proximity Detection

If an object gets too close to the drone, the ultrasonic sensors, which will be placed on the left, right and front sides of the drone, will be used to detect how close the object is to the drone. The microcontroller will then tell the drone to move away from the object by going in the opposite direction of the ultrasonic that was triggered. If there are objects on the left and right sides of the drone, the drone will land.

3.2.6.3. Navigation

3.2.6.3.1 Sensor Reading Errors

If there is a small one time failure of an ultrasonic sonic, the error will most likely be ignored and not taken into account due to how the algorithm for navigating the drone works, which expects that the sensors won't be perfect and may produce a false, one time, value. Values from the ultrasonics will be sent through an error detection and correction algorithm. This algorithm ensures that incorrect values from the ultrasonics will not affect the navigation of the drone. 0

4. Support Requirements

The Zach Drone requires that the battery be charged for two hours in between uses. The Zach Drone also requires the user to have a smartphone with the Zach Drone app downloaded.

Appendix A Acronyms and Abbreviations

Zach	Zachry Engineering Education Complex
USB	Universal Serial Bus
℃	Degrees Celsius
GPIO	General Purpose Input Output
APP	Application
NEC	National Electric Code
VDC	Volts Direct Current
KG	Kilogram
%	Percentage

Appendix B Definition of Terms

Flight Controller: A microcontroller that includes gps, compass and gyroscope that controls the motors to fly the drone.

Ultrasonic Sensor: A distance sensor that uses ultrasonic sound waves to determine distance to the nearest object.

Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Interface Control Document

REVISION – 2.0
28 April 2019

INTERFACE CONTROL DOCUMENT
FOR
Zach Drone

PREPARED BY:
TEAM <49>

_____
Author Date

APPROVED BY:

_____
Project Leader Date

_____
Prof. Stavros Kalafatis Date

_____
T/A Date

Change Record

Rev.	Date	Originator	Approvals	Description
0.0	9/6/18	Grayson Vansickle		Draft Release
0.2	9/23/18	Justin Armenta		Added Overview section
0.4	10/1/18	Grayson Vansickle		Added Communications Protocol section
0.6	10/1/18	Elias Duque		Added Physical Interface section
0.8	10/3/18	Jonah Morris		Added Electrical Interface section
1.0	10/4/18	Grayson Vansickle		Edited and Formatted
1.2	12/5/18	Grayson Vansickle		Updated information
2.0	4/28/19	Grayson Vansickle		Formatted
2.2	5/1/19	Jonah Morris		Updated information

Table of Contents

List of Tables

List of Figures

1. Overview

This document will cover the systems designed and used in the Zach Drone tour system. The drone that will be used to give tours of Zachry will be described on a functionally detailed level. The interface between the flight controller on the drone, the microprocessor, and all devices used to determine location will be described in terms of how they will function separately as well as how they will communicate with each other. The way the proximity sensors will interact with the microprocessor and flight controller will be addressed as well. The physical aspects of the drone such as its weight, dimensions, size, component location, and component size will also be described. Electrical aspects will be explained as well. This will include details such as power inputs and outputs for components, voltage, current and power levels at different points in the system, and signal interfaces for components. Lastly, the methods in which the user will interact with the drone will be explained.

2. References and Definitions

2.1. References

Document Title	**Revision/Release Date**	**Publisher**
National Electrical Code	2014	NECA
Raspberry Pi 3 B Data Sheet	2018	Raspberry Pi
Pixhawk 2.1 Data Sheet	2016	Pixhawk
HRLV-EZO Data Sheet	2015	Maxbotix
SimonK 30A ESC Data Sheet	Revision 1.0	Lynxmotion

Table 1 - References

2.2. Definitions

g.	Grams
kg.	Kilograms
m.	Meters
mm.	Millimeters
V.	Volts
A.	Amps
mA.	Milliamps
Ah.	Amp hour
mAh.	Milliampere hour
W.	Watt
ESC	Electronic Speed Controller

3. Physical Interface

3.1. Weight

Component	Weight	Amount	Total Weight
SunnySky X2212-13 KV980 II	69 g	4	276 g
1045 Propeller	7 g	4	28 g
SimonK 30A ESC	27 g	4	108 g
Matek PDB-XT60	11 g	1	11 g
Pixhawk 2.1 Flight Controller	50.5 g	1	50.5 g
Maxbotix Ultrasonic Rangefinders	4.23 g	3	12.69 g
Raspberry Pi 3 B	49.7 g	1	49.7 g
TATTU 4S 8 Ah 25C Li-Po Battery	785 g	1	785 g
Drone Frame	90 g	1	90 g
PX4FLOW	30 g	1	30 g

Table 2 - Physical Weight

3.2. Dimensions

Component	Length	Width	Height
SunnySky X2212-13 KV980 II	28 mm	28 mm	40 mm
1045 Propeller	254 mm	114.3 mm	6 mm
SimonK 30A ESC	216 mm	25 mm	8 mm
Matek PDB-XT60	50 mm	36 mm	24.2 mm
Pixhawk 2.1 Flight Controller	94.5 mm	44.3 mm	31.42 mm
Maxbotix Ultrasonic Rangefinders	22.36 mm	15.52 mm	20 mm
Raspberry Pi 3 B	87 mm	58.5 mm	18 mm
TATTU 4S 8 Ah 25C Li-Po Battery	165 mm	65 mm	53 mm
Drone Frame	500 mm	500 mm	150 mm
PX4FLOW	45.5 mm	35 mm	20 mm

Table 3 - Physical Dimensions

3.3. Mounting Locations

3.3.1. Mounting Location for Control Electronics

Electronics responsible for the control & flight of the drone include the flight controller & the Raspberry Pi which will manage the navigation. For optimum performance & accuracy the flight controller must be placed in the direct center of the drone's mass. The Raspberry Pi will be placed behind the flight controller at the tail end of the drone.

3.3.2. Mounting Location for Power Electronics

Electronics responsible for the power input, output, & distribution include the power distributor & the battery itself. The battery will be mounted in the lower half of the middle of the drone's mass to keep it balanced. The power distributor must be in the middle of the drone below the flight controller to allow for the brushless motor controls to reach evenly.

3.3.3. Mounting Location for Motors

The motors must be placed at the furthest end of the drone arms. Propeller blades are attached to the shafts of the motors. The brushless motor controls will be mounted along the arms themselves stretching from the motors to the power distributor in the middle.

3.3.4. Mounting Location for Sensors

Ultrasonic sensors must be placed on the three vital faces of the drone for maximum precision in collision avoidance. There will be one sensor per side: front, left, right.

3.3.5. Mounting Location for PX4FLOW

The PX4FLOW will be mounted to the bottom of the drone to allow direct view of the ground.

3.3.6. Mounting Location for Camera

The camera will be mounted on the rear of the drone using popsicle sticks to maintain the angle required for proper field of view.

4. Electrical Interface

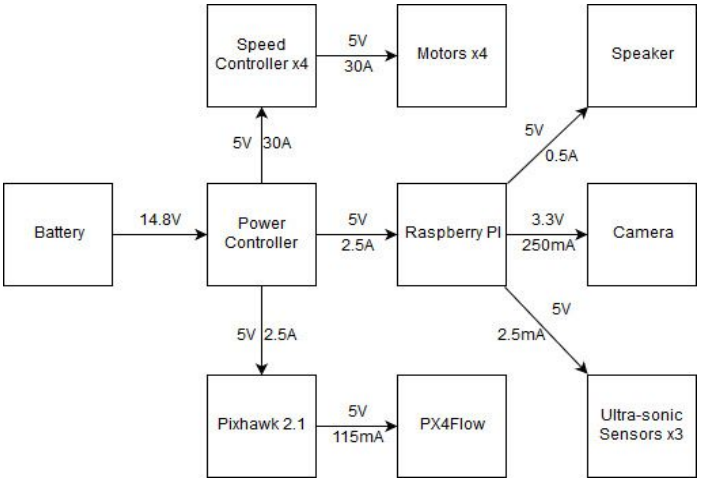


Figure 1: Electrical Interface diagram

4.1. Primary Input Power

The primary input power is the 14.8V 8000mAh battery pack that will be connected to the power distributor.

4.2. Voltage and Current Levels

Component	Voltage [V]	Current [A]	Power [W]
Pixhawk 2.1	5	2.5	12.5
Raspberry Pi	5	2.5	12.5
Speed Controller	5	2	10
Motor ESC	0-5	0-30	150 (max)
Speaker	5	0.5	2.5
Ultrasonic sensor	3.3	.0025	0.00825
PX4FLOW	5	0.125	0.625

Table 4 - Electrical Voltage, Current, & Power

Detailed in Table 4 above are the wattages consumed by each part of the drone in one hour. Taking the amount of components into account it is not unreasonable to see a total of 215 watt hours. It is unreasonable & unnecessary that our drone will fly for an hour; our target is 20 minutes. In a 20 minute window our drone will therefore use a third of the the total watt hour amount, equaling 71.7 watts. Every component is vital to normal operation of the drone except for the speaker. There is no lower power operation mode of the drone though the speaker only consumes power when in use during tours.

4.3. Signal Interfaces

The signal interfaces included are as follows:

- Serial port connection from Raspberry Pi to Pixhawk
- Pixhawk PWM signal outputs to speed controllers
- Raspberry Pi 3.5mm audio jack to speaker
- Ultrasonic sensors to Raspberry Pi GPIO headers
- USB serial between Pixhawk and PX4FLOW

4.4. User Control Interface

The user control interface is a mobile application. The user is given options for stopping, starting, and pausing the tour. The application communicates these options to the drone's Raspberry Pi via WiFi connection.

5. Communications / Device Interface Protocols

5.1. Audio Interface

The drone will have an attached USB 2.0 speaker connected to the 4-pole stereo output port on the Raspberry Pi 3 B. The speaker will be powered by the one of the Pi's four onboard USB 2.0 ports. This audio interface will be used to give the description of the point of interest the drone is currently located at.

5.2. Device Peripheral Interface

The ultrasonic sensors used by the drone will be attached to the Pi's extended 40-pin GPIO header. The Pixhawk flight controller will also be connected to the GPIO header. These ports will be used to communicate between the sensors and microcontroller.

The PX4FLOW will be connected the Pixhawk using a usb serial connection at 921600 baud.

5.3. WiFi Interface

The Raspberry Pi 3 B has onboard BCM43438 wireless LAN (WiFi). Using the Pi's WiFi connection, the Pi will be able to communication with the mobile application. The WiFi connection will comply with the IEEE 802.11 standards for wireless local area networks (WLANs)

6. References

Ultrasonic Sensors
https://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf

Raspberry Pi
https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md

Pixhawk Flight Controller
http://www.hex.aero/wp-content/uploads/2016/07/DRS_Pixhawk-2-17th-march-2016.pdf

SimonK Speed Controller
http://www.lynxmotion.com/images/document/PDF/Lynxmotion%20-%20SimonK%20ESC%20-%20User%20Guide.pdf

Zach Drone

Elias Duque

Tour Group Tracker Subsystem

REVISION – 2.0
28 April 2019

Table of Contents

List of Figures

List of Tables

1. Introduction

The purpose of this report is to detail the working aspects and the validation of the Tour Group Tracking subsystem of the Zachry Tour Drone. The Tour Group Tracking is primarily software based with the only hardware being the necessary camera sensor and the microcontroller. This subsystem interacts the most with the navigation subsystem as the data from this subsystem will generate a multiprocessing variable for the navigation subsystem to interpret as a speed up or a slow down. The Tour Group Tracking subsystem augments the tour by making it more comfortable for tourists allowing them to go at their own pace and to ensure that audio commentary is not delivered before the crowd is around to hear it. This report will cover aspects such as the design, data collection, accomplishments, and admissions of necessary future work.

2. Theory

The scope of the Tour Group Tracking subsystem is to create an appropriate response to the changing distance between the tour drone and the crowd of tourists. This response is necessary due to the variable pace that a crowd of people can move. As the drone reaches a point of interest it will begin to give commentary and without the appropriate responses may be giving the commentary to a crowd that has yet to catch up. This system monitors an appointed crowd leader and tracks them throughout the tour and uses them to gauge the distance of the overall crowd. OpenCV's histogram of ordered gradients human descriptor detector and MOSSE tracker work in tandem to track the crowd leader and estimate their distance. Based on this the drone may generate a response for the navigation subsystem to initiate a speed change. Should the crowd leader exceed 23 feet from the drone it will send a signal to navigation for a slower throttle. Should the crowd leader come closer than 10 feet then the microcontroller will send a signal to navigation to up the throttle. Speed changes are incremental such that if a speed change puts the crowd leader in the acceptable area, no more speed changes are generated.

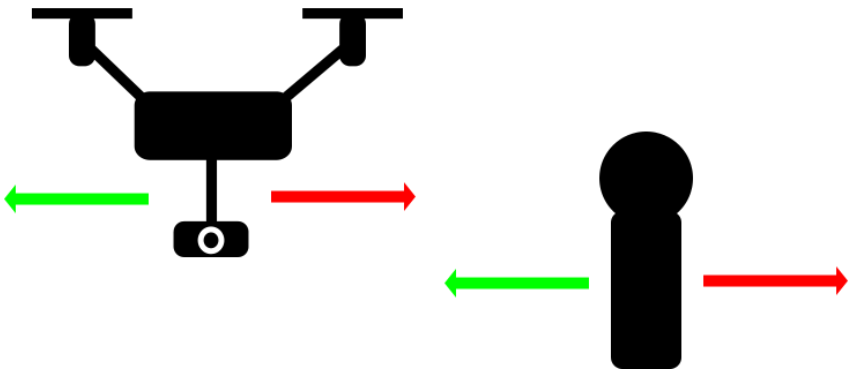


Figure 1: Conceptual Overview of System Response

3. Design

3.1. Hardware

The only hardware unique to the Tour Group Tracker subsystem would be the Raspberry Pi Camera Module v2. The other hardware that could be considered part of this system would be the Raspberry Pi microcontroller, Pixhawk flight controller, and the drone itself with all its components. The main interaction that this subsystem has with those other components is through the navigation subsystem via a signal that will command the drone to change speed. Otherwise this subsystem is mostly divorced from the other hardware and could function even without a drone. The camera hardware consists of the the camera module itself and the mount forged from popsicle sticks. The Raspberry Pi Camera Module v2 uses an extended ribbon cable connected to the camera port on the microcontroller and runs off 3.3 V and 250 mA of current. Exact data is difficult to find but with the lower resolutions the camera is running at these numbers are subject to change. The camera is capable of high resolution but the frames are downsampled to 480 x 360 pixels of resolution for processing speed. The camera sensor captures data at 30 frames per second but only analyzes frames as it is ready to accept them. Due to the camera being manufactured specifically for this microcontroller with the microcontroller having a specialized port just for it no other hardware is necessary for the operation of the camera besides obvious necessities such as a power source. The mount was built out of popsicle sticks due to them being cheap, easy to cut, and strong enough to prevent damage to the camera. The mount is merely meant to hold the camera in a position where it can see the crowd from the hovering height of the drone.

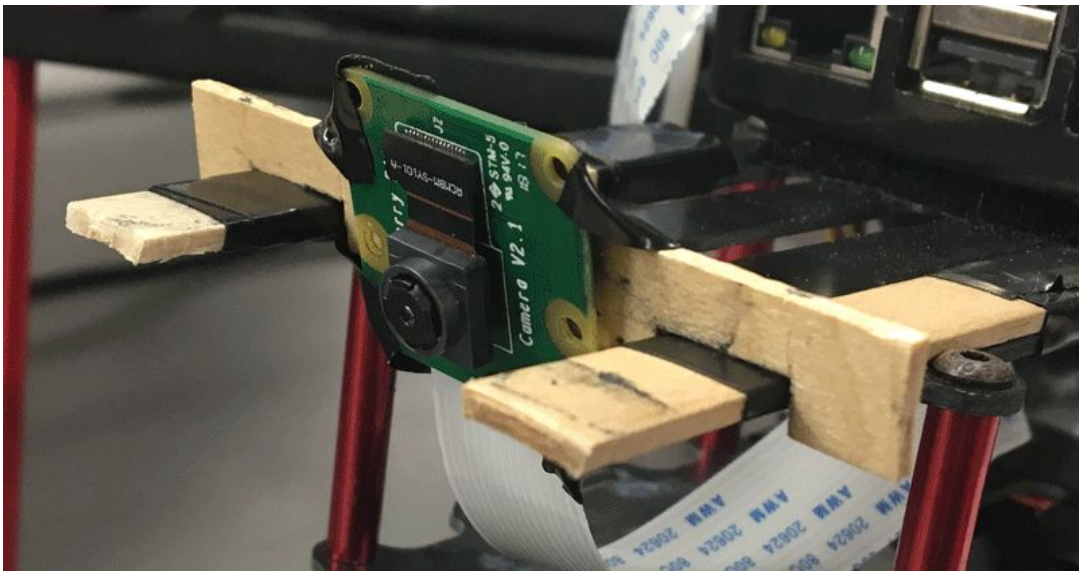


Figure 2: Camera Module attached to popsicle stick mount

3.2. Software

3.2.1. main.py, process 4

Since the Tour Group Tracker subsystem relies heavily on image processing to perform its duties it is very processor intensive. The programming language Python was used for the coding and OpenCV 4.0.0 was used for all steps of image processing while imutils was used for downsampling, cropping, and video-streaming. The actual image processing requires calling one or two function but the extensiveness of the program comes from the logic required to update the tracker and preserve what we are tracking. The tour group tracker process does not initiate beyond initialization unless the multiprocessing variables “connected.value” equals 1, corresponding to the app being actively connected to the drone, “application.value” equals 5, corresponding to a tour underway, and “need_camera.value” equals 1, a variable set by navigation stating that we are in a state where the camera is needed (i.e., drone’s not landed). To begin the tour an elected crowd leader will stand at 16 feet away. OpenCV’s Histogram of Ordered Gradients (HOG) detector is set to detect human descriptors. For every detection calculated by the HOG detector there is an associated confidence value. Higher values indicate greater confidence that what was detected is a person. Through testing a threshold value of 1.4 was discovered such that any value below this is a false-positive. Should detection fail or no values meet the threshold then the detector will wait five seconds and then try again. The detection is performed and a boundary box around the crowd leader is drawn. This box has a width and height that are saved as references along with the equivalent area (width times height). The boundary box is copied and cropped using a formula to get an approximation to the crowd leader’s shirt. x_0 is the upper left x value of the boundary box, y_0 is the upper left y value of the boundary box, w is the width of the bounding box from x_0, and h is the height of the boundary box from y_0.

$$(shirt\,x0,\ shirt\,y0,\ shirt\,w,\ shirt\,h) = (x0 + 0.33 * w,\ y0 + 0.25 * h,\ x0 + 0.66 * w,\ y + 0.5 * h)$$

The first cropped shirt image is saved as a reference and its histogram calculated; a histogram is an array of the number of pixels for every color value giving a numerical representation of the shirt image. This histogram is saved as a reference to be used later. A MOSSE tracker is generated and initialized with the generated boundary box. Movement of anything within this box will be monitored and followed allowing us to stay on the crowd leader. The MOSSE tracker is fast with less accuracy than other trackers but much cheaper computationally compared to constantly running the HOG detector. Speed was the main determiner of the tracker eliminating most but the MOSSE was ultimately chosen for proving the best at occlusion (when something covers the crowd leader momentarily) out of the two to choose from (MOSSE vs. MedianFlow). Having greater accuracy after occlusion means that we needn’t worry if non-tourists walk in front of our crowd.

```
tracker = cv2.TrackerMOSSE_create()
# initialize the person detector
hog = cv2.HOGDescriptor() #create descriptor
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector()) #set descriptor to find people
#initialize crowd leader and their boundary box - retry if failed
leader_found = 0
redetect_camera.value = 1 #tell snesors im going to do detection
cv2.startWindowThread()
while leader_found == 0:
    if connected.value == 1 and application.value == 5 and need_camera.value == 1:
        (rects, weights) = hog.detectMultiScale(feed, winStride=(4, 4), padding=(8, 8), scale=1.05) #detect
        print('number of crowd detections:', len(weights))
        if len(weights) != 0: #if we have detections
            if max(weights) > 1.4:
                print('leader found, continuing')
                leader_found = 1
                #assign names to dimensions and weight
                leadRect = rects[argmax(weights)]
                leadWeight = weights[argmax(weights)]
                x = leadRect[0];y = leadRect[1];w = leadRect[2];h = leadRect[3]
                cv2.rectangle(feed, (leadRect[0],leadRect[1]), (leadRect[0]+leadRect[2], leadRect[1]+leadRect[3]), (0, 0, 255), 2)
                redetect_camera.value = 0 #tell sensors im done detecting
                print("reference area is", w*h)
            else:
                print('camera retry in 5s')
                cv2.imshow("feed", feed)
                redetect_camera.value = 0 #give sensors some time to detect collisions
                time.sleep(0.8)
                redetect_camera.value = 1 #tell sensors i'm about to redetect
                time.sleep(0.2)
                #update the frame
                feed = strm.read()
                feed = imutils.resize(feed, width=360)
        else:
            print('camera retrying in 5s')
            cv2.imshow("feed", feed)
            redetect_camera.value = 0 #give sensors some time to detect collisions
            time.sleep(0.8)
            redetect_camera.value = 1 #tell sensors i'm about to redetect
            time.sleep(0.2)
            #update the frame
            feed = strm.read()
            feed = imutils.resize(feed, width=360)
```

Figure 3: Code Excerpt of Initial Detection of Crowd Leader

Once the tracker is initialized the tour begins and the tracker is continually updated as it moves the boundary box, following the crowd leader. This main tracking loop is inside of an if loop whose condition is that “need_camera.value” equals 1 and “application.value” equals 5, meaning the tour group tracker will not track or continue unless the camera is both needed and there is currently a tour underway. Testing of the natural inaccuracy and drift of the MOSSE tracker showed no problems for slow regular movement.This means that updating for accuracy was unnecessary but still necessary for updating box sizing. The MedianFlow tracker tested dynamically updates the box size but is less accurate meaning using either would guarantee the necessity of updating the new boundary box dimensions and thus relative distance. After ten seconds the recalibration begins and the HOG detector is run once more. Once humans in the camera frame are detected the recalibration logic loop begins. There are three possible scenarios based on the statuses of the detector and the tracker:

1. Detector found humans and the tracker is still active
2. Detector found humans and the tracker is not active
3. Detector found no humans and the tracker is either active or inactive

The first case is the most ideal. It is possible for the tracker to fail under large movement from either the crowd leader or the drone itself. When both are active then we have the greatest degree of accuracy. Firstly we want to ensure that we re-initialize on the right person so we need to find out which detection *is* the right person. A function was defined that checks the amount of overlap between two input boxes.

```
#define overlap area function
#finds overlapping rectangle dimensions and then
#returns overlappings rectangle area, otherwise 0 (no overlap)
def olap(a,b):
    oy = min(a[1]+a[3], b[1]+b[3]) - max(a[1], b[1])
    ox = min(a[0]+a[2], b[0]+b[2]) - max(a[0], b[0])
    if (ox > 0) and (oy > 0):
        return ox*oy
    else:
        return 0
```

Figure 4: Overlap Calculation Function

The minimum lower edge is subtracted from the maximum upper edge to get the overlap height. The minimum right edge is subtracted from the maximum left edge to get the overlap width. This function is iterated over all of the detections and compared with the original reference. Once all of the overlaps are calculated, the confidence values of each original box are checked. Only those above 1.4 are considered. The detection with the most overlap is chosen as the crowd leader and used to update the tracker and calculate the relative distance.

The second case is not the least ideal but it is the one that is most dreaded because it relies on the histogram function which is very computationally expensive. In the case where we have detections but no tracker to compare to and therefore confirm the crowd leader, we must manually do so. For all detections that pass the confidence threshold the boundary box is cropped to the shirt and the histogram calculated. Then all viable detections have their histogram compared to the original using a built-in histogram comparison function. The detection with the histogram that correlates the most to the reference is chosen again as the crowd leader. This detections bounding box is used to recalibrate.

The final case is the least ideal but the easiest to deal with. No special consideration is taken here as there is literally no data to work with besides possibly the tracker. As such the only thing to do is try and detect again to find detections so that we may enter either the first or second case.

Once the rectangle has been decided the tracker is re-initialized using this new box. The width and height of this new box are used to generate the associated area. The new area is compared to the original box area from 16 feet away and the appropriate response is sent out. If the new area is 35% greater than the original then they are too close and the program sends the signal to speed up. If the new area is only 68% of the original area (32% smaller) then we are too far and the signal is sent to slow down. If neither of these conditions is met then the crowd is

an acceptable distance and no change is initiated. The multiprocessing variable “tour_group.value” is manipulated and represents the state of the speed. “tour_group.value” is an integer that varies from 1 to -1 with 1 corresponding to 0.7 m/s, 0 corresponding to 0.5 m/s, and -1 corresponding to 0.3 m/s.

```
ratio = (float(newPosArea)/refArea)
#print(ratio)
#print(float(newPosArea/refArea))
#print((newPosArea/refArea)*100)
if ratio > 1.35:
            print('too close! let\'s speed up.')
            if tour_group.value != 1:
                tour_group.value += 1
            else:
                print('value capped on upper bound')
            print("speed set to", tour_group.value)

        elif ratio < 0.68:
            print('too far! let\'s slow down.')

            if tour_group.value != -1:
                tour_group.value -= 1

            else:
                print('value capped on lower bound')
            print("speed set to ",tour_group.value)

        else:
            print('just right :)')
            print("speed set to ", tour_group.value)

        resetTimer = 0 #reset our recalibration timer
        histValueList = [] #reset lists
        olapAreas = [] #reset overlap list
        newWeights = [] #reset newWeights list

        redetect_camera.value = 0 #inform sensors im done detecting/calculating
```

Figure 5: Excerpt of Distance Response

The last multiprocessing variable used is “redetect_camera.value”. This variable is used to communicate with the ultrasonic sensor process. It was discovered during testing that the strain on the processor from the redetection and histogram functions would delay the reading of the ultrasonic values. “redetect_camera.value” was the solution; three frames before the recalibration loop, or roughly 3/30 = 0.1 seconds, the variable is set to 1, alerting the sensor process to halt until the variable changes back to 0 when the functions are done executing.

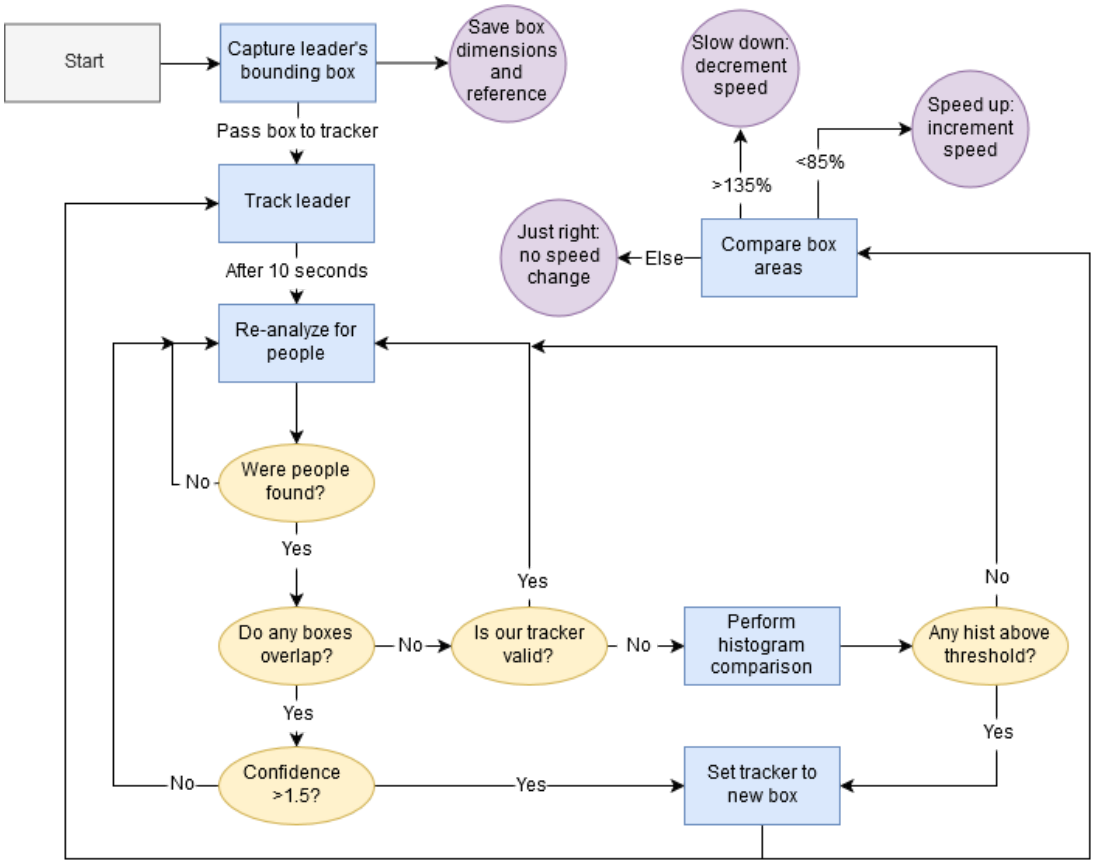


Figure 6: Flow Diagram of Crowd Tracker Program Logic

3.2.2. confidence.py

This code was written in addition to the main tour group tracker code in order to find the threshold value for which anything lower is a false-positive. It was also used to find the absolute The code consists of some initialization and a small loop that runs the HOG human descriptor detector on the current camera frame, draws the boundary box around the detections, and prints the confidence value above the box as well as to the console. The code then waits for a button press before repeating.

```
#setup videostream, warmup, read frame and resize it
strm = VideoStream(usePiCamera=1).start()
sleep(2)
hog = cv2.HOGDescriptor() #create descriptor
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector()) #set descriptor to find people
cv2.startWindowThread()
#initialize crowd leader and their boundary box - retry if failed
while 1:
    feed = strm.read()
    feed = imutils.resize(feed, width=360)
    (rects, weights) = hog.detectMultiScale(feed, winStride=(4, 4), padding=(8, 8), scale=1.05)
    for (rect, weight) in zip(rects, weights):
        x = rect[0];y=rect[1];w=rect[2];h=rect[3]
        cv2.rectangle(feed, (x,y), (x+w, y+h), (0, 0, 255), 2) #create a rectangle from detection
        yeah = numpy.array2string(weight)
        cv2.putText(feed, yeah, (x, y-5), cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 1)
        print(weight)
        print("Area = ",w*h)
        print("")
    cv2.imshow("Current Detections", feed)
    print("*****************")
    raw_input("Enter to continue")
```

Figure 7: confidence.py

3.2.3. dist_test.py

This code was written in addition to the main tour group tracker code in order to quantify the area ratios for the various distances. The code is essentially a stripped down recalibration loop without the histogram case. dist_test.py simply detects people in the frame and prints the area of the detected boundary boxes to the console. The data was then taken and manipulated to find the area ratios.

```
#setup videostream, warmup, read frame and resize it
strm = VideoStream(usePiCamera=1).start()
sleep(2)
# initialize the person detector
hog = cv2.HOGDescriptor() #create descriptor
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector()) #set descriptor to find people
cv2.startWindowThread()
while 1:
    feed = strm.read()
    feed = imutils.resize(feed, width=360)
    (rects, weights) = hog.detectMultiScale(feed, winStride=(4, 4), padding=(8, 8), scale=1.05) #detect
    if len(weights) != 0: #if we have detections
        if max(weights) > 1.5:
            #assign names to dimensions and weight
            leadRect = rects[argmax(weights)]
            leadWeight = weights[argmax(weights)]
            x = leadRect[0];y = leadRect[1];w = leadRect[2];h = leadRect[3]
            cv2.rectangle(feed, (leadRect[0],leadRect[1]), (leadRect[0]+leadRect[2], leadRect[1]+leadRect[3]), (0, 0, 255), 2)
            cv2.imshow("current detection", feed)
            print("current area", w*h)
            raw_input("Enter to continue")
        else:
            print("bad weight")
    else:
        print("no detections")
```

Figure 8: dist_test.py

4. Operation

Since the entirety of the the Tour Group Tracker subsystem consists of image processing most steps of validation and execution are performed visually. During testing the image output was enabled so that the current graphical output of the code could be monitored but this is disabled for normal operation due to the processing power displaying the image requires.

Operation of the code was performed visually and causally; a case would be setup and the outcome was analyzed. For example, testes were performed to confirm the operation of the distance estimation. Data would be gathered beforehand stating that the area ratio for 24 feet away would be 0.65. A tape measure would be set out and the case would be tested and it would be confirmed that each step executed properly by monitoring the console and debugging print statements added. The area ratio found would be checked if it was near the tested value of 0.65 and if the correct speed change was set out.

Other tests, especially early ones, were all visual confirmation. Testing correct operation of detecting and tracking meant looking at what was being detected and tracked and confirming it detected or tracked either successfully or what it was supposed to.

During normal operation the program was validated through the output at the console. Since there were no actual indoor flying tests due to safety, this is the best that could be done. As we would pretend to fly the drone down the hall the distance between the drone and one of our members varied and the output was checked to see if it matched what happened. If the appropriate action is taken then the tracker is tracking the correct person and the responses execute properly. Should the responses be inappropriate or there be no response then the tracking failed. The output of the ultrasonic sensor process was also monitored due to the fact that the multiprocessing variable in the detect and track code was implemented. The drone was setup in a way where the sensors measured known distances and while the ultrasonic and detect and track processes ran the values were checked to ensure accuracy with the multiprocessing variable.

Once everything is complete and integrated the system will be unnoticeable for the tourists (save for the initial calibration) unless a malfunction occurs. Were one of the components of this subsystem to fail then the drone would end the tour.

5. Data Collection

Early data gathered was mostly pass/fail or visually confirmed and left little in the way of presentable data. Either way this data has been included here as it was imperative to the operation of the code.

5.1. Distance Boundaries

The most important data to be collected when it comes to the operation of the subsystem is the actual boundaries at which detection can occur. This data was not the first to occur and was tested during the second semester since it was not until integration that the camera was finally mounted to the drone. It was important that the distance boundaries be tested from this point since the camera was now in the position, height, and angle that it would be during final testing.

The data was captured by using “confidence.py” to find the absolute distances beyond which detection was no longer possible. These distance represent where we don’t want our crowd to venture and thus the actual distances used for speed-up/slow-down thresholds are moved in some. The confidence data captured for the distance boundaries is below.

Distance	Confidence
< 9 ft	N/A
9 ft	0.876
12 ft	1.347
15 ft	2.158
18 ft	2.747
21 ft	3.362
24 ft	3.492
> 24 ft	N/A

Table 1: Distance Confidence Values

Closer than nine feet to the drone meant that, due to the height and angle of the camera, not enough of the person was visible to allow detection. Beyond twenty-four feet meant the person was too far away and thus too low resolution to be detected. To keep the crowd leader from getting to these points the speed-up threshold was set at roughly thirteen feet and the slow-down threshold set at about twenty-two feet

5.2. Confidence Threshold for Filtering of False Positives

The “confidence.py” code was also used to test for the threshold beyond which would signify a true positive human detection. The tour begins at the Anadarko learning center whose front wall is entirely glass. A crowd leader would stand in front of this glass during the initial detection which caused issues early on due to reflections. This area was used for testing as it offered the most room for false-positives. A team member would stand against the glass and the drone would be set up from various angles and distances in an attempt to find detections that we would want to filter. Most were reflections but some were random arrangements of objects that whose histogram of ordered gradients apparently somewhat matches those of a human. Several cases were also gathered for true positives to find data on the average and minimum confidence. The table of data gathered can be seen below along with some images captured during data gathering.

+Desc.	Confidence Values	Confidence Values	-Desc.
(Positives)	Positive	False Positive	(False Positives)

Forward	3.04	0.22	Reflection: Person
Forward	2.83	0.87	Reflection: Person
Forward	3.17	0.56	Reflection: Person
Forward, arms out	1.89	1.16	Trash Bags
Forward, hand to face	2.46	0.47	Random
Forward	1.83	0.52	Reflection: Random
Forward	2.43	0.67	Extra
Forward	2.91	0.65	Reflection: Random
Holding Laptop	1.7	0.77	Extra
Forward	3.74	0.78	Extra
Holding Laptop	1.82		
Holding Jacket	2.89		
Holding Laptop	2.18		
Phone, Extra?	1.42		
Hands in pockets	3.09		
Blurry	3		
Side View	1.85		
Forward	3.08		
Hands on Hips	2.97		
Scratching Face	2.38		
Hands in front	3.27		
Profile View	2.73		
Forward	3.49		
Average	2.616086957	0.678333333	**Average**
Min	1.42		**Min**

Max		1.16	**Max**
Median	2.83	0.56	**Median**

Table 2: Confidence Values for Positives and False Positives

Data above was captured at the optimum distance of around sixteen feet. The leftmost and rightmost columns list descriptions of the detection listed. Positive descriptions refer to the position of the person detected, such as facing forward or what they were doing at that moment. For false positives descriptions refer to what was erroneously captured such as reflections of people, reflections of miscellaneous objects that must have resembled a person, and extra refers to boundary boxes that are larger than normal due to “extra” information in the frame. Extra cases are technically positives that are incorrectly sized due to false positive information such as reflections. Descriptions for each detection found were recorded for the sake of rationalizing why confidence values were higher or lower. Information can be gleaned about the HOG human descriptor such as the fact that relaxed arms-at-side poses register best versus arms-out-Vitruvian-man style poses. Other important information such as the fact that even under movement causing blur the descriptor works well. Only the minimum confidence was important for the positives as we need to find the worst case that is still viable for use whereas the maximum false positive confidence value is important in finding the the most deceiving case.

Figure 9: Example of Positive Detections Collected

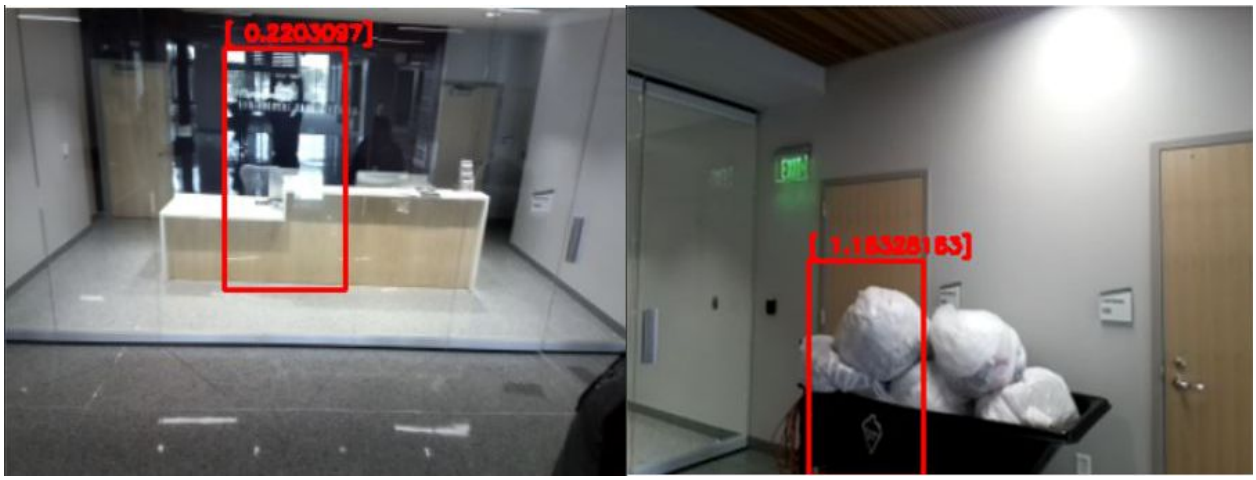


Figure 10: Examples of False Positive Detections Collected

5.3. Relative Distance Data

The first data to be collected was how the area of the boundary boxes changed with respect to the distance. This data was updated later with the camera in its final mounted position and angle. The crowd leader's boundary box area is used as a reference as opposed to an actual calculation of distance using focal length because all algorithms using that method require knowing the size of the object in question beforehand.. By using the initial boundary box area we dynamically create a constant tailored to that person's height and width as opposed to dictating an average that would "hopefully" work for everyone.

By using the box area instead of a direct distance a method needed to be established to determine the change in distance. The ratio between the new area and the reference area was chosen. To determine this a test was performed where a teammate stood on a measuring tape and a test program was set to run such that the ratio of the box to the new area was calculated and printed. Several iterations were run such that the boundary box varied naturally and an average box area for that distance was calculated. Finally, the percent difference between the boxes at the maximum and minimum distances were calculated for those averages, giving us an average percentage change from the reference at that distance.

Reference Area = 18432 px^2

Distance (ft)	**Avg. Ratio to Reference**
25	0.56
7	1.22

Table 3: 403 Distance Ratios

Reference Area = 15051 px^2, at 15 feet

Distance (ft)	**Avg. Ratio to Reference**
15 ft (second pass)	1.1
20 ft	0.727

22 ft	0.656
24 ft	0.61
18 ft	0.83
12 ft	1.41

Table 4: 404 Distance Ratios

The variables used in the “main.py” process 4 code come from this second collection of data. The ratio for maximum distance was set at 0.65 and the ratio for minimum distance was set at 1.4.

5.4. Tracker Decision

The next set of data that needed to be collected was which tracker to be used. After narrowing down to the Median Flow and the MOSSE trackers due to their speed and efficiency it had to be chosen which tracker performed the best at its job. A set of cases were set up and each tracker was initialized to find their strengths and weaknesses. Each of them performed as expected. Median Flow is the least costly but fails under occlusion and large motion. This held true as during the occlusion and large motion tests the tracker failed. The data has been compiled in the following table.

Test / Tracker	**MOSSE**	**Median Flow**
Normal Motion	Pass	Pass
Occlusion (with movement)	Pass	Fail
Large Motion	Fail	Fail

Table 5: Tracker Test Cases

As can be seen in the table above both trackers did not do well with large movement. This is mainly because they are meant for slower normalized movement (such as walking down a hallway). As lamentable as a failing test is the choice is clear; the MOSSE tracker failed the least and thus was the obvious choice. Both succeeded in normal fluid motion but only the MOSSE passed the test where a teammate passed in front of the teammate being tracked. The MOSSE tracker successfully reset itself on the right person after being occluded. For now, large motion will be treated as an anomaly seeing as a crowd of tourists are not expected to make wild sweeping motions across the hallways.

5.5. Logic Validation

The validation testing for the logic and cases was purely boolean; did the correct thing happen? The code was run and the output images and console text analyzed to see if what

should have happened in that scenario happened. This data is not wholly quantifiable beyond the fact that everything worked as expected. An example of a test is shown below. In the image we can see the output of the detector, the current tracker, and the console. In the console we see the correct output that the person being tracked is not too far away. We can also see the three humans detected previously and that in the above window the correct person (in the middle) is still being tracked. The ratio of the old and new boxes is also referenced to verify if the correct speed change message was sent out.

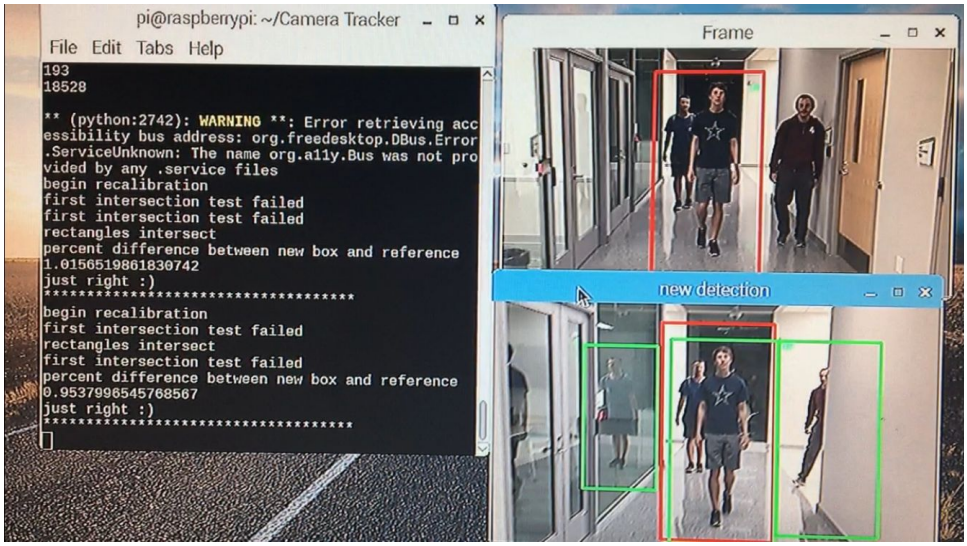


Figure 11: Example of Code Running for Visual Confirmation

5.6. Performance Measurement and Ultrasonic Correction

Image processing is inherently processor-intensive. It was important to quantify several aspects of how much processing time the tour group tracker code used as well as how it affected the other systems. It was discovered early on during integration that the complex functions like the histogram calculation and the histogram of ordered gradients analysis would slow down or stop the pipeline for other processes. This meant that the polling for ultrasonic data was thrown off; the ultrasonic sensors send a pulse and read the returned value after a certain amount of time. While the complex functions were running the length of the execution was added to the time until the reading of the return. The amount of time until the reading of the returned value is important because of the analog nature of the sensor; delaying the reading means you read an incorrect value.The amount of delay added had to be quantified and a solution developed to ensure the proper operation of the ultrasonic sensors due to their critical importance. It was discovered that the histogram comparison calculation added a maximum delay of one second to the reading of the returned ultrasonic value. To circumvent this a multiprocessing variable was implemented that, when read by the ultrasonic transceiver code, would halt the transmission and receiving of the ultrasonic ping. This leaves us blind for about one second but the HOG detector is only run once every ten seconds and the histogram functions are used only when necessary. This solution is not exactly elegant but it is the best

that can be done because it is a problem inherent to the functions, the hardware, and the multiprocessing pipeline.

6. Accomplishments

There were a number of landmark accomplishments in the development of the Tour Group Tracker subsystem. Small programs were written to test the various components that makeup the entire subsystem for testing and verification. Interfacing them together was another issue.

The first hurdle that had to be passed was actually choosing to use the camera for this subsystem. There were a few proposed ideas that were tossed around and considered before the camera due to the relative complexity. It seemed overkill to use a camera and image processing to simply detect how far away a crowd was from the drone. As it turned out the complexity was important as most other methods, namely Bluetooth, proved too simple to provide accurate measurements. Bluetooth signal to distance conversion was the initial choice for tracking the crowd distance. This proved disastrous as Bluetooth signals are not reliable. With all other choices stricken the camera was the only choice left.

Once the camera was chosen and the various components of the actual tracking were confirmed to work the problem of integration became apparent. Many hours were spent attempting to get the HOG human detector to interface with the tracking; mostly due to ignorance and a lack of documentation or examples. With plenty of work they eventually integrated.

The last big accomplishment was the biggest; the logic for re-initializing the tracker. Logic consists of several nested loops each with conditions that must be passed. These make up the various cases each with specific checks. The loops were built outside in with the beginning and ends of everything written first and the more complex inner functions added afterward. First the loop to redetect, recalibrate, and calculate area was created. With new detections performed there needed to be a way to iterate through them and eliminate unnecessary ones. Cases were discovered that needed addressing such as how to reorient with no tracker. Conditions needed to be added after integration for loops to either start or break and for communication with navigation or the ultrasonic measurement code. As more conditions and checks were needed as the logic was amended and new cases needed to be considered. The standalone and integrated versions of the code were run numerous times each with a scrutinous eye to ensure proper operation.

In the end a working program that could detect a crowd leader, track them, reinitialize itself, and respond to their distance was created.

7. Future Work

While the program works well and takes many possible cases into consideration there is still more than can be done. More conditions could be implemented for possible cases not yet considered. As it stands, not all cases can be considered due to the sheer number of combinations of possibilities. Optimization is a big chunk of work that could be done to ensure

smoother operation of other subsystems. The code was trimmed down as is but there are certainly better coding techniques that could be utilized instead of, for example, nested loops or incremented timer variables to count frames. Functionality is complete and any future work would be merely improvement or additional.

8. Conclusion

A lot was learned these past semesters. What was originally thought to be a hardware heavy project turned out to be one where everyone ended up programming. I learned a lot about programming in general as well as Python specifically. Image processing was a subject I had never dipped my feet into and for this project I dove into the deep end. This required lot of research into how detection is done and what others have accomplished with it. I'd like to say that I have gained a few new skill sets through this class and I am proud of the work that I have accomplished. The Tour Group Tracker subsystem is by no means perfect and could benefit from some more hard work to make it something marketable. As it stands the program is beyond a proof-of-concept having done more than just prove it's possibility. On the whole the project came out better than was expected at our low points. All of us encountered troubles that we did not expect but were able to adapt and overcome them to get to the point where, even with compromise, we achieved what we set out to do. Through our combined effort both in our individual subsystems and the willpower to meet every day for several hours we have come out with a project we can all be proud of.

Zach Drone

Grayson Vansickle

Mobile Application Subsystem

REVISION – 2.0
28 April 2019

Table of Contents

List of Figures

List of Tables

No tables at this time.

1. Introduction

The purpose of the Mobile Application subsystem of the Zach Drone, which is an automated drone tour throughout the new Zachry Engineering Education Complex, is to allow the user to connect to the drone and tell it which tour to take. The mobile application will also allow the user to pause and resume the tours if someone in their tour group needs to use the bathroom or needs to stop for an extended period of time, and the user will be able to end the current tour and select a new tour. This subsystem will involve a user interface between the user and the drone and will be connected using a WiFi signal to communicate between the mobile application and the microcontroller on the drone, a Raspberry Pi 3 B. The mobile application will be available for the user to download on both Android and iOS platforms. Based on the user input, this subsystem interacts with the navigation subsystem to communicate which tour and tour destinations the user has picked. The user input is a major factor regarding the navigation of the drone, since different tours will need to be navigated differently in order to visit all of the destinations specific to the tour. The mobile application consists of a main page, an art tour page, a classroom tour page, a basic tour page and a drone page. The three tour pages will show the user which destinations are on the tour of the page they selected and will allow the user to start the tour on the page. The drone page will allow the user to connect and disconnect from the drone, along with pause, resume and end the current tour. The Raspberry Pi 3 B will have a message decoder running to ensure the drone performs the correct action in response to the user's input. The subsystem overview is shown in Figure 1. This document will describe the theory, design, operation, data collection and validation of the mobile application subsystem.

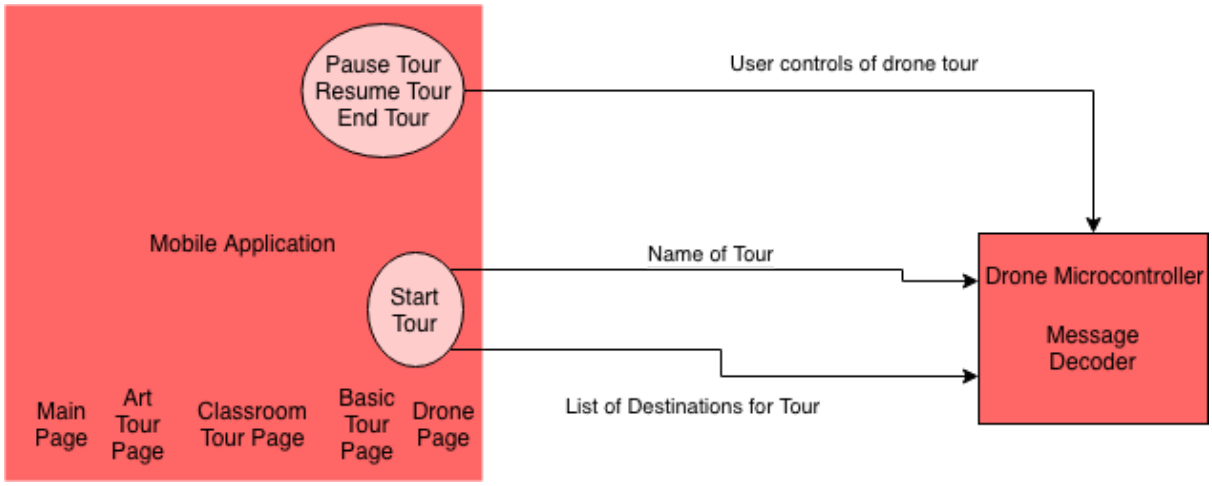


Figure 1: Mobile Application Subsystem

2. Theory

2.1. Mobile Application Code Logic

The mobile application will connect to the drone's onboard Raspberry Pi 3 B. This connection will allow the user to communicate with the drone to give it commands. The logic used in the iOS and Android applications follow the diagram below. The logic in the figure gives the user some control of what the application sends to the drone but restricts them to an extent that they won't make the drone malfunction due to an overload of messages. The users must first connect to the drone before any other function of the application will work. After the user connects to the drone, they will then be able to start a tour and disconnect from the drone. The pause and end tour buttons only function after the user has started a tour. The resume tour button only functions after the user has paused a tour.

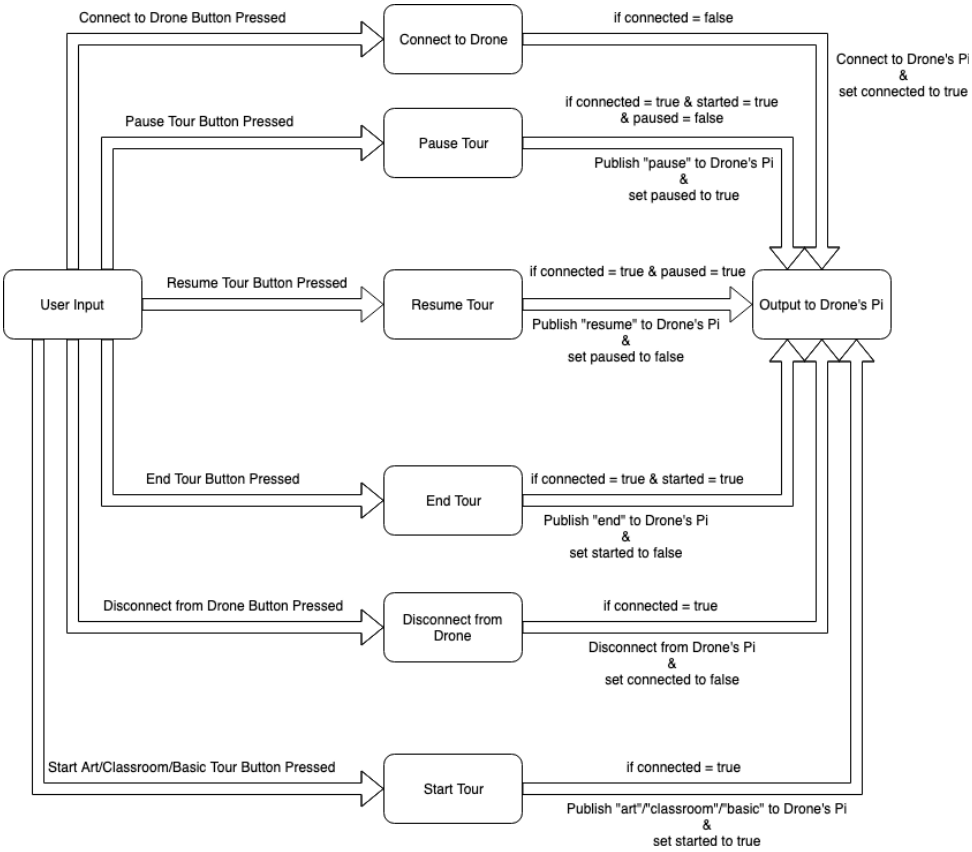


Figure 2: Application Logic used for the code

3. Design

3.1. Swift Code

The backend of the iOS application is written using Xcode in the Swift programming language. The logic of the the following swift files follows the logic defined in Figure 2 in the Theory section.

3.1.1. MainViewController.swift

MainViewController.swift contains only one function and that function is used to load the view. This function is needed so that the iOS application shows the correct page of the app. There is a print statement included in this function which was used when debugging that shows that the correct view was loaded.

```
import UIKit

class MainViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        print("Main View Loaded")
    }
}
```

Figure 3: Load MainViewController function

3.1.2. DroneViewController.swift

DroneViewController.swift contains a struct used for the client details and six functions. This struct contains the client itself and three boolean values used to make sure the user can online push certain buttons. The client uses the CocoaMQTT external library which is given a client id name, a host ip address and a port number. In this case the client id name is Zach Drone iOS Device, the host ip address is the ip address of the drone's Raspberry Pi 3 B on the WiFi network, and the port number is 1883. The three boolean values are connected, paused and started. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The six functions in DroneViewController.swift are load view, connect button, pause button, resume button, end button and disconnect button functions. The load view function is the same as the load view function in MainViewController.swift, but it loads the drone view instead of the main view.

```
struct Client {
    //home WIFI
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "192.168.1.138", port: 1883)
    //Hotspot
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "172.20.10.10", port: 1883)
    //Zach Guest
    //static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "10.236.61.140", port: 1883)
    //TAMU WPA
    static let mqttClient = CocoaMQTT(clientID: "Zach Drone iOS Device", host: "10.236.3.30", port: 1883)
    static var connected = Bool()
    static var paused = Bool()
    static var started = Bool()
}

class DroneViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        print("Drone View Loaded")
    }
```

Figure 4: Client struct and load view function defined

The connect button function is executed when the Connect to Drone button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value is true. If the connected value is true then the function will publish the message “connect” to the topic used for this project, “zach/drone/rpi/tour”, and print “Already connected to drone” for debugging. There will be an alert that pops on the screen to tell the user that they have already connected to the drone. If the connected value is false then the function will connect the the mqttClient defined in the Client struct, print the client’s host name along with connect, and set the connected value to true.

```
//Executes when Connect to Drone button gets pressed
@IBAction func connectButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "connect")
        print("Already connected to drone")
        let alert = UIAlertController(title: "Did you already connect to the drone?", message: "You can only press the \"Connect to Drone\" button once.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        Client.mqttClient.connect()
        print(Client.mqttClient.host + "  connect")
        Client.connected = true
    }
}
```

Figure 5: Connect button function defined

The pause button function is executed when the Pause Tour button is pressed. This function checks to see if the drone is already connected to first, a tour has already been started and the pause button hasn’t already been pressed, if the connected and started boolean values are true and paused boolean value is false. If the connected and started values are true then the function will publish the message “pause” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name along with pause, and set the paused value to true. If the connected boolean value is true but the started boolean value is false then the function will print “Need to start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to start a tour before pausing a tour. If the connected and started boolean values are false then the function will print “Need to connect to drone and start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and start a tour before pausing a tour.

```
//Executes when Pause Tour button gets pressed
@IBAction func pauseButton(_ sender: UIButton) {
    if(Client.connected && Client.started && !Client.paused) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "pause")
        print(Client.mqttClient.host + "  pause")
        Client.paused = true
    }
    else if(Client.connected && !Client.started) {
        print("Need to start a tour first")
        let alert = UIAlertController(title: "Did you start a tour?", message: "You must press the \"Start
            Tour\" button before pressing \"Pause Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else if(Client.paused) {
        print("Tour already paused")
        let alert = UIAlertController(title: "Tour already paused", message: "To resume the tour press the
            \"Resume Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and start a tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and start a tour?", message:
            "You must press the \"Connect to Drone\" and \"Start Tour\" buttons before pressing the
            \"Pause Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 6: Pause button function defined

The resume button function is executed when the Resume Tour button is pressed. This function checks to see if the drone is already connected to first and a tour has already been paused, if the connected and paused boolean values are true. If the connected and paused values are true then the function will publish the message “resume” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name along with resume, and set the paused value to false. If the connected boolean value is true but the paused boolean value is false then the function will print “Need to pause tour first” for debugging and an alert will pop on the screen to tell the user that they have to pause a tour before resuming a tour. If the connected and paused boolean values are false then the function will print “Need to connect to drone and pause a tour first” for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and pause a tour before resuming a tour.

```
//Executes when Resume Tour button gets pressed
@IBAction func resumeButton(_ sender: UIButton) {
    if(Client.connected && Client.paused) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "resume")
        print(Client.mqttClient.host + "  resume")
        Client.paused = false
    }
    else if(Client.connected && !Client.paused) {
        print("Need to pause tour first")
        let alert = UIAlertController(title: "Did you pause a tour?", message: "You must press the \"Pause Tour\" button before pressing \"Resume Tour\" button.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and pause tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and pause a tour?", message: "You must press the \"Connect to Drone\" and \"Pause Tour\" buttons
            before pressing the \"Resume Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 7: Resume button function defined

The end button function is executed when the End Tour button is pressed. This function checks to see if the drone is already connected to first and a tour has already been started, if the connected and started boolean values are true. If the connected and started values are true then the function will publish the message “end” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name along with end, and set the started value to false. If the connected boolean value is true but the started boolean value is false then the function will print “Need to start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to start a tour before ending a tour. If the connected and started boolean values are false then the function will print “Need to connect to drone and start a tour first” for debugging and an alert will pop on the screen to tell the user that they have to connect to the drone and start a tour before ending a tour.

```
//Executes when End Tour button gets pressed
@IBAction func endButton(_ sender: UIButton) {
    if(Client.connected && Client.started) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "end")
        print(Client.mqttClient.host + "  end")
        Client.started = false
    }
    else if(Client.connected && !Client.started) {
        print("Need to start a tour first")
        let alert = UIAlertController(title: "Did you start a tour?", message: "You must press the \"Start Tour\" button before pressing \"End Tour\" button.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
    else {
        print("Need to connect to drone and start a tour first")
        let alert = UIAlertController(title: "Did you connect to the drone and start a tour?", message: "You must press the \"Connect to Drone\" and \"Start Tour\" buttons
            before pressing the \"End Tour\" button.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 8: End button function defined

The disconnect button function is executed when the Disconnect from Drone button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value is true. If the connected value is true then the function will publish the message “disconnect” to the topic used for this project, “zach/drone/rpi/tour”, print the client’s host name and disconnect, disconnect the mqttClient defined the the Client struct, and set the connected value to false. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```swift
//Executes when Disconnect from Drone button gets pressed
@IBAction func disconnectButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "disconnect")
        print(Client.mqttClient.host + "  disconnect")
        Client.mqttClient.disconnect()
        Client.connected = false
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button before pressing any other buttons.",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 9: Disconnect button function defined

3.1.3. ArtViewController.swift

ArtViewController.swift contains two functions and an array of strings. The two functions are the view load and start art button functions. The array of strings contains the list of all the destinations that the art tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the art view instead of the main view.

```swift
class ArtViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Art Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1.  Anadarko Welcome Center", "2.  Smoke Painting #44 by Rosemarie Fiore", "3.  A-Gadda-Da-Vida by Jay Shinn", "4.
        Lathocyte by Andy Vogt", "5.  Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales", "6.  Shapeshifting to Transcend Limbo
        by Lyndi Sales", "7.  What it Takes to Make by Daniel Rozin", "8.  Pulse by Daniel Canogar", "9.  Prototype for Stellar Interloper
        (Silver Surfer) by Inigo Manglano-Ovalle", "10. Infinitesimals by Rusty Scruby"]
```

Figure 10: Load view function and art tour destinations defined

The start art button function is executed when the Start Art Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “art” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and art, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```
//Executes when Start Art Tour button gets pressed
@IBAction func startArtButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "art")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  art")
        Client.started = true;
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 11: Art start function defined

3.1.4. ClassroomViewController.swift

ClassroomViewController.swift contains two functions and an array of strings. The two functions are the view load and start classroom button functions. The array of strings contains the list of all the destinations that the classroom tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the classroom view instead of the main view.

```
class ClassroomViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Classroom Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1. Anadarko Welcome Center", "2. 106 - Virginia Brown Atrium", "3. 282 - Leach Learning Resource Center", "4. 212 - Small
        Learning Studio", "5. 343A", "6. 318 - Open Study Area", "7. 317 - Design Studio", "8. Large Learning Studios", "9. Fischer Engineering
        Design Center"]
```

Figure 12: Load view function and classroom tour destinations defined

The start classroom button function is executed when the Start Classroom Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “classroom” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and classroom, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```
//Executes when Start Classroom Tour button gets pressed
@IBAction func startClassroomButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "classroom")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  classroom")
        Client.started = true
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 13: Classroom start function defined

3.1.5. BasicViewController.swift

BasicViewController.swift contains two functions and an array of strings. The two functions are the view load and start basic button functions. The array of strings contains the list of all the destinations that the basic tour will visit. The load view function is the same as the load view function in MainViewController.swift, but it loads the basic view instead of the main view.

```
class BasicViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Basic Tour View Loaded")

        // Do any additional setup after loading the view.
    }

    let destinations = ["1.  Anadarko Welcome Center", "2.  Smoke Painting #44 by Rosemarie Fiore", "3.  Fischer Engineering Design Center", "4.
        Common Labs", "5.  A-Gadda-Da-Vida by Jay Shinn", "6.  HPE Tech Deck", "7.  481 - Academic Advising", "8.  Infinitesimals by Rusty Scruby",
        "9.  Photo Spot", "10. Large Learning Studios", "11. 343A", "12. 318 - Open Study Area", "13. 317 - Design Studio", "14. 212 - Small
        Learning Studio", "15. Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales", "16. 280 - ConocoPhillips Atrium", "17. 282 -
        Leach Learning Resource Center", "18. Pulse by Daniel Canogar", "19. What it Takes to Make by Daniel Rozin", "20. Prototype for Stellar
        Interloper (Silver Surfer) by Inigo Manglano-Ovalle", "21. 284", "22. Starbucks - 271", "23. Shapeshifting to Transcend Limbo by Lyndi
        Sales", "24. Lathocyte by Andy Vogt", "25. South-facing window", "26. 106 - Virginia Brown Atrium"]
```

Figure 14: Load view function and basic tour destinations defined

The start basic button function is executed when the Start Basic Tour button is pressed. This function checks to see if the drone is already connected to first, if the connected boolean value from the Client struct in the DroneViewController.swift file is true. If the connected value is true then the function will publish the message “basic” to the topic used for this project, “zach/drone/rpi/tour”, send all the destinations in the array of strings to the same topic, print the client’s host name from the Client struct in the DroneViewController.swift file and basic, and set the started boolean value from the Client struct in the DroneViewController.swift file to true. If the connected value is false then the function will print “Need to connect to drone first” for debugging and an alert will pop on screen to tell the user that they have to connect to the drone before doing anything else.

```swift
//Executes when Start Basic Tour button gets pressed
@IBAction func startBasicButton(_ sender: UIButton) {
    if(Client.connected) {
        Client.mqttClient.publish("zach/drone/rpi/tour", withString: "basic")
        for destination in destinations {
            Client.mqttClient.publish("zach/drone/rpi/tour", withString: destination)
        }
        print(Client.mqttClient.host + "  basic")
        Client.started = true
    }
    else {
        print("Need to connect to drone first")
        let alert = UIAlertController(title: "Did you connect to the drone?", message: "You must press the \"Connect to Drone\" button on the
            Drone (far right) page before pressing any other buttons.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .cancel, handler: nil))
        self.present(alert, animated: true)
    }
}
```

Figure 15: Basic start function defined

3.2. Java Code

The backend of the Android application is written using Android Studio in the Java programming language. The logic of the the following java files follows the logic defined in Figure 2 in the Theory section.

3.2.1. ArtActivity.java

ArtActivity.java contains three strings, three boolean values, and two main functions, on create and art functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are ART_CONNECT, ART_START and ART_PAUSE. The ART_CONNECT string is used to store the connected boolean value. The ART_START string is used to store the started boolean value. The ART_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The art function has a string array that has all of the destination that the art tour will visit.

```java
public class ArtActivity extends AppCompatActivity {

    public static final String ART_CONNECT = "com.example.graysonvansickle.zachdrone.ART_CONNECT";
    public static final String ART_START = "com.example.graysonvansickle.zachdrone.ART_START";
    public static final String ART_PAUSE = "com.example.graysonvansickle.zachdrone.ART_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 16: Art Activity strings and booleans defined

The on create function starts the art activity layout xml file and displays this layout to the application. The function then sets the art tour item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named ART_CONNECT, ART_START or ART_PAUSE. If the intent has an ART_CONNECT extra then

the connected boolean value is set to the value of the intent's ART_CONNECT extra. If the intent has an ART_START extra then the started boolean value is set to the value of the intent's ART_START extra. If the intent has an ART_PAUSE extra then the paused boolean value is set to the value of the intent's ART_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_art);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 1);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(ART_CONNECT))
        connected = getIntent().getBooleanExtra(ART_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(ART_START))
        started = getIntent().getBooleanExtra(ART_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(ART_PAUSE))
        paused = getIntent().getBooleanExtra(ART_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + ART_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 17: Art Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the ArtActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the ArtActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the BasicActivity class. The

connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the ArtActivity class to the BasicActivity class is then started so the application will then run the BasicActivty.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the ArtActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the ArtActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: ArtActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent2 = new Intent( packageContext: ArtActivity.this, ClassroomActivity.class);
                intent2.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent2.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent2.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: ArtActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: ArtActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 18: Art Activity switch pages function defined

The art function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to art, an alert will pop up at the bottom of the screen that says art to tell the user that the art tour is start, and the started boolean value is set to true. The function then publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the art tour.

```
@RequiresApi(api = Build.VERSION_CODES.N)
public void art(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("art".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: ArtActivity.this,  text: "art", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1.  Anadarko Welcome Center",
                "2.  Smoke Painting #44 by Rosemarie Fiore",
                "3.  A-Gadda-Da-Vida by Jay Shinn",
                "4.  Lathocyte by Andy Vogt",
                "5.  Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales",
                "6.  Shapeshifting to Transcend Limbo by Lyndi Sales",
                "7.  What it Takes to Make by Daniel Rozin",
                "8.  Pulse by Daniel Canogar",
                "9.  Prototype for Stellar Interloper (Silver Surfer) by Inigo Manglano-Ovalle",
                "10. Infinitesimals by Rusty Scruby"};

        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: ArtActivity.this,  text: "Unable to Start Art Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 19: Art Activity art function defined

3.2.2. ClassroomActivity.java

ClassroomActivity.java contains three strings, three boolean values, and two main functions, on create and classroom functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are CLASSROOM_CONNECT, CLASSROOM_START and CLASSROOM_PAUSE. The CLASSROOM_CONNECT string is used to store the connected boolean value. The CLASSROOM_START string is used to store the started boolean value. The CLASSROOM_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the classroom tour will visit.

```
public class ClassroomActivity extends AppCompatActivity {

    public static final String CLASSROOM_CONNECT = "com.example.graysonvansickle.zachdrone.CLASSROOM_CONNECT";
    public static final String CLASSROOM_START = "com.example.graysonvansickle.zachdrone.CLASSROOM_START";
    public static final String CLASSROOM_PAUSE = "com.example.graysonvansickle.zachdrone.CLASSROOM_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 20: Classroom Activity strings and booleans defined

The on create function starts the classroom activity layout xml file and displays this layout to the application. The function then sets the classroom tour item in the bottom navigation

bar to be the selected item. The function also checks to see if the intent has an extra named CLASSROOM_CONNECT, CLASSROOM_START or CLASSROOM_PAUSE. If the intent has an CLASSROOM_CONNECT extra then the connected boolean value is set to the value of the intent's CLASSROOM_CONNECT extra. If the intent has an CLASSROOM_START extra then the started boolean value is set to the value of the intent's CLASSROOM_START extra. If the intent has an CLASSROOM_PAUSE extra then the paused boolean value is set to the value of the intent's CLASSROOM_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_classroom);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 2);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(CLASSROOM_CONNECT))
        connected = getIntent().getBooleanExtra(CLASSROOM_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(CLASSROOM_START))
        started = getIntent().getBooleanExtra(CLASSROOM_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(CLASSROOM_PAUSE))
        paused = getIntent().getBooleanExtra(CLASSROOM_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + CLASSROOM_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 21: Classroom Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the ClassroomActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named

ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the ClassroomActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the ClassroomActivity class to the BasicActivity class is then started so the application will then run the BasicActivty.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the ClassroomActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the ClassroomActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: ClassroomActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: ClassroomActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: ClassroomActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: ClassroomActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 22: Classroom Activity switch pages function defined

The classroom function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to classroom, an alert will pop up at the bottom of the screen that says classroom to tell the user that the classroom tour is start, and the started boolean value is set to true. The function then

publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the classroom tour.

```
public void classroom(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("classroom".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: ClassroomActivity.this,  text: "classroom", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1. Anadarko Welcome Center",
                "2. 106 - Virginia Brown Atrium",
                "3. 282 - Leach Learning Resource Center",
                "4. 212 - Small Learning Studio",
                "5. 343A",
                "6. 318 - Open Study Area",
                "7. 317 - Design Studio",
                "8. Large Learning Studios",
                "9. Fischer Engineering Design Center"};

        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: ClassroomActivity.this,  text: "Unable to Start Classroom Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 23: Classroom Activity classroom function defined

3.2.3. BasicActivity.java

BasicActivity.java contains three strings, three boolean values, and two main functions, on create and basic functions. The three strings are used to pass the boolean values between the different pages within the application. The three strings are BASIC_CONNECT, BASIC_START and BASIC_PAUSE. The BASIC_CONNECT string is used to store the connected boolean value. The BASIC_START string is used to store the started boolean value. The BASIC_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the basic tour will visit.

```
public class BasicActivity extends AppCompatActivity {

    public static final String BASIC_CONNECT = "com.example.graysonvansickle.zachdrone.BASIC_CONNECT";
    public static final String BASIC_START = "com.example.graysonvansickle.zachdrone.BASIC_START";
    public static final String BASIC_PAUSE = "com.example.graysonvansickle.zachdrone.BASIC_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 24: Basic Activity strings and booleans defined

The on create function starts the basic activity layout xml file and displays this layout to the application. The function then sets the basic tour item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named BASIC_CONNECT, BASIC_START or BASIC_PAUSE. If the intent has an BASIC_CONNECT extra then the connected boolean value is set to the value of the intent's BASIC_CONNECT extra. If the intent has an BASIC_START extra then the started boolean value is set to the value of the intent's BASIC_START extra. If the intent has an BASIC_PAUSE extra then the paused boolean value is set to the value of the intent's BASIC_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_basic);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 3);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(BASIC_CONNECT))
        connected = getIntent().getBooleanExtra(BASIC_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(BASIC_START))
        started = getIntent().getBooleanExtra(BASIC_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(BASIC_PAUSE))
        paused = getIntent().getBooleanExtra(BASIC_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + BASIC_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 25: Basic Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the BasicActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean

value is then added to this intent as an extra named ART_PAUSE. The intent created from the BasicActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the BasicActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the BasicActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the BasicActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: BasicActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: BasicActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent3 = new Intent( packageContext: BasicActivity.this, ClassroomActivity.class);
                intent3.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent3.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent3.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_basic: //Basic Tour
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: BasicActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 26: Basic Activity switch pages function defined

The basic function first checks to see if the connected boolean value is true. If connected is true then the function will create a new MqttMessage and this message is then set to basic, an alert will pop up at the bottom of the screen that says basic to tell the user that the basic tour is start, and the started boolean value is set to true. The function then publishes the message to the topic and client defined in the DroneActivity class. The function then goes through the array of destinations and makes a new MqttMessage with each destination and publishes that

message to the topic and client defined in the DroneActivity class. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before starting the basic tour.

```
public void basic(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("basic".getBytes());

        System.out.println("bool: " + connected);

        Toast.makeText( context: BasicActivity.this,  text: "basic", Toast.LENGTH_LONG).show();

        started = true;

        String[] destinations = new String[] {"1.  Anadarko Welcome Center",
                "2.  Smoke Painting #44 by Rosemarie Fiore",
                "3.  Fischer Engineering Design Center",
                "4.  Common Labs",
                "5.  A-Gadda-Da-Vida by Jay Shinn",
                "6.  HPE Tech Deck",
                "7.  481 - Academic Advising",
                "8.  Infinitesimals by Rusty Scruby",
                "9.  Photo Spot",
                "10. Large Learning Studios",
                "11. 343A",
                "12. 318 - Open Study Area",
                "13. 317 - Design Studio",
                "14. 212 - Small Learning Studio",
                "15. Transcending Realms; Chaos and Flow, Love and Fear by Lyndi Sales",
                "16. 280 - ConocoPhillips Atrium",
                "17. 282 - Leach Learning Resource Center",
                "18. Pulse by Daniel Canogar",
                "19. What it Takes to Make by Daniel Rozin",
                "20. Prototype for Stellar Interloper (Silver Surfer) by Inigo Manglano-Ovalle",
                "21. 284",
                "22. Starbucks - 271",
                "23. Shapeshifting to Transcend Limbo by Lyndi Sales",
                "24. Lathocyte by Andy Vogt",
                "25. South-facing window",
                "26. 106 - Virginia Brown Atrium"};
```

```
        try {
            DroneActivity.client.publish(DroneActivity.topic, message);
            for(String d : destinations){
                System.out.println(d);
                message = new MqttMessage();
                message.setPayload(d.getBytes());
                DroneActivity.client.publish(DroneActivity.topic, message);
            }
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: BasicActivity.this,  text: "Unable to Start Basic Tour! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
}
```

Figure 27: Basic Activity basic function defined

3.2.4. DroneActivity.java

DroneActivity.java contains five strings, three boolean values, a MqttAndroidClient and six main functions. The three of the strings are used to pass the boolean values between the different pages within the application. The other two strings are used to store the client connection address and topic for the client. The six functions are on create, connect, pause, resume, end and disconnect functions. The three strings used to pass boolean values are DRONE_CONNECT, DRONE_START and DRONE_PAUSE. The two strings strings used to store the client information are MQTTHOST and topic. The DRONE_CONNECT string is used to store the connected boolean value. The DRONE_START string is used to store the started boolean value. The DRONE_PAUSE string is used to store the started boolean value. The MQTTHOST string is used to store the connection address for the client, which is a tcp connection to the ip address of the drone's Raspberry Pi 3 B and the port number 1883. The topic string is used to store the topic used for this project, which is "zach/drone/rpi/tour". The

three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application. The classroom function has a string array that has all of the destination that the basic tour will visit.

```
public class DroneActivity extends AppCompatActivity {
    //static String MQTTHOST = "tcp://192.168.1.138:1883";    //home wifi
    //static String MQTTHOST = "tcp://172.20.10.10:1883";     //hotspot
    static String MQTTHOST = "tcp://10.236.3.30:1883";  //TAMU
    public static String topic = "zach/drone/rpi/tour";
    MqttMessage message = new MqttMessage();

    public static final String DRONE_CONNECT = "com.example.graysonvansickle.zachdrone.DRONE_CONNECT";
    public static final String DRONE_START = "com.example.graysonvansickle.zachdrone.DRONE_START";
    public static final String DRONE_PAUSE = "com.example.graysonvansickle.zachdrone.DRONE_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;

    public static MqttAndroidClient client;
```

Figure 28: Drone Activity strings and booleans defined

The on create function starts the drone activity layout xml file and displays this layout to the application. The function then sets the drone item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named DRONE_CONNECT, DRONE_START or DRONE_PAUSE. If the intent has an DRONE_CONNECT extra then the connected boolean value is set to the value of the intent's DRONE_CONNECT extra. If the intent has an DRONE_START extra then the started boolean value is set to the value of the intent's DRONE_START extra. If the intent has an DRONE_PAUSE extra then the paused boolean value is set to the value of the intent's DRONE_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_drone);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 4);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(DRONE_CONNECT))
        connected = getIntent().getBooleanExtra(DRONE_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(DRONE_START))
        started = getIntent().getBooleanExtra(DRONE_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(DRONE_PAUSE))
        paused = getIntent().getBooleanExtra(DRONE_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + DRONE_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 29: Drone Activity on create function defined

If the user selected the Main page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the MainActivity class. The connected boolean value is then added to this intent as an extra named MAIN_CONNECT. The started boolean value is then added to this intent as an extra named MAIN_START. The paused boolean value is then added to this intent as an extra named MAIN_PAUSE. The intent created from the DroneActivity class to the MainActivity class is then started so the application will then run the MainActivity.java file. If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the DroneActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the DroneActivity class to the ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the DroneActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named

BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the DroneActivity class to the BasicActivity class is then started so the application will then run the BasicActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                Intent intent1 = new Intent( packageContext: DroneActivity.this, MainActivity.class);
                intent1.putExtra(MainActivity.MAIN_CONNECT, connected);
                intent1.putExtra(MainActivity.MAIN_START, started);
                intent1.putExtra(MainActivity.MAIN_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent2 = new Intent( packageContext: DroneActivity.this, ArtActivity.class);
                intent2.putExtra(ArtActivity.ART_CONNECT, connected);
                intent2.putExtra(ArtActivity.ART_START, started);
                intent2.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent3 = new Intent( packageContext: DroneActivity.this, ClassroomActivity.class);
                intent3.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent3.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent3.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent4 = new Intent( packageContext: DroneActivity.this, BasicActivity.class);
                intent4.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent4.putExtra(BasicActivity.BASIC_START, started);
                intent4.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent4);
                break;
            case R.id.navigation_drone: //Drone
                break;
        }
        return false;
    }
});
```

Figure 30: Drone Activity switch pages function defined

The connect function first checks to see if the connected boolean value is false. If connected is false then the function will create string for the client id of “Zach Drone Android Device”, create the MqttAndroidClient using the MQTTHOST and client id strings, set the connected boolean value to true, create a new MqttConnectOptions object, and then connect to the client using the options object. The function then checks if the connection was successful or not. If the connection was successful then the function creates a new MqttMessage that says connect, publishes that message to the topic defined earlier, and an alert will pop up at the bottom of the screen to tell the user the connection was successful. If the connection was unsuccessful then an alert will pop up at the bottom of the screen to tell the user the connection failed. If connected is true then an alert will pop up at the bottom of the screen to tell the user they are already connected to the drone.

```
public void connect(View v) {
    if(!connected) {
        String clientId = "Zach Drone Android Device";
        client = new MqttAndroidClient(this.getApplicationContext(), MQTTHOST, clientId);

        Toast.makeText( context: DroneActivity.this,  text: "connect", Toast.LENGTH_LONG).show();

        connected = true;

        MqttConnectOptions options = new MqttConnectOptions();

        try {
            IMqttToken token = client.connect(options);
            token.setActionCallback(new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    MqttMessage message = new MqttMessage();
                    message.setPayload("connect".getBytes());
                    try {
                        client.publish(topic, message);
                    } catch (MqttException e) {
                        e.printStackTrace();
                    }
                    Toast.makeText( context: DroneActivity.this,  text: "Connected!!", Toast.LENGTH_LONG).show();

                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                    Toast.makeText( context: DroneActivity.this,  text: "Connect Failed!!", Toast.LENGTH_LONG).show();
                }
            });
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else
        Toast.makeText( context: DroneActivity.this,  text: "Already Connected to Drone", Toast.LENGTH_LONG).show();
}
```

Figure 31: Drone Activity connect function defined

The pause function first checks to see if the connected and started boolean values are true, and the paused boolean value is false. If connected and started are true and pause is false then the function creates a new MqttMessage that says pause, an alert pops up at the bottom of the screen that says pause to tell the user the tour is paused, and the paused boolean value is set to true. The function then publishes the message to the topic defined earlier. If connected is true and started is false then an alert pops up at the bottom of the screen to tell the user to start a tour before pausing a tour. If connected and started are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and start a tour before pausing a tour.

```
public void pause(View v) {
    if(connected && started && !paused) {
        MqttMessage message = new MqttMessage();
        message.setPayload("pause".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "pause", Toast.LENGTH_LONG).show();

        System.out.println("pause");

        paused = true;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !started) {
        AlertDialog.Builder builder = new AlertDialog.Builder( context: this, R.style.LightDialogTheme);
        builder.setTitle("Did you start a tour?");
        builder.setMessage("You must press the \"Start Tour\" button before pressing \"Pause Tour\" button.");
        builder.setPositiveButton( text: "OK",  listener: null);
        builder.show();
    }
    else if(paused) {
        AlertDialog.Builder builder = new AlertDialog.Builder( context: this, R.style.LightDialogTheme);
        builder.setTitle("Tour already paused");
        builder.setMessage("To resume the tour press the \"Resume Tour\" button.");
        builder.setPositiveButton( text: "OK",  listener: null);
        builder.show();
    }
    else {
        AlertDialog.Builder builder = new AlertDialog.Builder( context: this, R.style.LightDialogTheme);
        builder.setTitle("Did you connect to the drone and start a tour?");
        builder.setMessage("You must press the \"Connect to Drone\" and \"Start Tour\" buttons before pressing the \"Pause Tour\" button.");
        builder.setPositiveButton( text: "OK",  listener: null);
        builder.show();
    }
}
```

Figure 32: Drone Activity pause function defined

The resume function first checks to see if the connected and paused boolean values are true. If connected and paused are true then the function creates a new MqttMessage that says resume, an alert pops up at the bottom of the screen that says resume to tell the user the tour is resumed, and the paused boolean value is set to false. The function then publishes the message to the topic defined earlier. If connected is true and paused is false then an alert pops up at the bottom of the screen to tell the user to pause a tour before resuming a tour. If connected and paused are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and pause a tour before resuming a tour.

```
public void resume(View v) {
    if(connected && paused) {
        MqttMessage message = new MqttMessage();
        message.setPayload("resume".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "resume", Toast.LENGTH_LONG).show();

        paused = false;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !paused)
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Resume Tour! Must Pause Tour first!", Toast.LENGTH_LONG).show();
    else
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Resume Tour! Must Connect to Drone and Pause Tour first!", Toast.LENGTH_LONG).show();
}
```

Figure 33: Drone Activity resume function defined

The end function first checks to see if the connected and started boolean values are true. If connected and started are true then then function creates a new MqttMessage that says end, an alert pops up at the bottom of the screen that says end to tell the user the tour is ended, and the started boolean value is set to false. The function then publishes the message to the topic defined earlier. If connected is true and started is false then an alert pops up at the bottom of the screen to tell the user to start a tour before ending a tour. If connected and started are false then an alert pops up at the bottom of the screen to tell the user to connect to the drone and start a tour before ending a tour.

```
public void end(View v) {
    if(connected && started) {
        MqttMessage message = new MqttMessage();
        message.setPayload("end".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "end", Toast.LENGTH_LONG).show();

        started = false;

        try {
            client.publish(topic, message);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else if(connected && !started)
        Toast.makeText( context: DroneActivity.this,  text: "Unable to End Tour! Must Start a Tour first!", Toast.LENGTH_LONG).show();
    else {
        Toast.makeText( context: DroneActivity.this,  text: "Unable to End Tour! Must Connect to Drone and Start a Tour first!", Toast.LENGTH_LONG).show();
    }
}
```

Figure 34: Drone Activity end function defined

The disconnect function first checks to see if the connected boolean value is true. If connected ist true then the function will create a new MqttMessage and this message is then set to disconnect, an alert will pop up at the bottom of the screen that says disconnect to tell the user that the application was disconnected from the drone, and the connected boolean value is set to false. The function then publishes the message to the topic defined earlier. The function then checks if the disconnection was successful or not. If the disconnection was successful then

an alert will pop up at the bottom of the screen to tell the user the disconnection was successful. If the disconnection was unsuccessful then an alert will pop up at the bottom of the screen to tell the user the disconnection failed. If connected is false then an alert will pop up at the bottom of the screen to tell the user that they must connect to the drone before disconnecting from the drone.

```
public void disconnect(View v) {
    if(connected) {
        MqttMessage message = new MqttMessage();
        message.setPayload("disconnect".getBytes());

        Toast.makeText( context: DroneActivity.this,  text: "disconnect", Toast.LENGTH_LONG).show();

        connected = false;

        try {
            client.publish(topic, message);
            IMqttToken token = client.disconnect();
            token.setActionCallback(new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    Toast.makeText( context: DroneActivity.this,  text: "Disconnected!!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                    Toast.makeText( context: DroneActivity.this,  text: "Disconnect Failed!!", Toast.LENGTH_LONG).show();
                }
            });
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
    else {
        Toast.makeText( context: DroneActivity.this,  text: "Unable to Disconnect from Drone! Must Connect to Drone first!", Toast.LENGTH_LONG).show();
    }
}
```

Figure 35: Drone Activity disconnect function defined

3.2.5. MainActivity.java

MainActivity.java contains three strings, three boolean values, and one main function, on create function. The three strings are used to pass the boolean values between the different pages within the application. The three strings are MAIN_CONNECT, MAIN_START and MAIN_PAUSE. The MAIN_CONNECT string is used to store the connected boolean value. The MAIN_START string is used to store the started boolean value. The MAIN_PAUSE string is used to store the started boolean value. The three boolean values are used to make sure the user can online push certain buttons. The three boolean values are connected, started and paused. The connected value is used to make the user connect to the drone first before doing anything else. The paused value is used to make the user pause a tour first before resuming a tour. The started value is used to make the user start a tour first before pausing, resuming or ending a tour. The on create function has a sub-function in it that deals with the changing the pages within the application.

```
public class MainActivity extends AppCompatActivity {

    public static final String MAIN_CONNECT = "com.example.graysonvansickle.zachdrone.MAIN_CONNECT";
    public static final String MAIN_START = "com.example.graysonvansickle.zachdrone.MAIN_START";
    public static final String MAIN_PAUSE = "com.example.graysonvansickle.zachdrone.MAIN_PAUSE";
    boolean connected = false;
    boolean started = false;
    boolean paused = false;
```

Figure 36: Main Activity strings and booleans defined

The on create function starts the main activity layout xml file and displays this layout to the application. The function then sets the main item in the bottom navigation bar to be the selected item. The function also checks to see if the intent has an extra named

MAIN_CONNECT, MAIN_START or MAIN_PAUSE. If the intent has an MAIN_CONNECT extra then the connected boolean value is set to the value of the intent's MAIN_CONNECT extra. If the intent has an MAIN_START extra then the started boolean value is set to the value of the intent's MAIN_START extra. If the intent has an MAIN_PAUSE extra then the paused boolean value is set to the value of the intent's MAIN_PAUSE extra. If the intent doesn't have any of the extras then the boolean values will stay as false. The sub-function of the on create function takes the user input and then switches the current page of the application to the one the user selected.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    BottomNavigationView bottomNavigationView = (BottomNavigationView) findViewById(R.id.navigation);
    Menu menu = bottomNavigationView.getMenu();
    MenuItem menuItem = menu.getItem( index: 0);
    menuItem.setChecked(true);

    if(getIntent().hasExtra(MAIN_CONNECT))
        connected = getIntent().getBooleanExtra(MAIN_CONNECT, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_CONNECT);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(MAIN_START))
        started = getIntent().getBooleanExtra(MAIN_START, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_START);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    if(getIntent().hasExtra(MAIN_PAUSE))
        paused = getIntent().getBooleanExtra(MAIN_PAUSE, defaultValue: false);
    else {
        try {
            throw new IllegalAccessException("Activity cannot find extras " + MAIN_PAUSE);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 37: Main Activity on create function defined

If the user selected the Art Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the ArtActivity class. The connected boolean value is then added to this intent as an extra named ART_CONNECT. The started boolean value is then added to this intent as an extra named ART_START. The paused boolean value is then added to this intent as an extra named ART_PAUSE. The intent created from the MainActivity class to the ArtActivity class is then started so the application will then run the ArtActivity.java file. If the user selected the Classroom Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the ClassroomActivity class. The connected boolean value is then added to this intent as an extra named CLASSROOM_CONNECT. The started boolean value is then added to this intent as an extra named CLASSROOM_START. The paused boolean value is then added to this intent as an extra named CLASSROOM_PAUSE. The intent created from the MainActivity class to the

ClassroomActivity class is then started so the application will then run the ClassroomActivity.java file. If the user selected the Basic Tour page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the BasicActivity class. The connected boolean value is then added to this intent as an extra named BASIC_CONNECT. The started boolean value is then added to this intent as an extra named BASIC_START. The paused boolean value is then added to this intent as an extra named BASIC_PAUSE. The intent created from the MainActivity class to the BasicActivity class is then started so the application will then run the BasicActivity.java file. If the user selected the Drone page in the bottom navigation bar, then a new intent is made that connects the MainActivity class to the DroneActivity class. The connected boolean value is then added to this intent as an extra named DRONE_CONNECT. The started boolean value is then added to this intent as an extra named DRONE_START. The paused boolean value is then added to this intent as an extra named DRONE_PAUSE. The intent created from the MainActivity class to the DroneActivity class is then started so the application will then run the DroneActivity.java file.

```
bottomNavigationView.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            case R.id.navigation_home:  //Main
                break;
            case R.id.navigation_dashboard: //Art Tour
                Intent intent1 = new Intent( packageContext: MainActivity.this, ArtActivity.class);
                intent1.putExtra(ArtActivity.ART_CONNECT, connected);
                intent1.putExtra(ArtActivity.ART_START, started);
                intent1.putExtra(ArtActivity.ART_PAUSE, paused);
                startActivity(intent1);
                break;
            case R.id.navigation_classroom: //Classroom Tour
                Intent intent2 = new Intent( packageContext: MainActivity.this, ClassroomActivity.class);
                intent2.putExtra(ClassroomActivity.CLASSROOM_CONNECT, connected);
                intent2.putExtra(ClassroomActivity.CLASSROOM_START, started);
                intent2.putExtra(ClassroomActivity.CLASSROOM_PAUSE, paused);
                startActivity(intent2);
                break;
            case R.id.navigation_basic: //Basic Tour
                Intent intent3 = new Intent( packageContext: MainActivity.this, BasicActivity.class);
                intent3.putExtra(BasicActivity.BASIC_CONNECT, connected);
                intent3.putExtra(BasicActivity.BASIC_START, started);
                intent3.putExtra(BasicActivity.BASIC_PAUSE, paused);
                startActivity(intent3);
                break;
            case R.id.navigation_drone: //Drone
                Intent intent4 = new Intent( packageContext: MainActivity.this, DroneActivity.class);
                intent4.putExtra(DroneActivity.DRONE_CONNECT, connected);
                intent4.putExtra(DroneActivity.DRONE_START, started);
                intent4.putExtra(DroneActivity.DRONE_PAUSE, paused);
                startActivity(intent4);
                break;
        }
        return false;
    }
});
```

Figure 38: Main Activity switch pages function defined

3.3. Python Code

The message decoder for the mobile application subsystem is written in python and takes in messages from the Mosquitto server. These messages are then decoded in the python program and outputs an appropriate message that will be later sent to the navigation subsystem.

3.3.1. Client.py

Client.py contains two functions. These two functions are the message decoder and the connection status. The message decoder takes the message received from the client and then

outputs the correct action based on the message received. The connection status function subscribes the client to the correct topic so that the client can see the messages that are published to this topic. The client is created using Eclipse Paho given the client name, which in this case is just Zach Drone Raspberry PI. The client is connected to the defined server address, which is the IP address of the Raspberry Pi on the WiFi. When the client is connected to the server, the program runs the connection status function. When the client receives a message from the server, the program runs the message decoder function. The client runs forever, unless the user connects to the Raspberry Pi and terminates the program. Client.py was used to test the message decoder separately from the other subsystems.

```python
#Execute when a connection has been established to the MQTT server
def connectionStatus(client, userdata, flags, rc):
	#Subscribe client to a topic
	mqttClient.subscribe("zach/drone/rpi/tour")

#Execute when a message has been received from the MQTT server
def messageDecoder(client, userdata, msg):
	#Decode message received from topic
	message = msg.payload.decode(encoding='UTF-8')

	#Print out correct message
	if message == "art":
		print("\n\nArt Tour Selected!")
	elif message == "classroom":
		print("\n\nClassroom Tour Selected!")
	elif message == "basic":
		print("\n\nBasic Tour Selected!")
	elif message == "pause":
		print("\nTour Paused!")
	elif message == "resume":
		print("\nTour Resumed!")
	elif message == "end":
		print("\nTour Ended!")
	elif message == "connect":
		print("\nConnected to Drone!")
	elif message == "disconnect":
		print("\nDisconnected from Drone!")
	else:
		#print("Unknown message!")
		print(message)
```

```python
#Set client name
clientName = "Zach Drone Raspberry PI"

#Set MQTT server address
	#Home wifi
#serverAddress = "192.168.1.138"
	#Hotspot
#serverAddress = "172.20.10.10"
	#Zach Guest
#serverAddress = "10.236.61.140"
	#TAMU
serverAddress = "10.236.3.30"

#Instantiate Eclipse Paho as mqttClient
mqttClient = mqtt.Client(clientName)

#Set calling functions to mqttClient
mqttClient.on_connect = connectionStatus
mqttClient.on_message = messageDecoder

#Connect client to server
mqttClient.connect(serverAddress)

#Monitor client activity forever
mqttClient.loop_forever()
```

Figure 39: Defined functions (left) and variables (right) for the client.py code

3.3.2. Main.py

The client.py functions were added into main.py in the proc3 multiprocessing process. The proc3 process takes in the connected and application multiprocessing variables. The connected multiprocessing variable is used to tell the other processes when the user has pressed the “Connect to Drone” button on the app, which will then start the other processes. The connected variable will either be 1 or 0, 1 if the user is connected with the application and 0 if the user is disconnected with the application. Connected will be set to 1 when the user presses the “Connect to Drone” button and will be set to 0 when the user presses the “Disconnect from Drone” button on the mobile applications. The application multiprocessing variable is used to tell the other processes when the user presses “Start Art Tour”, “Pause Tour”, “Resume Tour”, or “End Tour” buttons. The application variable will either be 5, 1 or -1. Application will be set to 5 when the user presses the “Start Art Tour” button, set to 1 when the user presses the “Pause Tour” button, and set to -1 when the user presses the “End Tour”

button. When the user presses the “Resume Tour” button, the application variable will be set back to 5.

```
#Application
def proc3(application,connected):
    #Execute when a connection has been established to the MQTT server
    def connectionStatus(client, userdata, flags, rc):
        #Subscribe client to a topic
        mqttClient.subscribe("zach/drone/rpi/tour")
    #Execute when a message has been received from the MQTT server
    def messageDecoder(client, userdata, msg):
        #Decode message received from topic
        message = msg.payload.decode(encoding='UTF-8')
        #Print out correct message
        if message == "art":
            application.value = 5
            print("\nArt Tour!")
        elif message == "classroom":
            application.value = 5
            print("\nClassroom Tour!")
        elif message == "basic":
            application.value = 5
            print("\nBasic Tour!")
        elif message == "pause":
            application.value = 1
            print("\nTour Paused!")
        elif message == "resume":
            application.value = 5
            print("\nTour Resumed!")
        elif message == "end":
            application.value = -1
            print("\nTour Ended!")
        elif message == "connect":
            connected.value = 1
            print("\nConnected!")
        elif message == "disconnect":
            connected.value = 0
            print("\nDisconnected!")

    #Set client name
    clientName = "Zach Drone Raspberry PI"
    #Set MQTT server address
    serverAddress = commands.getoutput('hostname -I').split(' ')[0]
    #Instantiate Eclipse Paho as mqttClient
    mqttClient = mqtt.Client(clientName)
    #Set calling functions to mqttClient
    mqttClient.on_connect = connectionStatus
    mqttClient.on_message = messageDecoder
    #Connect client to server
    mqttClient.connect(serverAddress)
    #Monitor client activity forever
    mqttClient.loop_forever()
```

Figure 40: Functions and variables from client.py implemented in the proc3 multiprocessing process

3.4. Mobile Application Layout

The mobile application subsystem is dependent on the layout of the application to make the app easy for the user to interact with. The application takes input from the user and then outputs the correct action to the drone’s microcontroller. The application is written for both Android and iOS platforms.

3.4.1. Android Application Layout

The layout of the Android application is written using the Android Studio application on a Macbook Pro. The application is separated into 5 different pages, which in Android Studio are written as xml files. On the main page there is just the name of the project and the team member’s names. On the art tour, classroom tour and basic tour pages the list of all the destinations for that particular tour are listed in order and there is a button that will start that tour. On the drone page there are buttons to connect to and disconnect from the drone, along with buttons to pause, resume and end the current tour. All the buttons on the pages follow the app logic in figure 2 and will send the correct information to the message decoder on the drone’s Raspberry Pi 3 B.

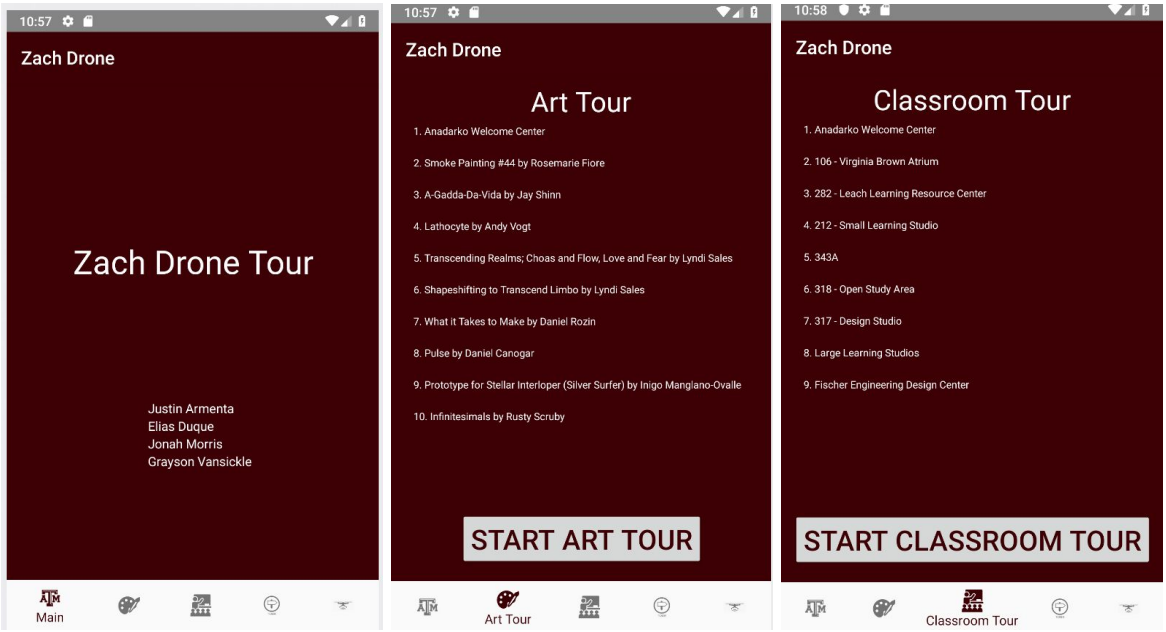
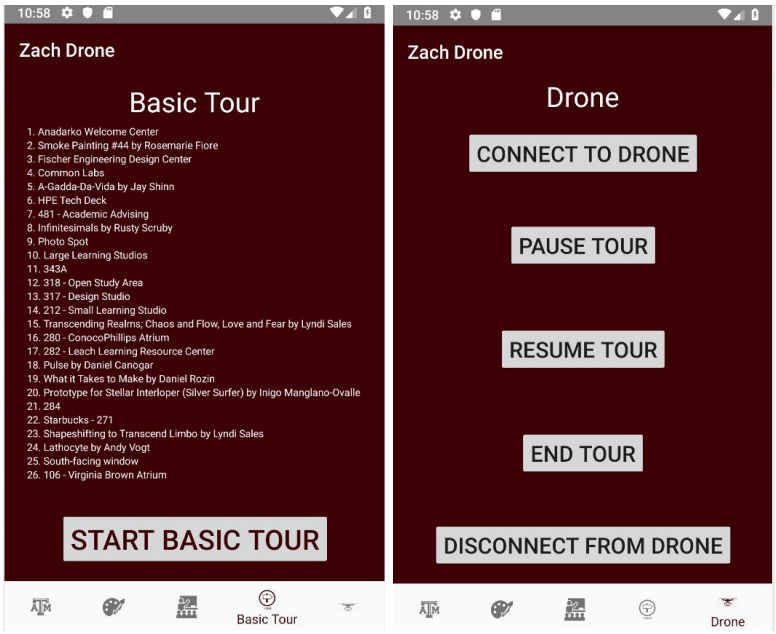


Figure 41: Android Application Layout (Main, Art Tour and Classroom Tour pages above; Basic Tour and Drone pages below)



3.4.2. iOS Application Layout

The layout of the iOS application is written using the Xcode application on a Macbook Pro. The application is separated into 5 different pages, which in Xcode are written one Main.storyboard file and given separate view controllers. On the main page there is just the name of the project and the team member's names. On the art tour, classroom tour and basic tour pages the list of all the destinations for that particular tour are listed in order and there is a button that will start that tour. On the drone page there are buttons to connect to and disconnect from the drone, along with buttons to pause, resume and end the current tour. All the buttons on

the pages follow the app logic in figure 2 and will send the correct information to the message decoder on the drone's Raspberry Pi 3 B.

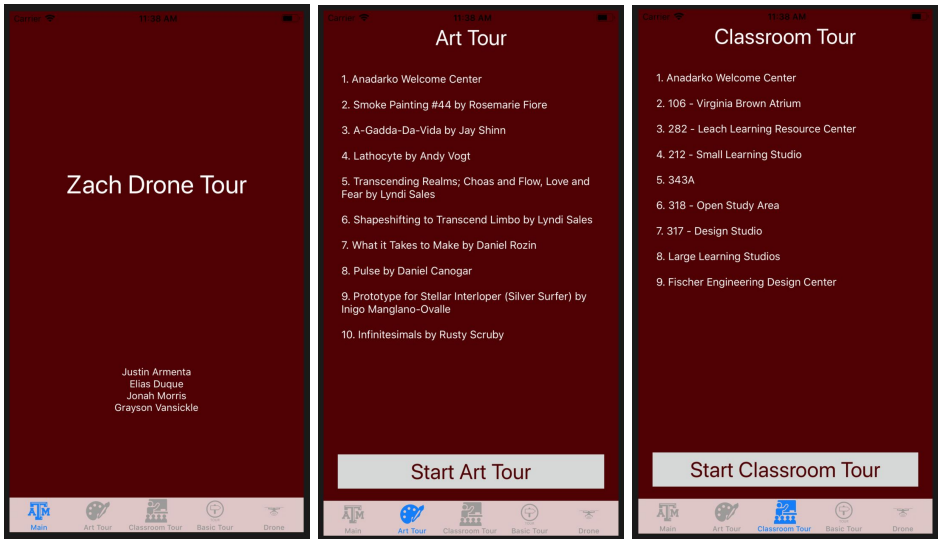
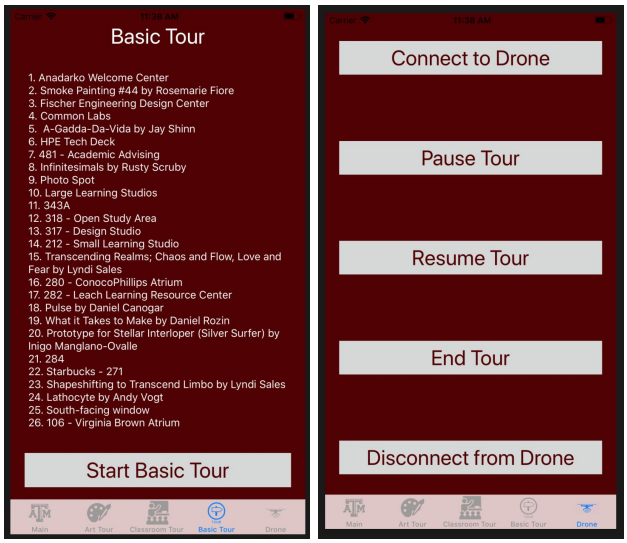


Figure 42: iOS Application Layout (Main, Art Tour and Classroom Tour pages above; Basic Tour and Drone pages below)



4. Operation

In order to show the mobile application subsystem working without connecting it to the other subsystems, statements were printed to the console to show the buttons working as expected. The statements were added to both the iOS and Android applications. A python code file was written to show that the buttons sent the right messages to the drone's Raspberry Pi 3 B. The operation of both the Android and iOS applications at the end of 404 works the same way as they did at the end of 403 since there was only one minor change to make sure the user can only press "Pause Tour" once.

```
Main View Loaded
Art Tour View Loaded
Classroom Tour View Loaded
Basic Tour View Loaded
Drone View Loaded
Need to connect to drone first
Need to connect to drone and start a tour first
Need to connect to drone and pause tour first
Need to connect to drone and start a tour first
Need to connect to drone first
Need to connect to drone first
Need to connect to drone first
10.236.3.30  connect
Need to start a tour first
Need to pause tour first
Need to start a tour first
10.236.3.30  classroom
10.236.3.30  art
10.236.3.30  basic
Need to pause tour first
10.236.3.30  pause
10.236.3.30  end
10.236.3.30  resume
10.236.3.30  disconnect
```

```
I/System.out: connect
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
I/ickle.zachdron: Background concurrent copying GC freed 15597(1581KB) A
I/Choreographer: Skipped 42 frames!  The application may be doing too mu
I/OpenGLRenderer: Davey! duration=721ms; Flags=0, IntendedVsync=48120913
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
W/ActivityThread: handleWindowVisibility: no activity for token android.
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
I/System.out: art
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
W/ActivityThread: handleWindowVisibility: no activity for token android.
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
I/System.out: classroom
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
W/ActivityThread: handleWindowVisibility: no activity for token android.
D/EGL_emulation: eglMakeCurrent: 0xe45522a0: ver 3 0 (tinfo 0xe4503670)
I/System.out: basic
```

Figure 43: Terminal output for iOS (left) and Android (right) applications

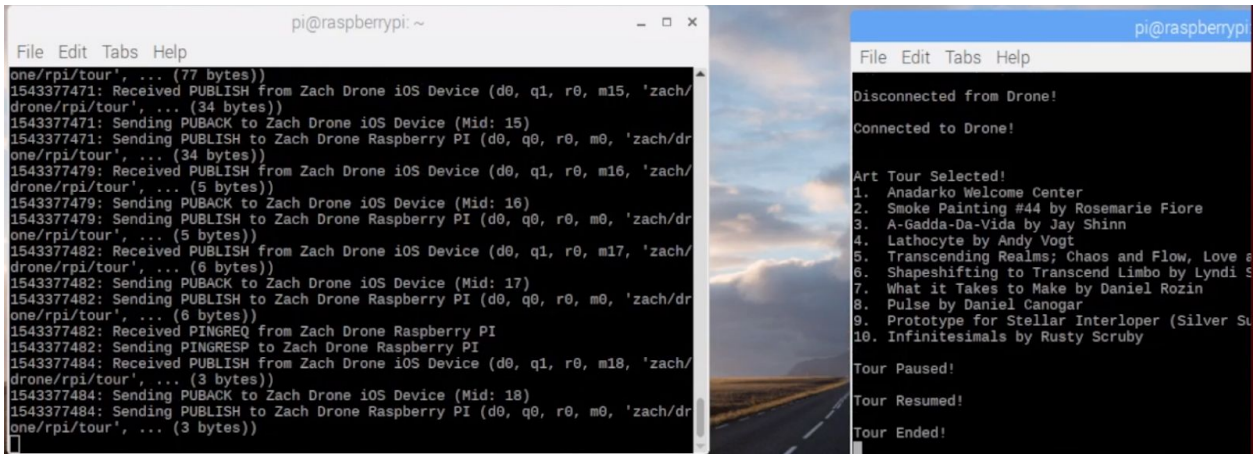


Figure 44: Terminal output for Raspberry Pi 3 B on drone

5. Data Collection

Since this application wasn't connected to any of the other subsystems, there wasn't any actual data since all the application does is send a string to the drone. The string sent to the drone will eventually be passed into the navigation subsystem's function so that the correct tour is started on the drone. However, when sending the destinations to the drone multiple times there was sometimes missing destinations on the drone.

The data collected for the mobile application subsystem is described in the entire system report. The application was able to connect to the drone and the drone reacted correctly to the user's input on the application. The sending of destinations was removed from the application due to bugs when sending all destinations and the fact that the navigation subsystem uses a map model of the building which has all the destinations stored in it.

6. Accomplishments

Throughout the semester in 403, the mobile application subsystem reached many accomplishments. The mobile apps were able to be loaded onto test iOS and Android devices. Using the applications on the devices, the apps were able to connect to the drone's onboard Raspberry Pi 3 B. The applications were also able to send the three different tours to the Pi and send commands to pause, resume and end a tour. The user was able to communicate with the drone and send it messages that will be used to control the drone.

Throughout the semester in 404, the mobile application subsystem didn't see too many updates due to the applications having all the functionality needed at the end of 403. The mobile apps were able to be loaded onto test iOS and Android devices again like in 403. The devices were able to be used to connect to the drone's onboard Raspberry Pi 3 B and control the tour with all the buttons.

7. Future Work

Although the mobile application was able to send messages to the drone's Raspberry Pi 3 B, there are some changes that needed to be made to make the mobile application easier to incorporate with the other subsystems. The application will need to made faster to connect to the drone and send messages to the drone. The message decoder function on the drone's Raspberry Pi needs to be changed to better decode the incoming messages on the drone. The main focus of next semester in ECEN 404 is to combine the mobile application with the other three subsystems of the Zach Drone project, the subsystem that will be most closely related to the mobile application is the navigation subsystem.

The speed at which the mobile applications' messages were detected and used in the other subsystems increased greatly due to all subsystems running in multiple processes and communicating through multiprocessing variables. The connection speed of the applications was increased to due to changing to using our personal Wi-Fi hotspots as opposed to the tamulink Wifi which caused connection issues. The only downside to using Wifi was that the mobile application and Raspberry Pi must be connected to the same Wifi connections and the applications needed to know the Raspberry Pi's IP address to be able to connect. In the future, this would be the major area of improvement to the applications to make it so the applications and drone's Raspberry Pi could be on separate Wifi networks and that the application wouldn't have to be changed if the Raspberry Pi's IP address changed.

8. Conclusion

The iOS and Android applications performed very well and sent the messages to the drone quicker than I expected. The applications performed all of the functions it was supposed to and sent the correct messages to the drone. When the application is combined with the drone, there might need to be slight changes to the way the messages are sent or the message decoder to allow the drone to work efficiently. Looking forward to the next semester there is still

work that needs to be done to make this a complete working project, but the mobile application subsystem was a great success for the amount of time in this semester. The task at hand for next semester isn't an easy one, but completing the mobile application takes one thing off the list that needs to be finished next semester.

The iOS and Android applications performed just as expected and were able to send messages to the drone in a fast manner to avoid any lag in controlling the drone to react to these messages. Using multiprocessing variables meant that the message decoder could constantly run on the Raspberry Pi while the other processes were running and not take away processing power from the more computationally heavy processes.

Zach Drone

Justin Armenta

Navigation Subsystem Report

REVISION – 2.0
28 April 2019

Table of Contents

List of Figures

List of Tables

No tables in this document.

1. Introduction

The purpose of the navigation system is to determine the drone's location within Zachry and navigate to the next destination based on this information. This subsystem will consist of three ultrasonic sensors that will be placed on the front, left, and right of the drone, as well as a Raspberry Pi 3 B to run the Python code, which takes the distance information from the sensors to navigate. Based on the current and previous values of the ultrasonic sensors the Raspberry Pi will determine if a new checkpoint has been reached using an algorithm that compares the sensor values to the map model or if it was a false reading and should be ignored. Based on the determined location, a command for distance and direction will be given to the flight controller. A model of Zachry has been written into the Raspberry Pi that gives the dimensions of all hallways that the drone will fly near. This will help the drone determine if it is at a new checkpoint or if a sensor value has made an error and should be ignored.

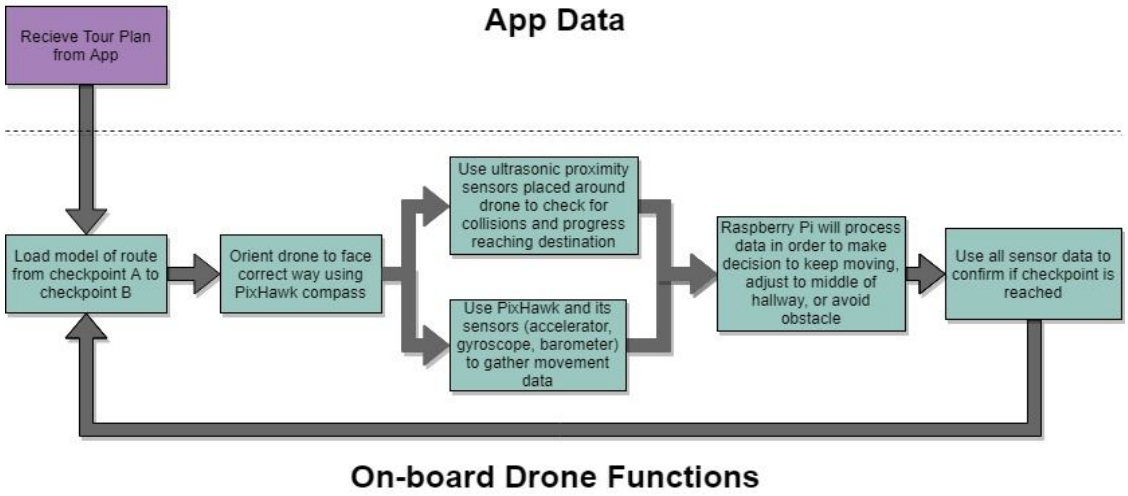


Figure 1: Navigation Subsystem

2. Theory

2.1. Tracking Location of Drone

The main first and second floor hallways within Zachry have been inserted into the Raspberry Pi with different areas called "checkpoints" that make up the hallway. A "checkpoint" is an area with unique dimensions within the hallway, such as the area before a divot for a doorway appears on the right side of the wall. The key thing to note about every checkpoint is that there is a significant change to a particular side of a wall. Checkpoints were created as a point where ultrasonic sensors can see a large enough change that a flag can be thrown which starts the process of determining if this change in value was a sensor misreading or a new checkpoint being reached. Dimensions on each checkpoint were manually entered in the Raspberry Pi such as: the change in the left and right wall compared to the previous checkpoint, the distance from the beginning of the destination to the beginning of the checkpoint, the ceiling height, and wall width at the beginning of the checkpoint. All of this data is used in a checkpoint

algorithm to determine which checkpoint the drone is located in. The Raspberry Pi will check to see if there is greater than 0.45 m change in the ultrasonic sensors compared to their last readings. If a large difference is detected the rest of the algorithm is ran to determine the checkpoint the drone is in. Since the ultrasonic sensors can only be relied on up to about 4.5 m away and some hallways are much wider than 9 m, every destination has a “side reliance” which basically means the drone will stay near one side of the hallway and rely on the ultrasonic sensor on the reliant side more than the other. The algorithm is shown below in figure 2.

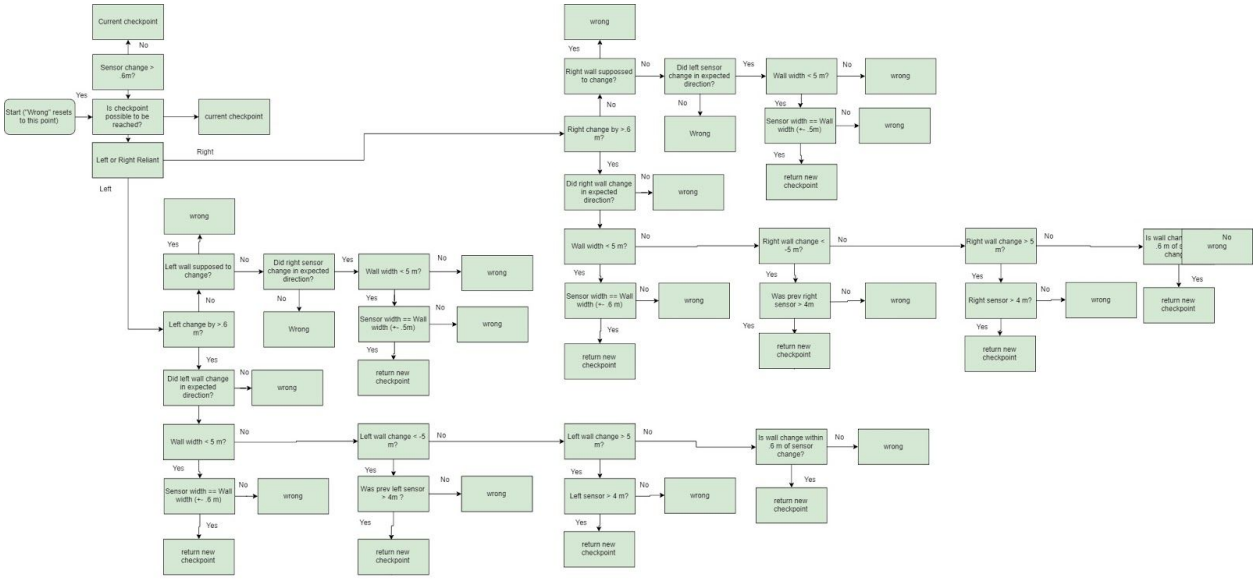


Figure 2: Checkpoint Algorithm

Once the last checkpoint has been reached within a destination, the “attraction” has been reached, which means the drone will give a speech about the attraction. At this point the drone will orient itself and load a new destination with all new checkpoints. When the last destination is reached this will signal the end of the tour.

2.2. Outputting Print Commands to Pixhawk

Unfortunately due to the drone’s unstable nature, indoor tests weren’t able to be performed. Instead we outputted print commands to the command window to show what commands would be given to the flight controller if the drone was safe to fly indoors. The print commands were statements such as “Move forward at 0.5 m/s for 15 meters”, and the person holding the drone would follow the commands exactly. The different commands used were “Move forward x meters”, “Drift left/right x meters”, “Adjust yaw angle x degrees”, “Stop Drone”, “Land drone”, “Change ground speed to x m/s” and “Take off”.

3. Design

3.1. Python Codes

The navigation subsystem is entirely in python code and takes ultrasonic inputs and outputs speeds to the flight controller. The three main programs that will be used are the Map_Model.py, Checkpoint_Checker.py, and Main.py.

3.1.1. Map_Model.py

Map_Model.py contains only three actual functions. They all are very trivial functions that simply iterate to the next checkpoint or destination. The rest of the code is stating the dimensions of the building so Main.py can call on that information later when the navigation function needs it to determine the current checkpoint. Dictionaries were used to create the attributes of the tour. The tour choice will be populated first between the art tour, class tour, or the basic tour. Then within a specific tour, all destinations within that tour's dictionary will be declared. Then all the checkpoints within the destination will be declared. Shown below in figure 3 is the different dictionaries being declared.

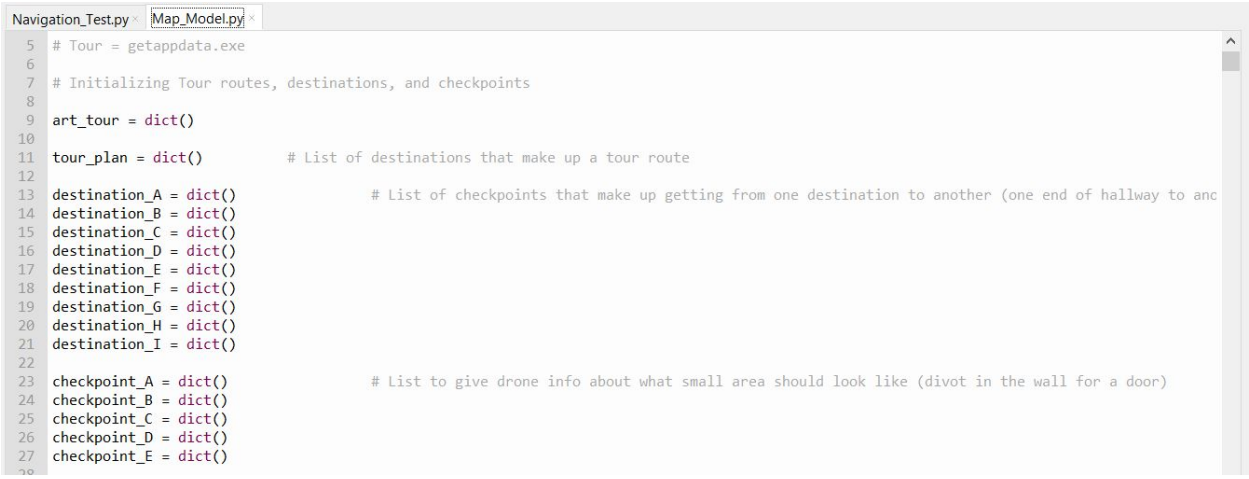


Figure 3: Declaring Dictionaries

In figure 4 the different attributes that go inside checkpoint and destination are shown. As you can see checkpoints are put inside unique destinations. Together any attribute of a checkpoint can be found as long as you have the tour, the destination, and the checkpoint. All values were in meters.

```
checkpoint_E['Floor'] = 0.0
checkpoint_E['Ceiling'] = 3.048
checkpoint_E['Desired Altitude'] = 2.438
checkpoint_E['Left wall change'] = 50
checkpoint_E['Right wall change'] = 7
checkpoint_E['Wall width'] = 7.493
checkpoint_E['Z Position'] = 11.938
checkpoint_E['Max Distance'] = 20               # Change in distance from previous right wall to current right wall
checkpoint_E['Slope'] = 0


destination_A['Compass Change'] = 0         # Compass direction drone should face before beginning flight path (in degrees)
destination_A['Orientation'] = 'Left'               # Will be used to tell drone which side sensor will have priority over the other when
destination_A['Speech'] = 'None'
destination_A['Checkpoint A'] = checkpoint_A
destination_A['Checkpoint B'] = checkpoint_B
destination_A['Checkpoint C'] = checkpoint_C
destination_A['Checkpoint D'] = checkpoint_D
destination_A['Checkpoint E'] = checkpoint_E
destination_A['Checkpoint F'] = 'End'
destination_A['Hallway Length'] = 15
```

Figure 4: Attributes of destinations and checkpoints

3.1.2. Main.py

Main.py is where the important functions associated to finding the location, determining the distance and directions, and adjusting from one destination to the next is found. Main.py consists of two while loops that run. The first is a while loop that runs until the end of the tour is reached, at which point the drone lands and the tour ends. Inside this while loop is where the Raspberry Pi will give a command to the flight controller and it will take off and head towards the next destination. The second while loop is inside the first while loop and checks if a new checkpoint has been reached, it needs to avoid an obstacle, it needs to adjust its angle, it needs to change its speed due to the tour group falling behind or getting too close, or it has to pause or stop the tour due to the tour group pausing or stopping the tour on their mobile application. Once the drone has taken off, the sensors gather data approximately every 0.3 seconds. At that time the checkpoint algorithm is used, using the function named “checkpoint_checker” inside “Checkpoint_Checker.py”, to determine which checkpoint the drone is at. If a new checkpoint is found then the “current checkpoint” variable will become this new checkpoint, given that the new checkpoint is within two of the current one. The drone will also change it’s “guessed z position” variable to the value stated in the new checkpoint’s z position. This “guessed z position” also changes based on the z speed to keep track of the likely z position of the drone, unless a new checkpoint is found then the z position will be changed to the value stored inside the new checkpoint dictionary. This guessed z position variable is important when determining whether it is actually possible for the drone to be at the next checkpoint.

3.1.3. Checkpoint_Checker.py

“Checkpoint_Checker.py” is where the algorithm shown in figure 2 is ran. The only reason it has its own file is because it is such a long function that it would take up too many lines within “Main.py” and make it even harder to read through the code. Checkpoint_Checker

takes the previous and current ultrasonic sensor values, current checkpoint, destination, and tour in order to figure out which checkpoint the drone is currently at. It then outputs this checkpoint.

4. Operation

In order to show the navigation subsystem working without flying the actual drone a pseudo flight involving at least two people had to be used. One person held the drone above their head while the other help a laptop that showed the command window of the Raspberry Pi. The commands were read to the person holding the drone and they would follow them as closely as possible. The figure below shows an example of the command window during the flight. This figure was a test that included the camera and app subsystems so there is a few unnecessary print statements shown. But you can still see the operation of the navigation system commanding the drone to perform different actions as well as showing significant wall changes to confirm that the checkpoint_checker function is working correctly.

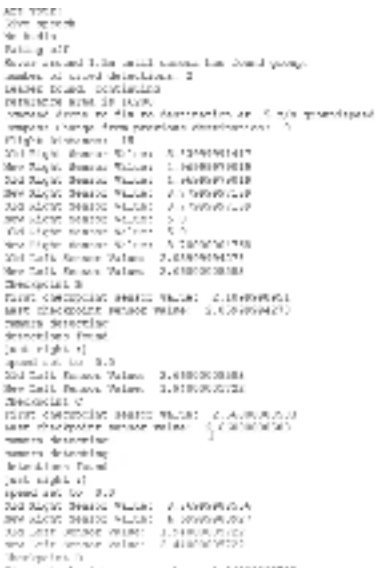


Figure 5: Example of command window during test flight

As the drone reached the end of a destination it would land to give a speech, which saved battery and made the audio easier to hear, then take off turn to face the correct direction and adjust to the middle of the hallway if necessary and then begin flying to the next destination. Shown below is a figure of what the command window looked like as it reached the end of a destination.

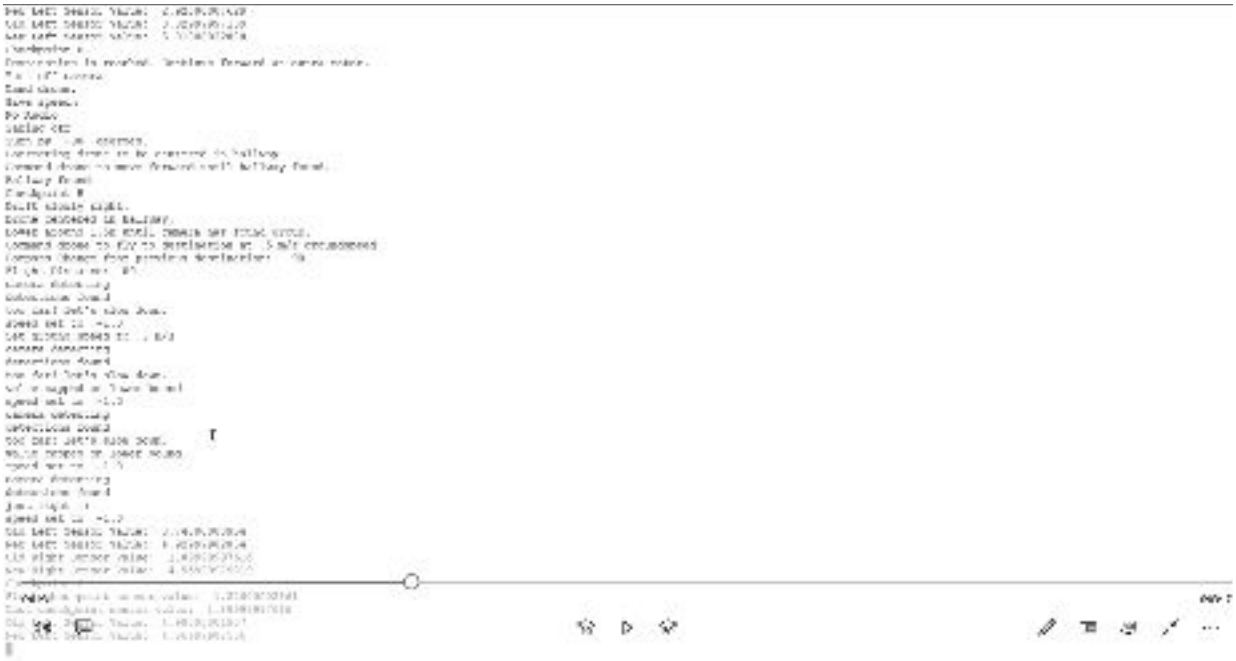


Figure 6: Data from transition from destination A to destination B

The drone also accounts for certain worst case scenarios and can correct based on what the ultrasonics are reading and where the drone thinks it is. If the drone senses an object is too close and there might be a collision, it will stop and check the sensor again to make sure it wasn't a false value then drift away a short distance and continue its normal flight. Also, if the drone drifts too far away from a wall it can stop and drift back to the wall so it doesn't get inaccurate readings because it is too far away. This can be seen in the figure below. This command window was shown because the drone was slowly moved to the left (too close to the wall) then after it had successfully corrected itself, it was moved to the right (too far away from the wall).

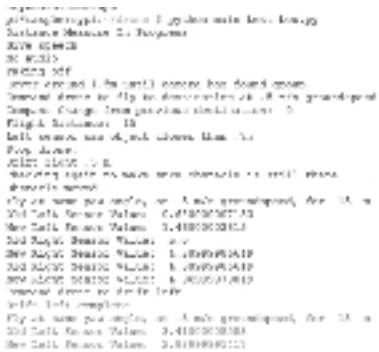


Figure 7: Command window for collision avoidance

For certain spots during the flight it can also correct the location of the drone and the angle based on what which sensor was reading a potential collision. The drone can also adjust for small angle errors by reading the very first value read at a checkpoint and compare that to

the last value received. If there is a great enough change that wasn't supposed to occur it does some basic trigonometry to adjust the angle. Both cases can be seen in the figure below for a specific case during the drone flight.

Figure 8: Command Window for angle correction

5. Data Collection

The checkpoint_checker function was the most vital part of the navigation subsystem and required many tweaks to the algorithm to make sure it minimized false positives and find the changes for real checkpoints. While running just the sensors constantly and walking the route, we found out that the sensor would consistently read all wall changes .05 to .15 meters less than what they should be. It was concluded that the ultrasonics were always getting a small reflection from the closer wall which would make the sensors think that the farther wall wasn't as far away as it really was. So Map_Model.py had to be updated to reflect those differences between the ideal case and reality. During test runs we also outputted the change in sensor values when a greater than 0.4 meter change was read which helped validate the sensors and checkpoint function to make sure they were working correctly together. Below is a figure of what one run of the drone outputted when it was going through destination B.

Figure 9: Command Window for destination B

As you can see on the command window above the checkpoint changed to "I" when a change of 1.25 was found. For the change to "J" a change of -2.44 was found, for "K" a 2.5

change was found, for “L” a -1.03 change was found. When we compare this to the map model figure shown below we see that all those values are very close to the exact values we were expecting.

```
checkpoint_BI['Right wall change'] = 1.25
checkpoint_BI['Wall width'] = 26.476
checkpoint_BI['Z Position'] = 38.33085
checkpoint_BI['Max Distance'] = 20
checkpoint_BI['Slope'] = 0


checkpoint_BJ['Floor'] = 0.0
checkpoint_BJ['Ceiling'] = 3.048
checkpoint_BJ['Desired Altitude'] = 2.438
checkpoint_BJ['Left wall change'] = 0
checkpoint_BJ['Right wall change'] = -2.4
checkpoint_BJ['Wall width'] = 22
checkpoint_BJ['Z Position'] = 41.31535
checkpoint_BJ['Max Distance'] = 20
checkpoint_BJ['Slope'] = 0

checkpoint_BK['Floor'] = 0.0
checkpoint_BK['Ceiling'] = 3.048
checkpoint_BK['Desired Altitude'] = 2.438
checkpoint_BK['Left wall change'] = 0
checkpoint_BK['Right wall change'] = 2.45
checkpoint_BK['Wall width'] = 25.2
checkpoint_BK['Z Position'] = 42.12815
checkpoint_BK['Max Distance'] = 40
checkpoint_BK['Slope'] = 0

checkpoint_BL['Floor'] = 0.0
checkpoint_BL['Ceiling'] = 3.048
checkpoint_BL['Desired Altitude'] = 2.438
checkpoint_BL['Left wall change'] = 0
checkpoint_BL['Right wall change'] = -1.175
checkpoint_BL['Wall width'] = 22
checkpoint_BL['Z Position'] = 45.112
checkpoint_BL['Max Distance'] = 40
```

Figure 10: Map Model for destination B

6. Accomplishments

Over the course of the semester the navigations subsystem had many accomplishments. The code was able to take only values from the app, camera, and the outputs from the sensors and give correct commands to the flight controller. Many pseudo flights were ran using the method talked about above to show that the drone knew where it was and that it should do next. The map model was updated to values closer to what the ultrasonics thought the change was. A simulation on a computer of the walls and fake sensors was changed to work in real life with the actual building and real sensor values. And we were able to communicate with the app, camera, and ultrasonic sensors and make many decisions based on what they were telling my navigation system.

7. Future Work

Although the simulation worked there may need to be changes to fix some missed checkpoints in the future since there were occasionally erratic sensor values that weren’t

accurate. A secondary check to verify the location could be a good addition. This could be done using the BLE beacons already in place in Zachry. Unfortunately there wasn't enough time to add this to the project. Only the first two floors were done since creating a model of all of Zachry is very time consuming and doesn't really add much to the proof of concept of this project. There was also no good way to transition between floors since we never flew the drone inside. Although trying to take the stairs seemed like a feasible option since the drone had an ultrasonic sensor facing downwards so it could maintain altitude.

8. Conclusion

The code for navigating the drone performed very well and exceeded my own expectations. It did everything it was supposed to do in terms of figuring out where it is within the building and outputting the commands based on that information. Although the drone never actually flew indoors I believe that if we had a very stable drone with great sensors (accelerometer, gyroscope, etc) it could feasibly do it. It was very hard to test for every possible thing that could go wrong so there's a chance some errors weren't accounted for in the code. But I believe many very helpful and intuitive safety features were added that made this project a realistic one if given more time and a better drone. Flying a drone autonomously with no GPS is a massive task in itself, and asking it to also lead a crowd, and get to each destination in a timely manner is massive undertaking but the whole group did a great job of showing that it can be done!

Zach Drone

Jonah Morris

Drone Subsystem

REVISION – 2.0
28 April 2019

Table of Contents

List of Figures

List of Tables

1. Introduction

The Zach drone tour will need a functioning drone to implement a tour. The drone subsystem consists of all the of the hardware aspects of this project. This subsystem includes the drone frame, battery, power converter, motors, electric speed converters, propellers, Pixhawk, PX4FLOW, Raspberry Pi, and ultrasonic sensors.

2. Theory

The scope of this subsystem is to provide all of the hardware aspects of the project needed by the other subsystems. There are essentially two subsystems of this subsystem, the ultrasonic sensors and the drone.

2.1. Drone

Our drone uses a battery that is connected to a power distributor that supplies the motor ESCs, Pixhawk and Raspberry Pi with power. When powered on the Raspberry Pi controls the Pixhawk using the MAVlink protocol over serial connection giving it the direction and speed to move. The Pixhawk will then tell the electronic speed converters (ESC) how much thrust to give the motors.

2.2. Ultrasonic Sensors

There are three ultrasonic sensors connected to the Raspberry Pi via GPIO pins. These sensors send out ultrasonic sound and measure the time it takes to receive the sound back. It then converts that time into a high voltage pulse that is high for 1 µs per 1 mm to the nearest detectable object is. This data is used by the navigation subsystem to determine where the drone is inside of the building based on a model.

3. Design

3.1. Power Connections

A single four cell Lipo battery will power the entire drone and all microcontrollers. The drone uses a TATTU 8000mAh lipo battery plugged into a PDB-XT60 power regulator through the Pixhawk's own power converter connection. The motor electronic speed converters are powered from the ESC tabs and the Raspberry Pi is powered from the 5V tab on the PDB-XT60. The three ultrasonic sensors are powered from the Raspberry Pi's 3.3V pin. Voltage and current flow is shown in the following figure.

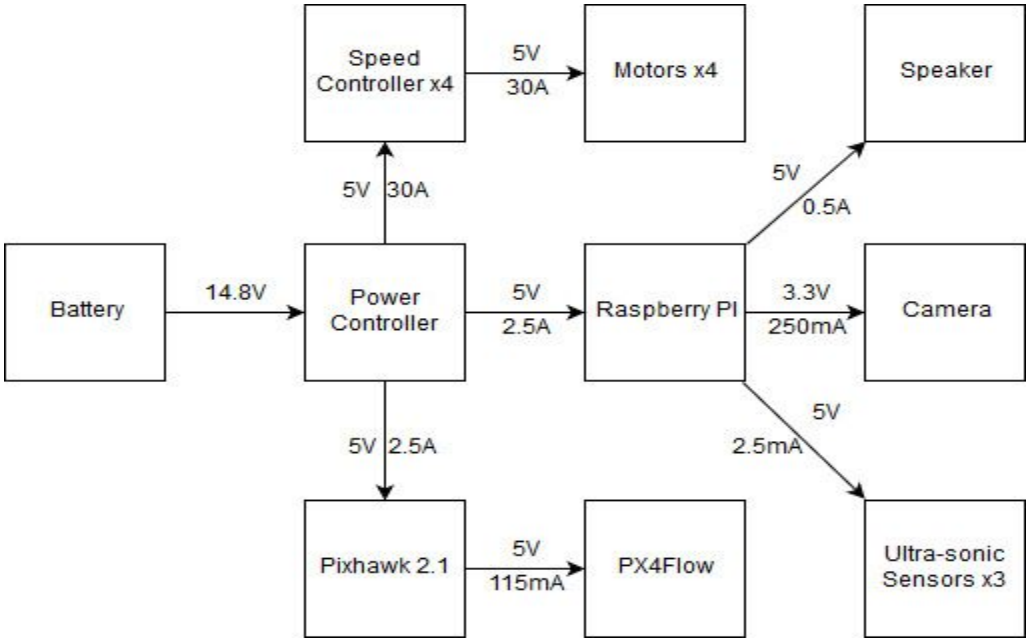


Fig. 1: Power connections of the drone subsystem with maximum voltage and current ratings

3.2. Ultrasonic Sensors

Ultrasonic sensors are used to detect the drones distance from nearby objects. The drone uses six Maxbotix HRLV-EZ0 ultrasonic rangefinders connected to the Raspberry Pi to determine the distance to the nearest object. These sensors are rated for distances of 30 cm to 5m. The Raspberry Pi uses these sensors in a trigger then receive relationship allowing for readings to only be requested when they are needed by the navigation subsystem. Ultrasonics sensor were chosen over infrared sensors because many walls inside of Zachry are glass which does not reflect infrared light well. The following figure displays the connections made between the Raspberry Pi and ultrasonic sensors.

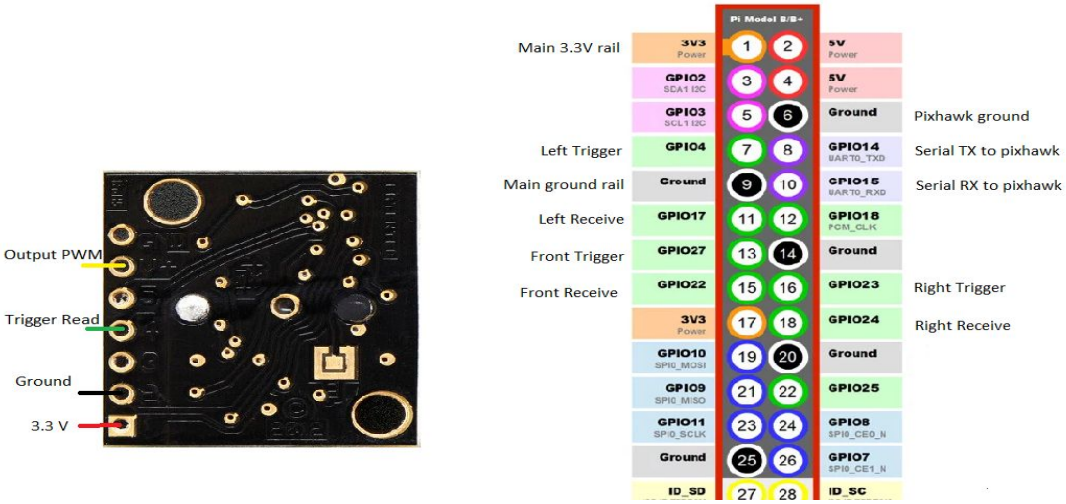


Fig. 2: Connections between raspberry pi GPIO pins and ultrasonic sensors.

The Raspberry Pi gathers ranging data by sending a 25us pulse to the trigger pin and waits for a PWM signal from the output pin. This signal is stepped down using a resistor divider to match the 3.3V GPIO input voltage. The signal output is 1us long per 1 mm distance sensed.

Sensors are error detected through a double check based algorithm. Through observation, we determined that when the sensors get a value that was incorrect it only happened once. Whenever the sensors receive a value that is more than 0.4 cm different than the previous value we immediately check another value to see if it is within 5 cm before updating the distance.

3.3. Flight Time

We would like our flight time of the drone to be as long as possible to allow for longer tours. To calculate our estimated time of flight we used the following formula:

$$Flight\ time\ (t) = \frac{Battery\ Capacity * Discharge}{Weight * Light\ Power} * Voltage$$

The combined weight of all components is 2.2kg. The calculated estimated time of flight is 22.3 minutes. We limit the time of tours to be 15 minutes of flight time so that the drone has time to fly back to the starting position before it's battery life fails.

3.4. Raspberry Pi to Pixhawk Connection

The Raspberry Pi tells the Pixhawk which direction to fly and how fast whereas the Pixhawk sends information to the Raspberry Pi about the orientation of the drone and its altitude. This is done by using a serial connection as show in the following figure.

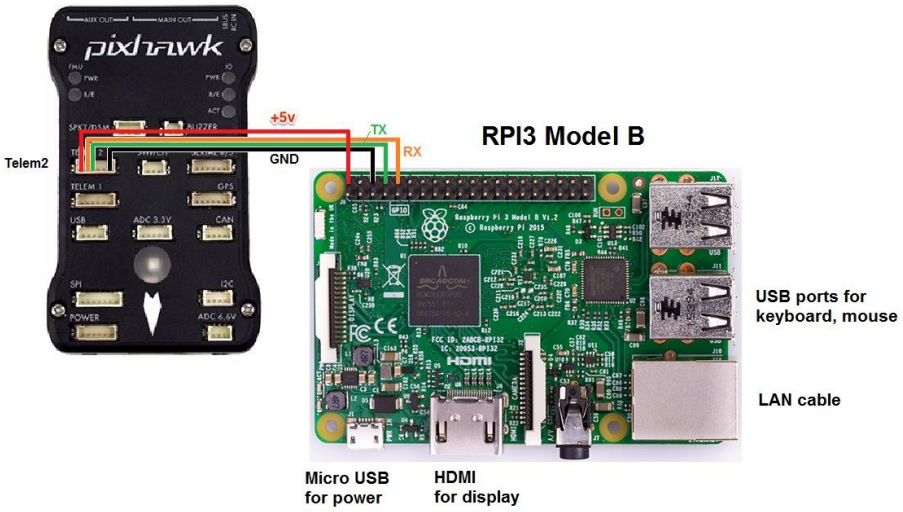


Fig. 3: Pixhawk to Raspberry Pi serial connection

This connection in conjunction with Dronekit and GUIDED flight mode will be used give directional commands to the drone for takeoff and flight.

3.5. 3D Printed Parts

3D printed legs and blade guards were designed using solidworks and printed using the high fidelity printers in the FEDC for this project. Because we used a drone from a previous project, we needed longer legs to created more room for the sensors on the bottom and blade guards for safety. There are two iterations of the blade guards, for clockwise and counterclockwise propellers. Below are schematics for the 3D printed parts as well as a picture of the final drone product.

Fig. 4: 3D printed blade guards and leg schematic

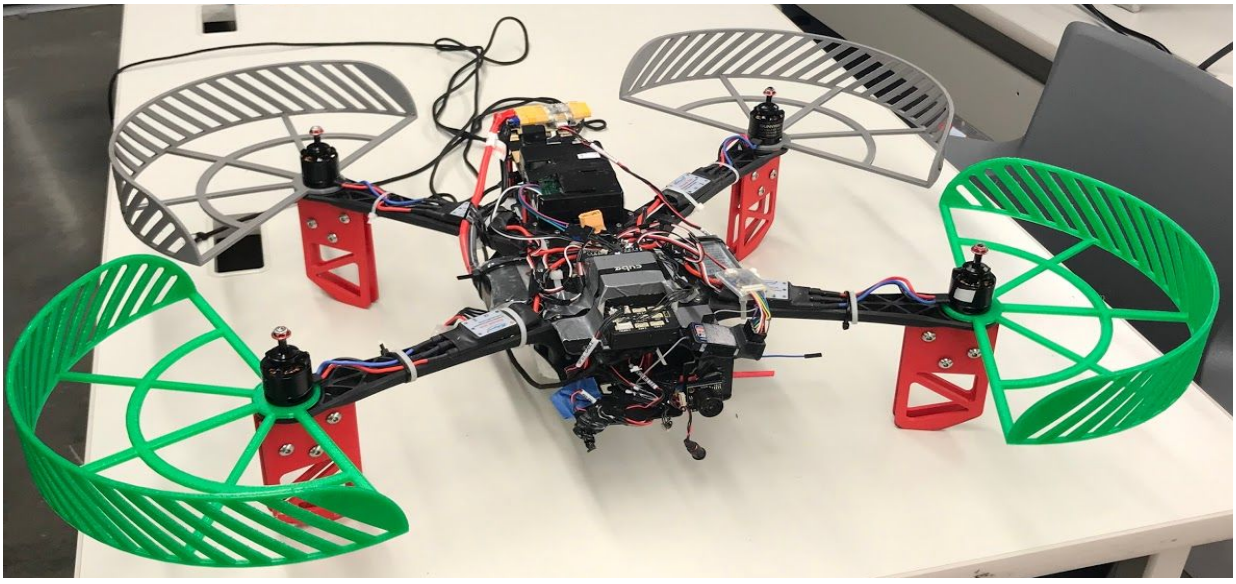


Fig. 5: Final drone layout

4. Operation

4.1. Ultrasonic Sensors

The ultrasonic sensors were validated by first connecting one sensor to the Raspberry Pi and placing an object at incremental distances from the sensor and reading the outputs. The sensor proved to be within 5 centimeters of the expected value for up to 4.3 meters. Next, the three sensors were attached to the drone and connected to the Raspberry Pi. They were tested by placing the drone in the center of a room and placing objects around the drone and testing the Raspberry Pi output to the actual distance.

Further testing of the ultrasonic sensors was conducted along side the navigation subsystem during building runs. Values that the sensors were receiving were printed to the command prompt to be analyzed after each run.

4.2. Power connections

The power connections were tested by attaching all components to the drone and turning them all on using only the battery. The functionality of the Raspberry Pi to Pixhawk connection and ultrasonic sensors were tested again while the battery powered the system.

4.3. Raspberry Pi to Pixhawk Connection

The connection between the Raspberry Pi and Pixhawk was validated by sending flight commands to the Pixhawk through the Raspberry Pi terminal. Changing flight modes and arming the throttle were both tested and validated.

5. Data Collection

5.1. Ultrasonic Sensors

To characterize typical accuracy of the ultrasonic sensors, I tested one sensor by placing it at the end of an empty table and placing a small box in front of it and taking many distance reading at different distances. Below is an excel sheet showing the average distance sensed, the width of the beam at that distance and the standard deviation.

	A	B	C	D	E	F	G	H	I	J
1	distance:	26.4 cm	62.98 cm	92.44 cm	122.96 cm	153.4 cm	183.88 cm	214.36 cm	305.8 cm	427.72 cm
2	values:	30.04	65.37	94.91	125.6	155.71	185.2	215.34	305.22	424.86
3		30.23	64.68	94.39	125.38	156.66	185.97	215.63	305.1	424
4		30.02	64.99	94.7	124.28	156.24	185.68	215.39	304.17	425.15
5		29.85	64.83	93.77	125.81	155.9	185.7	215.27	305.68	424.81
6		30.21	65.64	94.1	125.48	156.12	185.92	216.01	305.27	424.03
7		30.28	65.04	94.46	125.17	156.16	186.13	215.63	305.39	425.05
8		30.23	65.3	94.84	124.98	156.26	185.99	215.7	306.2	424.93
9		29.64	65.09	95.39	125.12	156.09	185.51	215.01	306.22	424.89
10		29.83	64.28	94.87	125.74	156.16	186.3	215.53	305.65	424.55
11		30.21	65.02	94.51	124.91	155.14	185.97	215.17	305.53	424.7
12		30.35	64.95	94.37	125.53	156	185.25	216.03	305.22	425.15
13		29.73	64.8	95.08	125.43	155.69	186.23	214.82	305.68	424.7
14		29.78	65.26	94.91	125.41	156.04	185.94	215.53	305.49	424.86
15		30.35	64.44	94.75	125.43	156.02	185.54	214.91	305.13	425.08
16		29.8	64.9	94.03	125.48	155.78	186.42	215.79	304.94	424.41
17		29.95	64.75	94.7	125.31	155.54	185.51	216.03	305.94	427.87
18		29.54	65.47	94.63	125.79	155.71	185.31	215.79	305.99	424.05
19		29.87	64.68	93.98	125.36	155.93	185.94	215.77	305.94	424.31
20		30.23	64.64	94.22	124.96	155.59	185.97	215.67	305.77	424.84
21		30.72	65.14	94.37	125.48	155.04	185.39	215.58	305.53	425.1
22		30.04	65.04	94.99	125.69	155.76	186.13	215.55	304.84	424.58
23	averages:	30.04286	64.96714	94.57	125.3495	155.8829	185.8095	215.531	305.4714	424.8533
24	beamwidth:		35.56	53.34	50.8	53.34	60.96	38.1	30.48	20.32
25	STD	0.278348	0.326892	0.397851	0.348486	0.363477	0.347953	0.338511	0.47896	0.761998

Table 1: Characteristics of the Maxbotix HRLV-EZO ultrasonic sensors.

Included in the data is the measured distance using a tape measure, the values output from the ultrasonic sensor 20 times, the averages, the beamwidth and the standard deviation. This data helped characterize the sensors and gave the navigation subsystem more guidelines as to how accurate its distance reading are going to be. Below is a chart that characterizes the difference between measures and actual as well as the standard deviation with increasing distances.

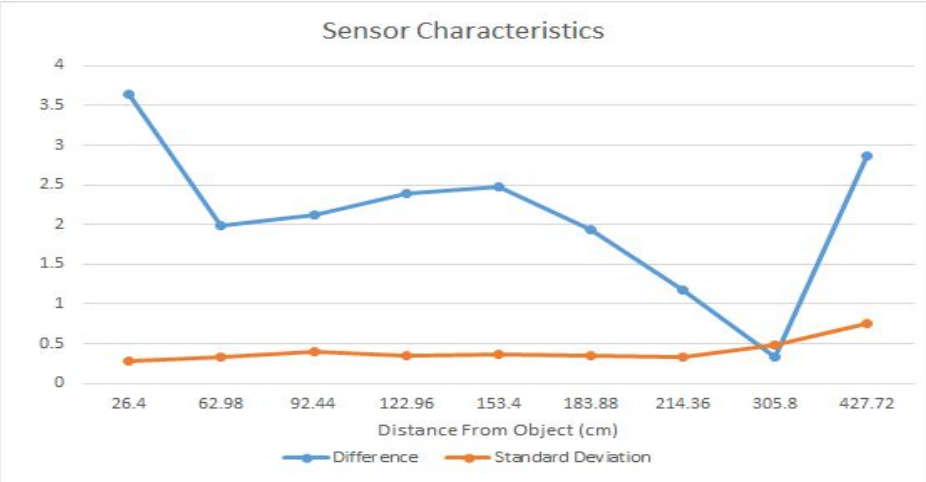


Fig. 6: Actual distance minus measured distance and standard deviation vs. distance from sensor

After attaching and connecting all three ultrasonic sensors I ran 8 tests with objects at different distances compared them to expected results. The following table contains the list of the tests, the results and the results were within 5 cm accuracy.

Measure Distances	Distance Results	Within 5cm?
L = 120cm R = 120cm F = 120cm	L = 118.4cm R = 120.5cm F = 119.4cm	Yes
L = 180cm R = 180cm F = 180cm	L = 178.7cm R = 181.3cm F = 180.9cm	Yes
L = 250cm R = 250cm F = 250cm	L = 250.1cm R = 247.6cm F = 250.2cm	Yes
L = 400cm R = 400cm F = 400cm	L = 404.4cm R = 402.1cm F = 401.6cm	Yes
L = 60cm R = 60cm F = 150cm	L = 59.3cm R = 60.3cm F = 58.8cm	Yes
L = 20cm R = 20cm F = 20cm	L = 30.1cm R = 30.4cm F = 30.2cm	No, Range must be greater than 30cm
L = 200cm R = 250cm F = 120cm	L = 199.2cm R = 251.1cm F = 117.3cm	Yes
L = 90cm R = 90cm F = 60cm	L = 87.4cm R = 88.6cm F = 58.7cm	Yes

Table 2: This table shows the rangefinding data for three sensors placed on the drone.

Finally, numerous test runs were performed with the navigation subsystem without sensor errors. This validates that the system works as intended.

6. Future Work

For full functionality of indoor flight without GPS, a smaller, more stable drone is needed. No GPS flight was tested outside and is talked about in the system report.

7. Conclusion

I created an array of ultrasonic sensors that were error detected and corrected to the point that the system is reliable enough for navigation inside. I designed 3D printed parts for the drone that met the needs of the project. Finally, I aided the navigation subsystem in detecting errors and key cases that were addressed.

Creating this subsystem put into perspective the amount of work that goes into designing, testing and documenting a full project. Through working on this subsystem I learned the values of planning ahead and being thorough with research and design so that implementation goes smoothly. I also learned about the challenges of working as a team and integrating elaborate subsystems. I feel like my subsystem and this project were overall successful.

Zach Drone

Justin Armenta
Elias Duque
Jonah Morris
Grayson Vansickle

Zach Drone System

REVISION – 2.0
28 April 2019

Table of Contents

List of Figures

List of Tables

1. Overview

The Zachry Drone Tour is a tour designed to guide guests around the building to show them the different attractions Zachry has to offer. The goals of this tour are to have different tours that the guests can choose from, keep track of the guests so none falls behind, and give information to the guests about the attractions. The drone has four main subsystems which include: Navigation, Tour Guiding, Drone, and Mobile Application. Each subsystem is necessary to make sure all goals are met.

2. Development Plan and Execution

In order to combine the App, Navigation, Drone (specifically ultrasonics), and Tour Group systems, multiprocessing variables were used so they could communicate with each other. All four subsystems run at the same time on the Raspberry Pi. The multiprocessing variables were used so while each code was running, it could check the status of the other systems using any relevant multiprocessing variables.

2.1. Design Plan

The Navigation subsystem communicates with all other subsystems running on the Raspberry Pi in order to make flight decisions and give commands to the flight controller. Its multiprocessing variables are: “need_camera” which is used to tell the camera whether it should be on or off, “destination” which lets the ultrasonics know which ultrasonic needs to be checked more than the others. The App communicates with all subsystems as well. Its variables are: “connected” which tells if someone is connected to the application so the other subsystems should start, and “application” which tells navigation whether the drone is in the start, pause, or end status. The camera communicates with the navigation and ultrasonic code in order to let them know if people are detected. The variables used are: “tour_group” which tells the navigation the speed the drone should fly at, “group_ready” to let navigation know that the tour leader is found and the drone can continue flying, and “redetect camera” to pause the ultrasonics since they would get inaccurate values in 10 second increments when the redetection code was running. The drone subsystem code was to run the ultrasonics constantly and gave values for the most recent ultrasonic values and if the sensors had an error. These values are: “distance_right” which gave the most recent right ultrasonic value, “distance_left” to give the most recent left value, “distance_front” which gave the most recent front value, and “sensor_error” which let navigation know if there was a problem with the sensors.

2.2. Execution Plan

At the beginning of the second semester the table below was created to show what we hoped to accomplish on different dates. Our initial goals this semester were to find a good way for the subsystems to communicate with each other and to get the drone flying. The first goal involved tweaking our individual subsystems so it would use the multiprocessing variables to both send and receive important information. After that it was just a matter of making sure they

worked well together and that each subsystem would be able to work now that it was all running together. Then near the end of the semester, we worked on worst case scenarios such as the sensors getting bad values/not working at all, camera having problems finding the leader, and the drone getting too close to walls. After all of this was complete were were able to finish our execution plan and begin validation.

Task	Validation	Completion Date
Combine App & Navigation	App used to start tour navigation	January 24th, 2019
No GPS Dronekit Flight	Drone able to takeoff and land	January 31st, 2019
Tracking Loss Contingency	Tracker reset after tracking loss	February 7th, 2019
Integrate PX4Flow with Pixhawk	Have Drone move short distance	February 7th, 2019
Tour Destination Audio Recorded	Able to play tour audio on PI	February 14th, 2019
3D Printing Order Placed	Confirmed order placed	February 14th, 2019
Camera Integrated	Read and manipulate camera	February 14th, 2019
Tour Tracker Completed	Runs off camera and holds tracking	February 21st, 2019
Fully Integrated Tour Tracker with Navigation	Drone flies tour and reacts to crowd	February 28th, 2019
Sensors Fully Integrated with Navigation	Drone says when new checkpoint reached	February 28th, 2019
Tour Destination Audio Fully Integrated with Navigation	Have drone give speech during test runs	February 28th, 2019
Drone Flight of First Floor Hallway	Drone flies through first floor	March 7th, 2019
Second Floor Hallway added to Tour	Drone flies through desired locations	March 14th, 2019
All subsystems integrated together	First full tour given without errors	March 21st, 2019
Drone Completes Tour on one Battery Charge	Drone completes full tour without low battery warning	March 28th, 2019
Drone Completes Tour without Collisions	Drone completes full tour without collisions	April 4th, 2019
Drone Completes Tour by Staying with Group	Drone completes full tour without losing tour group	April 11th, 2019
Project Validated	Drone tour completed	April 15th, 2019

Table 1 : Schedule of Execution and Testing.

2.3. Validation Plan

In order to validate the navigation, ultrasonics, camera, and app all working together we need to have print commands tell us what the Raspberry Pi was doing and also what it thought was happening. This was because we could not actually get the drone to fly indoors so the only way to “fly” was to have a person move it around and for them to follow print commands sent from the command window. Validation of all subsystems working together and correctly was performed by one person acting as the tour leader for the camera, someone connecting their phone with the app to connect and start the tour, one person holding the drone and moving it around, and another person reading out the flight commands to the person with the drone. The command window and the drone with the tour leader were both recorded to show that the command window was giving appropriate outputs based on what was actually happening.

3. Critical System Data

3.1. Ultrasonic Sensors Data

The primary concern we had with using ultrasonic sensors was that they were not going to be reliable or accurate enough for navigation purposes. The below graph shows the difference between actual distance and measures average distance of the ultrasonic sensors at different distances from an object.

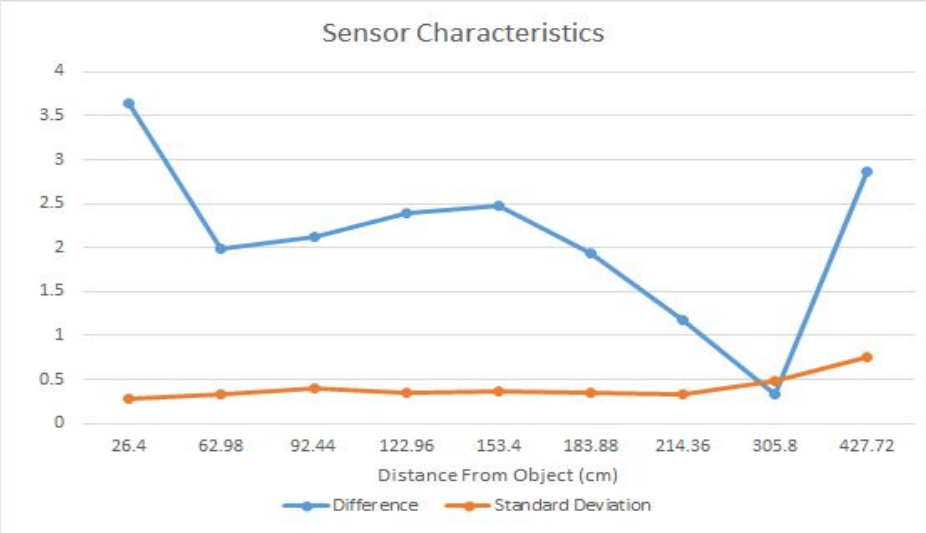


Figure 1: Sensor Characteristics, accuracy and precision

This data shows that between 0.3 m and 4.3 m we have an accuracy closer than 4 cm and standard deviation less than 1 cm. With this data we were able to conclude that these

sensors were well within that accuracy of 10 cm we estimated we would need at the beginning of the project. Below is the table that the above figure is based off of.

	A	B	C	D	E	F	G	H	I	J
1	distance:	26.4 cm	62.98 cm	92.44 cm	122.96 cm	153.4 cm	183.88 cm	214.36 cm	305.8 cm	427.72 cm
2	values:	30.04	65.37	94.91	125.6	155.71	185.2	215.34	305.22	424.86
3		30.23	64.68	94.39	125.38	156.66	185.97	215.63	305.1	424
4		30.02	64.99	94.7	124.28	156.24	185.68	215.39	304.17	425.15
5		29.85	64.83	93.77	125.81	155.9	185.7	215.27	305.68	424.81
6		30.21	65.64	94.1	125.48	156.12	185.92	216.01	305.27	424.03
7		30.28	65.04	94.46	125.17	156.16	186.13	215.63	305.39	425.05
8		30.23	65.3	94.84	124.98	156.26	185.99	215.7	306.2	424.93
9		29.64	65.09	95.39	125.12	156.09	185.51	215.01	306.22	424.89
10		29.83	64.28	94.87	125.74	156.16	186.3	215.53	305.65	424.55
11		30.21	65.02	94.51	124.91	155.14	185.97	215.17	305.53	424.7
12		30.35	64.95	94.37	125.53	156	185.25	216.03	305.22	425.15
13		29.73	64.8	95.08	125.43	155.69	186.23	214.82	305.68	424.7
14		29.78	65.26	94.91	125.41	156.04	185.94	215.53	305.49	424.86
15		30.35	64.44	94.75	125.43	156.02	185.54	214.91	305.13	425.08
16		29.8	64.9	94.03	125.48	155.78	186.42	215.79	304.94	424.41
17		29.95	64.75	94.7	125.31	155.54	185.51	216.03	305.94	427.87
18		29.54	65.47	94.63	125.79	155.71	185.31	215.79	305.99	424.05
19		29.87	64.68	93.98	125.36	155.93	185.94	215.77	305.94	424.31
20		30.23	64.64	94.22	124.96	155.59	185.97	215.67	305.77	424.84
21		30.72	65.14	94.37	125.48	155.04	185.39	215.58	305.53	425.1
22		30.04	65.04	94.99	125.69	155.76	186.13	215.55	304.84	424.58
23	averages:	30.04286	64.96714	94.57	125.3495	155.8829	185.8095	215.531	305.4714	424.8533
24	beamwidth:		35.56	53.34	50.8	53.34	60.96	38.1	30.48	20.32
25	STD	0.278348	0.326892	0.397851	0.348486	0.363477	0.347953	0.338511	0.47896	0.761998

Table 2: Sensor readings at different distances

3.2. Tour Group Tracker Data

The tour group tracker subsystem uses image processing and as such confirmation of logic was mostly visually confirmed. Quantifiable data necessary for the logic has been included here. Numbers were used for the threshold values used in condition checks for the various possible cases.

The data below shows the the value associated with the how confident the histogram of ordered gradients is that the detection it found is an actual person. These values were used to find the absolute thresholds beyond which detection is impossible, as well as possible thresholds for speed changes. As can be seen, closer than 9 feet results in too little of the crowd leader being visible and is therefore the absolute minimum threshold. Further than 24 feet means the crowd leader is too far away to be detected due to too little information to read, and is therefore the absolute maximum threshold.

Distance	**Confidence**
< 9 ft	N/A
9 ft	0.876

12 ft	1.347
15 ft	2.158
18 ft	2.747
21 ft	3.362
24 ft	3.492
> 24 ft	N/A

Table 3: Distance Ratios

The data set below details the confidence values for various detections, both positive and false positive. This data was used for finding the average confidence for good reading as well as the maximum value for incorrect detections in order to filter them out. The minimum viable confidence value for detections was set at 1.4 and used to filter out false positives.

+Desc.	**Confidence Values**	**Confidence Values**	**-Desc.**
(Positives)	**Positive**	**False Positive**	**(False Positives)**
Forward	3.04	0.22	Reflection: Person
Forward	2.83	0.87	Reflection: Person
Forward	3.17	0.56	Reflection: Person
Forward, arms out	1.89	1.16	Trash Bags
Forward, hand to face	2.46	0.47	Random
Forward	1.83	0.52	Reflection: Random
Forward	2.43	0.67	Extra
Forward	2.91	0.65	Reflection: Random
Holding Laptop	1.7	0.77	Extra
Forward	3.74	0.78	Extra
Holding Laptop	1.82		
Holding Jacket	2.89		
Holding Laptop	2.18		
Phone, Extra?	1.42		

Hands in pockets	3.09		
Blurry	3		
Side View	1.85		
Forward	3.08		
Hands on Hips	2.97		
Scratching Face	2.38		
Hands in front	3.27		
Profile View	2.73		
Forward	3.49		
Average	2.616086957	0.678333333	**Average**
Min	1.42		**Min**
Max		1.16	**Max**
Median	2.83	0.56	**Median**

Table 4: Confidence Values for Positives and False Positives

The data set below details the ratios of the areas between the original starting point, 15 feet, and various distances. This data was used to determine the ratios at which a certain amount of relative distance was achieved. As such, should a boundary box be smaller or larger than a certain threshold then the crowd leader is now either too far or too close. Gathered from this set and implemented were these values; 0.65 was used as the threshold for too far (slow down), and 1.4 was used as the threshold for too close (speed up).

Reference Area = 15051 px^2, at 15 feet

Distance (ft)	**Avg. Ratio to Reference**
15 ft (second pass)	1.1
20 ft	0.727
22 ft	0.656
24 ft	0.61
18 ft	0.83
12 ft	1.41

Table 5: 404 Distance Ratios

Further validation for the tour group tracker subsystem was done visually and therefore has no quantification.

3.3. Mobile Application Data

The mobile applications for both Android and iOS were able to connect to the Raspberry Pi using the Mosquitto server. The message decoder running on the Raspberry Pi was successfully integrated using multiprocessing variables to get the options the user picked on the app to the navigation subsystem. The figures below show the user's input on the app being sent to the drone's Raspberry Pi and working with the navigation running using the multiprocessing variables from the message decoder process.

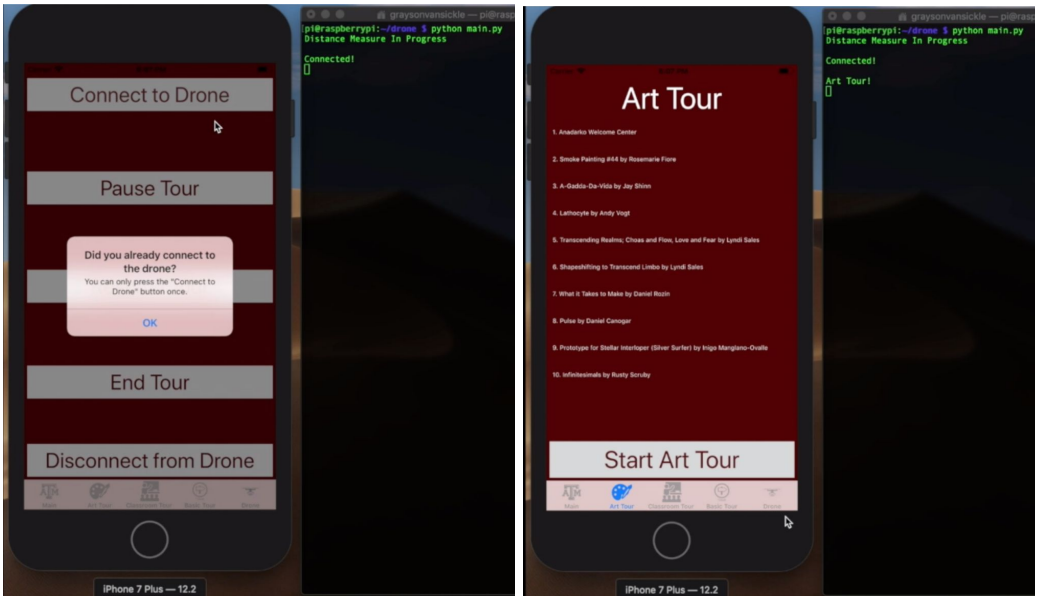


Figure 2: Shows the responses when the connect button (Left) and the start button (Right) are pressed.

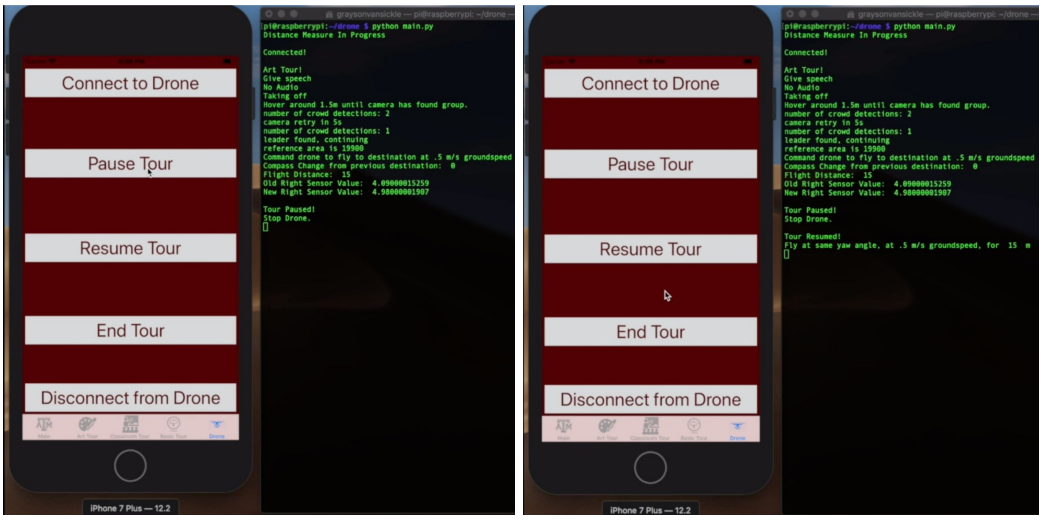


Figure 3: Shows the responses when the pause button (Left) and the resume button (Right) are pressed.

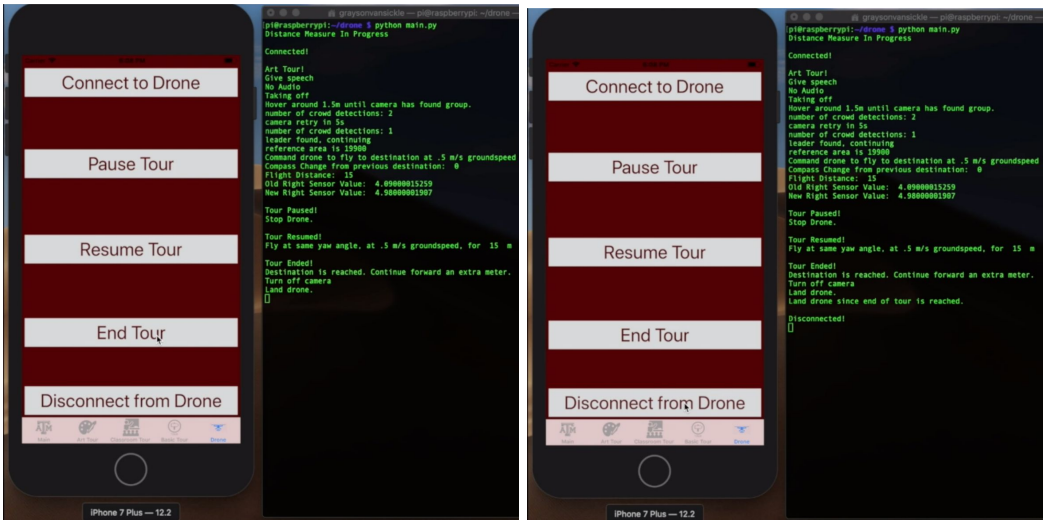


Figure 4: Shows the responses when the end button (Left) and the disconnect button (Right) are pressed.

3.4. Navigation Data

The drone successfully used information from the app which is shown in the figure below. As the app connected and started the Art Tour, the drone began taking off and beginning the tour. This also shows the drone taking information from the camera to know that the tour leader was found so the drone could continue its flight.

Figure 5: Mobile App and Tour Group Tracker Working with Navigation.

The drone was also able to change its speed based on the camera's information during the flight. As shown in the figure below when the tour leader fell too far behind the camera told the drone to slow down and then told it to maintain its current speed later when the leader was the correct distance.

Figure 6: Tour Group Tracker Working with Navigation.

Below is a figure of what one run of the drone outputted when it was going through destination B. As you can see the right sensor values are being shown right before new checkpoints are found.

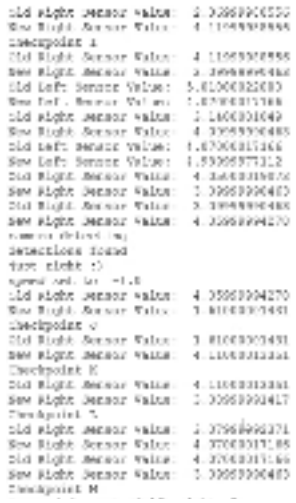


Figure 7: Command Window for destination B

As you can see on the command window above, the checkpoint changed to “I” when a change of 1.25 was found. For the change to “J” a change of -2.44 was found, for “K” a 2.5 change was found, for “L” a -1.03 change was found. When we compare this to the map model figure shown below we see that all those values are very close to the exact values we were expecting.

```
checkpoint_BI['Right wall change'] = 1.25
checkpoint_BI['Wall width'] = 26.476
checkpoint_BI['Z Position'] = 38.33085
checkpoint_BI['Max Distance'] = 20
checkpoint_BI['Slope'] = 0


checkpoint_BJ['Floor'] = 0.0
checkpoint_BJ['Ceiling'] = 3.048
checkpoint_BJ['Desired Altitude'] = 2.438
checkpoint_BJ['Left wall change'] = 0
checkpoint_BJ['Right wall change'] = -2.4
checkpoint_BJ['Wall width'] = 22
checkpoint_BJ['Z Position'] = 41.31535
checkpoint_BJ['Max Distance'] = 20
checkpoint_BJ['Slope'] = 0

checkpoint_BK['Floor'] = 0.0
checkpoint_BK['Ceiling'] = 3.048
checkpoint_BK['Desired Altitude'] = 2.438
checkpoint_BK['Left wall change'] = 0
checkpoint_BK['Right wall change'] = 2.45
checkpoint_BK['Wall width'] = 25.2
checkpoint_BK['Z Position'] = 42.12815
checkpoint_BK['Max Distance'] = 40
checkpoint_BK['Slope'] = 0

checkpoint_BL['Floor'] = 0.0
checkpoint_BL['Ceiling'] = 3.048
checkpoint_BL['Desired Altitude'] = 2.438
checkpoint_BL['Left wall change'] = 0
checkpoint_BL['Right wall change'] = -1.175
checkpoint_BL['Wall width'] = 22
checkpoint_BL['Z Position'] = 45.112
checkpoint_BL['Max Distance'] = 40
```

Figure 8: Map Model for destination B

3.5. Autonomous Drone Flight Data

The flying of the drone autonomously was mostly successful. Due to a rescope of the project that required flights be performed outdoors, the functionality of the drone flights that needed to be shown were: take off, land, move in all directions, and turn in all directions. To complete these tasks, the scripts used were written with DroneKit for Python and was connected to the Pixhawk via the Raspberry Pi's serial connection. Communication between DroneKit and the Pixhawk over serial was achieved using Mavlink commands. The functions used for takeoff and movements were adapted for our use without GPS from the official DroneKit-Python GitHub repository. The functions used are shown in the figures below along with photos of the drone flying using these functions which were taken from the video submitted on eCampus.

```python
def arm_and_takeoff_nogps(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude without GPS data.
    """

    ##### CONSTANTS #####
    DEFAULT_TAKEOFF_THRUST = 0.54

    print("Basic pre-arm checks")
    # Don't let the user try to arm until autopilot is ready
    # If you need to disable the arming check,
    # just comment it with your own responsibility.
    while not vehicle.mode != 'INITALISING' and vehicle._ekf_predposhorizabs:
        print(" Waiting for vehicle to initialise...")
        time.sleep(1)

    lower = vehicle.location.global_relative_frame.alt + (aTargetAltitude*0.95)
    upper = vehicle.location.global_relative_frame.alt + (aTargetAltitude*1.25)
    max_alt = vehicle.location.global_relative_frame.alt + (aTargetAltitude*2)

    print("Arming motors")
    # Copter should arm in GUIDED mode
    vehicle.mode = VehicleMode("GUIDED")
    vehicle.armed = True

    while not vehicle.armed:
        print(" Waiting for arming...")
        vehicle.armed = True
        time.sleep(1)

    print("Taking off!")

    thrust = DEFAULT_TAKEOFF_THRUST
    while True:
        #current_altitude = vehicle.rangefinder.distance    #rangefinder (used for indoor flights)
        current_altitude = vehicle.location.global_relative_frame.alt    #barometer (used for outdoor flights)
        print(" Altitude: %s  Desired: %f" %
              (current_altitude, lower))
    if (current_altitude >= lower and current_altitude <= upper) or current_altitude >= max_alt:
        print("Reached target altitude")
            break
        set_attitude(thrust = thrust)
        time.sleep(0.2)
```

Figure 9: Takeoff function to take off to a desired altitude (window of 95% and 125% of target altitude).

```python
def send_attitude(roll_angle = 0.0, pitch_angle = 0.0,
                  yaw_angle = None, yaw_rate = 0.0, use_yaw_rate = False,
                  thrust = 0.54, duration = 0.0):
    """
    use_yaw_rate: the yaw can be controlled using yaw_angle OR yaw_rate.
                  When one is used, the other is ignored by Ardupilot.
    thrust: 0 <= thrust <= 1, as a fraction of maximum vertical thrust.
            Note that as of Copter 3.5, thrust = 0.5 triggers a special case in
            the code for maintaining current altitude.
    """
    if yaw_angle is None:
        # this value may be unused by the vehicle, depending on use_yaw_rate
        yaw_angle = vehicle.attitude.yaw
    # Thrust >  0.5: Ascend
    # Thrust == 0.5: Hold the altitude
    # Thrust <  0.5: Descend
    msg = vehicle.message_factory.set_attitude_target_encode(
        0, # time_boot_ms
        0, # Target system
        0, # Target component
        0b00000000 if use_yaw_rate else 0b00000100,
        to_quaternion(roll_angle, pitch_angle, yaw_angle), # Quaternion
        0, # Body roll rate in radian
        0, # Body pitch rate in radian
        math.radians(yaw_rate), # Body yaw rate in radian/second
        thrust  # Thrust
    )
    vehicle.send_mavlink(msg)

    if duration != 0:
        modf = math.modf(duration)

        time.sleep(modf[0])

        for x in range(0, int(modf[1])):
            time.sleep(1)
            vehicle.send_mavlink(msg)
```

Figure 10: Send attitude function to control the drone's roll, pitch and yaw angles along with the throttle.

```
print("vehicle connected**********************************************************************************************************")
time.sleep(2)

# Take off 2.5m in GUIDED_NOGPS mode.
arm_and_takeoff_nogps(1.75)

time.sleep(1)

# Move the drone forward and backward.
# Note that it will be in front of original position due to inertia.
print("Move forward")
send_attitude(pitch_angle = -5, thrust = 0.54, duration = 5)

print("Move backward")
send_attitude(pitch_angle = 8, thrust = 0.54, duration = 5)

print("Move left")
send_attitude(roll_angle = -5, thrust = 0.54, duration = 5)

print("Move right")
send_attitude(roll_angle = 5, thrust = 0.54, duration = 5)

print("Turn right")
send_attitude(yaw_angle = 90, thrust = 0.54, duration = 3)

print("Turn left")
send_attitude(yaw_angle = -90, thrust = 0.54, duration = 3)

print("Turn around")
send_attitude(yaw_angle = 180, thrust = 0.54, duration = 3)

print("Setting LAND mode...")
vehicle.mode = VehicleMode("LAND")
while not vehicle.mode == "LAND":
    vehicle.mode = VehicleMode("LAND")
    time.sleep(1)
time.sleep(2)

# Close vehicle object before exiting script
print("Close vehicle object")
vehicle.close()

print("Completed")
```

Figure 11: Shows the calls to the functions described above used for validating results.

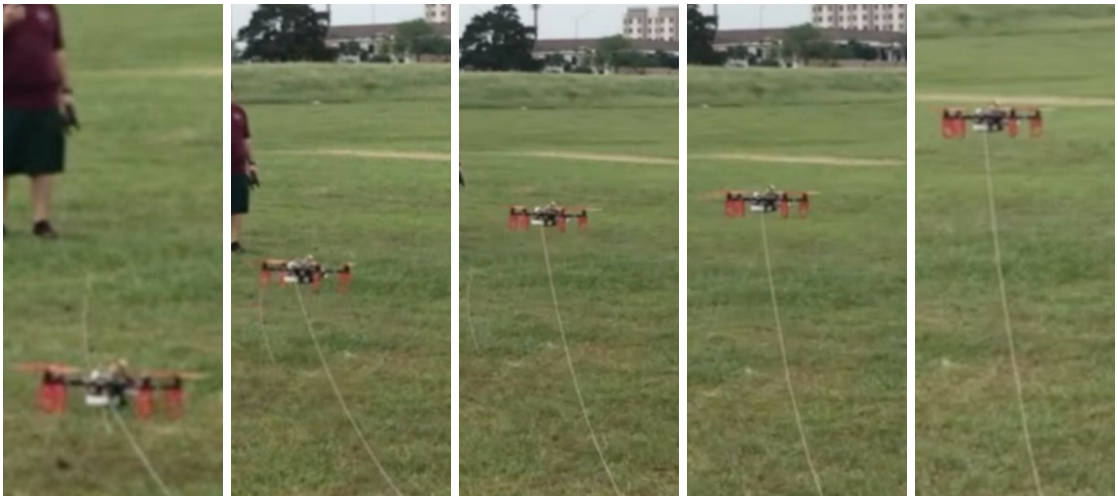


Figure 12: Shows the drone taking off using 1.75 as target altitude (photos taken from video turned in). Drone starts rising in altitude until the altitude is within the bounds for the target altitude.

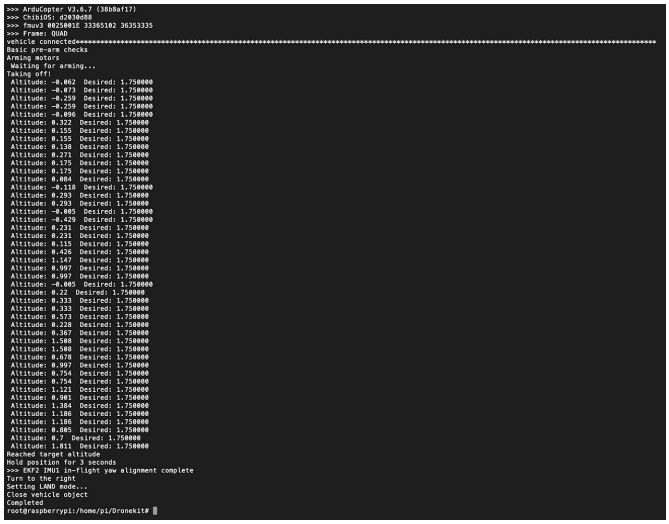


Figure 13: Shows the console output of the drone taking off using 1.75 as target altitude.

Figure 14: Shows the drone landing by setting it to land mode (photos taken from video turned in). Drone descends until it reaches the ground.

Figure 15: Shows the drone moving forward using -5 degree pitch angle (photos taken from video turned in). Drone starts level and then tilts forward to move forward then returns to level.

```
root@raspberrypi:/home/pi/drone/Dronekit# nano new_movements.py
root@raspberrypi:/home/pi/drone/Dronekit# python new_movements.py --connect /dev/serial0
Connecting to vehicle on: /dev/serial0
>>> ArduCopter V3.6.7 (38b8af17)
>>> ChibiOS: d2030d88
>>> fmuv3 0025001E 33365102 36353335
>>> Frame: QUAD
vehicle connected**********************************************************************************************************************
Basic pre-arm checks
Arming motors
 Waiting for arming...
Taking off!
 Altitude: 0.016  Desired: 1.924500
 Altitude: -0.175  Desired: 1.924500
 Altitude: -0.084  Desired: 1.924500
 Altitude: -0.084  Desired: 1.924500
 Altitude: -0.101  Desired: 1.924500
 Altitude: -0.101  Desired: 1.924500
 Altitude: -0.107  Desired: 1.924500
 Altitude: -0.107  Desired: 1.924500
 Altitude: -0.016  Desired: 1.924500
 Altitude: -0.073  Desired: 1.924500
 Altitude: 0.356  Desired: 1.924500
 Altitude: 0.356  Desired: 1.924500
 Altitude: -0.079  Desired: 1.924500
 Altitude: 0.562  Desired: 1.924500
 Altitude: 0.579  Desired: 1.924500
 Altitude: 0.875  Desired: 1.924500
 Altitude: 0.875  Desired: 1.924500
 Altitude: 0.72  Desired: 1.924500
 Altitude: 0.822  Desired: 1.924500
 Altitude: 1.347  Desired: 1.924500
 Altitude: 0.977  Desired: 1.924500
 Altitude: 0.977  Desired: 1.924500
 Altitude: 1.24  Desired: 1.924500
 Altitude: 1.339  Desired: 1.924500
 Altitude: 1.186  Desired: 1.924500
 Altitude: 1.186  Desired: 1.924500
 Altitude: 1.183  Desired: 1.924500
 Altitude: 1.285  Desired: 1.924500
 Altitude: 1.497  Desired: 1.924500
 Altitude: 1.449  Desired: 1.924500
 Altitude: 1.449  Desired: 1.924500
 Altitude: 1.741  Desired: 1.924500
 Altitude: 1.39  Desired: 1.924500
 Altitude: 1.39  Desired: 1.924500
 Altitude: 1.585  Desired: 1.924500
 Altitude: 1.46  Desired: 1.924500
 Altitude: 1.511  Desired: 1.924500
 Altitude: 1.862  Desired: 1.924500
 Altitude: 1.862  Desired: 1.924500
 Altitude: 1.551  Desired: 1.924500
 Altitude: 1.904  Desired: 1.924500
 Altitude: 1.966  Desired: 1.924500
Reached target altitude
Move forward
>>> EKF2 IMU1 in-flight yaw alignment complete
Setting LAND mode...
Close vehicle object
Completed
root@raspberrypi:/home/pi/drone/Dronekit#
```

Figure 16: Shows the console output of the drone moving forward using -5 degree pitch angle.

Figure 17: Shows the drone moving backward using 8 degree pitch angle (photos taken from video turned in). Drone starts level and then tilts forward to move forward then returns to level.

```
 Altitude: -0.22  Desired: 1.346500
 Altitude: -0.31  Desired: 1.346500
 Altitude: -0.35  Desired: 1.346500
 Altitude: -0.226  Desired: 1.346500
 Altitude: -0.226  Desired: 1.346500
 Altitude: -0.005  Desired: 1.346500
 Altitude: 0.104  Desired: 1.346500
 Altitude: 0.104  Desired: 1.346500
 Altitude: 0.014  Desired: 1.346500
 Altitude: 0.039  Desired: 1.346500
 Altitude: 0.07  Desired: 1.346500
 Altitude: 0.07  Desired: 1.346500
 Altitude: -0.141  Desired: 1.346500
 Altitude: 0.121  Desired: 1.346500
 Altitude: 0.064  Desired: 1.346500
 Altitude: 0.138  Desired: 1.346500
 Altitude: 0.138  Desired: 1.346500
 Altitude: 0.259  Desired: 1.346500
 Altitude: 0.387  Desired: 1.346500
 Altitude: 0.599  Desired: 1.346500
 Altitude: 0.599  Desired: 1.346500
 Altitude: 0.333  Desired: 1.346500
 Altitude: 0.559  Desired: 1.346500
 Altitude: 0.737  Desired: 1.346500
 Altitude: 0.737  Desired: 1.346500
 Altitude: 0.192  Desired: 1.346500
 Altitude: 0.192  Desired: 1.346500
 Altitude: 0.661  Desired: 1.346500
 Altitude: 0.545  Desired: 1.346500
 Altitude: -0.237  Desired: 1.346500
 Altitude: 0.5  Desired: 1.346500
 Altitude: 0.5  Desired: 1.346500
 Altitude: 0.305  Desired: 1.346500
 Altitude: 0.466  Desired: 1.346500
 Altitude: 0.833  Desired: 1.346500
 Altitude: 0.833  Desired: 1.346500
 Altitude: 0.678  Desired: 1.346500
 Altitude: 0.178  Desired: 1.346500
 Altitude: 0.175  Desired: 1.346500
 Altitude: 0.175  Desired: 1.346500
 Altitude: 0.214  Desired: 1.346500
 Altitude: 0.194  Desired: 1.346500
 Altitude: 0.194  Desired: 1.346500
 Altitude: 0.409  Desired: 1.346500
 Altitude: 0.486  Desired: 1.346500
 Altitude: 0.686  Desired: 1.346500
 Altitude: 0.87  Desired: 1.346500
 Altitude: 0.87  Desired: 1.346500
 Altitude: 0.977  Desired: 1.346500
 Altitude: 1.209  Desired: 1.346500
 Altitude: 1.209  Desired: 1.346500
 Altitude: 0.624  Desired: 1.346500
 Altitude: 0.853  Desired: 1.346500
 Altitude: 0.839  Desired: 1.346500
 Altitude: 1.449  Desired: 1.346500
Reached target altitude
Move backward
>>> EKF2 IMU1 in-flight yaw alignment complete
>>> EKF2 IMU0 in-flight yaw alignment complete
Setting LAND mode...
Close vehicle object
Completed
root@raspberrypi:/home/pi/drone/Dronekit#
```

Figure 18: Shows the console output of the drone moving backward using 8 degree pitch angle.

Figure 19: Shows the drone moving left using -5 degree roll angle (photos taken from video turned in). Drone starts level and then tilts left to move left then returns to level.

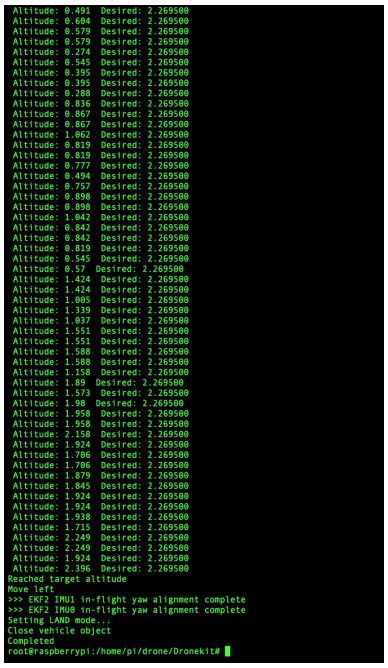


Figure 20: Shows the console output of the drone moving left using -5 degree roll angle.

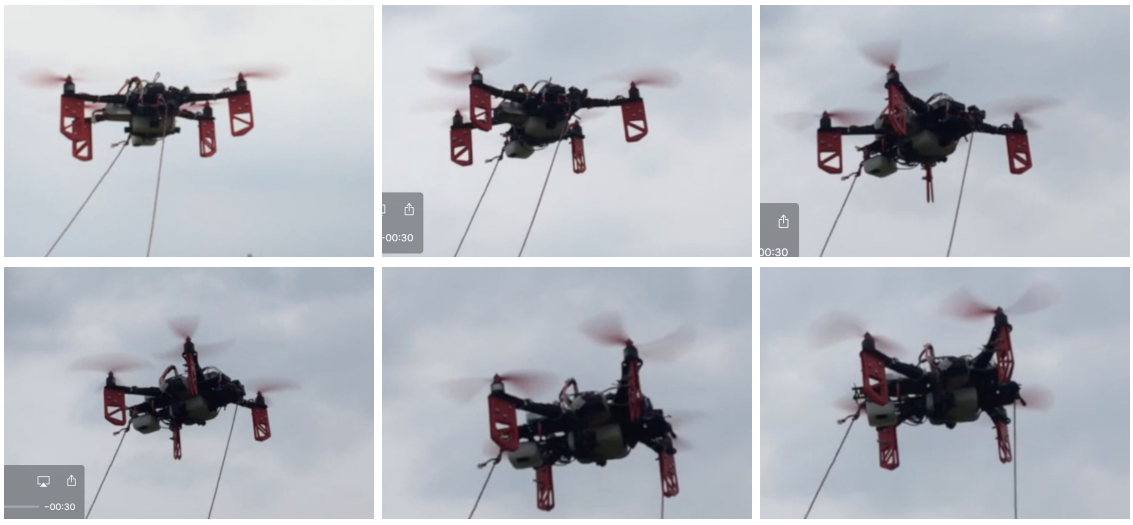


Figure 21: Shows the drone turning left using -90 degree yaw angle (photos taken from video turned in). Drone turns 90 degrees to the left.

```
Altitude: 0.932  Desired: 2.326500
Altitude: 1.2  Desired: 2.326500
Altitude: 0.952  Desired: 2.326500
Altitude: 1.384  Desired: 2.326500
Altitude: 1.384  Desired: 2.326500
Altitude: 0.816  Desired: 2.326500
Altitude: 1.285  Desired: 2.326500
Altitude: 1.155  Desired: 2.326500
Altitude: 1.155  Desired: 2.326500
Altitude: 0.632  Desired: 2.326500
Altitude: 1.175  Desired: 2.326500
Altitude: 1.175  Desired: 2.326500
Altitude: 1.13  Desired: 2.326500
Altitude: 0.895  Desired: 2.326500
Altitude: 0.729  Desired: 2.326500
Altitude: 0.729  Desired: 2.326500
Altitude: 1.2  Desired: 2.326500
Altitude: 0.743  Desired: 2.326500
Altitude: 1.232  Desired: 2.326500
Altitude: 1.059  Desired: 2.326500
Altitude: 1.059  Desired: 2.326500
Altitude: 0.935  Desired: 2.326500
Altitude: 1.121  Desired: 2.326500
Altitude: 1.045  Desired: 2.326500
Altitude: 1.045  Desired: 2.326500
Altitude: 1.198  Desired: 2.326500
Altitude: 1.322  Desired: 2.326500
Altitude: 1.22  Desired: 2.326500
Altitude: 1.22  Desired: 2.326500
Altitude: 0.604  Desired: 2.326500
Altitude: 1.093  Desired: 2.326500
Altitude: 1.356  Desired: 2.326500
Altitude: 1.356  Desired: 2.326500
Altitude: 1.602  Desired: 2.326500
Altitude: 1.299  Desired: 2.326500
Altitude: 1.367  Desired: 2.326500
Altitude: 1.367  Desired: 2.326500
Altitude: 1.251  Desired: 2.326500
Altitude: 1.788  Desired: 2.326500
Altitude: 1.788  Desired: 2.326500
Altitude: 1.788  Desired: 2.326500
Altitude: 1.907  Desired: 2.326500
Altitude: 1.588  Desired: 2.326500
Altitude: 1.777  Desired: 2.326500
Altitude: 1.777  Desired: 2.326500
Altitude: 1.215  Desired: 2.326500
Altitude: 1.975  Desired: 2.326500
Altitude: 1.449  Desired: 2.326500
Altitude: 1.449  Desired: 2.326500
Altitude: 1.562  Desired: 2.326500
Altitude: 2.302  Desired: 2.326500
Altitude: 2.107  Desired: 2.326500
Altitude: 2.107  Desired: 2.326500
Altitude: 2.26  Desired: 2.326500
Altitude: 2.466  Desired: 2.326500
Reached target altitude
>>> EKF2 IMU1 in-flight yaw alignment complete
Turn left
>>> EKF2 IMU0 in-flight yaw alignment complete
Setting LAND mode...
Close vehicle object
Completed
root@raspberrypi:/home/pi/drone/Dronekit#
```

Figure 22: Shows the console output of the drone turning left using -90 degree yaw angle.

Figure 23: Shows the drone turning right using 90 degree yaw angle (photos taken from video turned in). Drone turns 90 degrees to the right.

```
Altitude: 0.127  Desired: 2.057500
Altitude: -0.067  Desired: 2.057500
Altitude: 0.18  Desired: 2.057500
Altitude: 0.175  Desired: 2.057500
Altitude: 0.175  Desired: 2.057500
Altitude: 0.062  Desired: 2.057500
Altitude: 0.254  Desired: 2.057500
Altitude: 0.163  Desired: 2.057500
Altitude: 0.163  Desired: 2.057500
Altitude: 0.418  Desired: 2.057500
Altitude: 0.437  Desired: 2.057500
Altitude: 0.437  Desired: 2.057500
Altitude: 0.613  Desired: 2.057500
Altitude: 0.452  Desired: 2.057500
Altitude: 0.652  Desired: 2.057500
Altitude: 0.652  Desired: 2.057500
Altitude: 1.039  Desired: 2.057500
Altitude: 0.285  Desired: 2.057500
Altitude: 0.285  Desired: 2.057500
Altitude: 0.432  Desired: 2.057500
Altitude: 0.44  Desired: 2.057500
Altitude: 0.48  Desired: 2.057500
Altitude: 0.48  Desired: 2.057500
Altitude: 1.534  Desired: 2.057500
Altitude: 0.723  Desired: 2.057500
Altitude: 0.805  Desired: 2.057500
Altitude: 0.805  Desired: 2.057500
Altitude: 0.522  Desired: 2.057500
Altitude: 0.652  Desired: 2.057500
Altitude: 1.056  Desired: 2.057500
Altitude: 1.056  Desired: 2.057500
Altitude: 0.745  Desired: 2.057500
Altitude: 0.875  Desired: 2.057500
Altitude: 0.822  Desired: 2.057500
Altitude: 0.822  Desired: 2.057500
Altitude: 1.248  Desired: 2.057500
Altitude: 0.924  Desired: 2.057500
Altitude: 0.737  Desired: 2.057500
Altitude: 0.737  Desired: 2.057500
Altitude: 1.223  Desired: 2.057500
Altitude: 0.986  Desired: 2.057500
Altitude: 0.986  Desired: 2.057500
Altitude: 1.169  Desired: 2.057500
Altitude: 1.107  Desired: 2.057500
Altitude: 1.308  Desired: 2.057500
Altitude: 1.308  Desired: 2.057500
Altitude: 1.11  Desired: 2.057500
Altitude: 1.065  Desired: 2.057500
Altitude: 1.381  Desired: 2.057500
Altitude: 1.381  Desired: 2.057500
Altitude: 1.393  Desired: 2.057500
Altitude: 1.633  Desired: 2.057500
Altitude: 1.932  Desired: 2.057500
>>> EKF2 IMU1 in-flight yaw alignment complete
Altitude: 1.932  Desired: 2.057500
Altitude: 2.082  Desired: 2.057500
Reached target altitude
Turn right
Setting LAND mode...
>>> EKF2 IMU0 in-flight yaw alignment complete
Close vehicle object
Completed
root@raspberrypi:/home/pi/drone/Dronekit#
```

Figure 24: Shows the console output of the drone turning right using 90 degree yaw angle.

Figure 25: Shows the drone turning around using 180 degree yaw angle (photos taken from video turned in). Drone turns 180 degrees to the right.

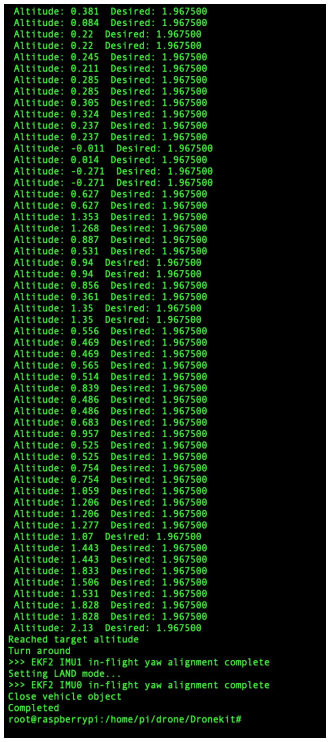


Figure 26: Shows the console output of the drone turning around using 180 degree yaw angle.

4. Conclusion

4.1. Key Decisions

Several key decisions were important in the development of each of the subsystems. For the overall project there were also key decisions that influenced and were influenced by the subsystems themselves.

The first major decision in the development of the whole project was whether the drone would be land or air based. The initial project description called for a flying drone but due to us foreseeing the issues that we eventually encountered we considered instead creating a land-based automaton. This new implementation would have come with its own problems but would have been safer to operate. The decision was made to stick to the original project's specifications of a flight-based drone upon discussion with our sponsor.

The second major decision was what type and size of drone to use. This decision had a large impact on the overall outcome because it affected the operation. We opted to go with a larger sized drone because we were able to get one for free from our previous TA Andrew Miller's lab. The decision was made based on the fact that the larger drone was already assembled, saving us time, had a large enough battery to sustain the time of flight required, and had enough space onboard for the modifications we eventually added. This decision meant that stability of the drone went down due to the size and weight. Looking back it may have been better to curtail our modifications and opt for a smaller drone that could be flown stably and safely.

The next major decision pertains to the navigation subsystem but was major in the overall scheme of things; how to actually navigate. Several choices were available each

with their own pros and cons. GPS was out because our flights are indoors where no GPS lock can be made. Initial ideas were to use the Aruba bluetooth beacons already placed around the Zachry building to triangulate position but this was thrown out after research showed that this method could not be easily performed on the Raspberry Pi. LiDAR was next considered due to its resolution and range but was thrown out after research suggested that the large amount of glass within the building would have thrown off readings. It was settled that the navigation would be done using ultrasonic sensors reading variations in the wall to tell location. Though everything was done to error-correct, this method is not the most reliable. This method was chosen mostly due to other methods not even being viable.

4.2. Learnings

Over the course of this project we learned that it's very important to have as accurate of equipment as possible. This held us back with the drone due to poor gyroscope and accelerometer readings as well as with our ultrasonics which, at first, would gives us very poor values occasionally. We also learned that safety always needs to come first in a design. After realizing how unsteady our drone flew and how many unexpected things can happen during a drone flight, we concluded that it would be unsafe to fly our project inside.

4.3. Future Work

If the drone were to become an actual product it would need a more steady and accurate flight. Currently it doesn't fly straight enough to be safe for indoor flight. There are also so many possibilities of things that could go wrong during the tour and we probably didn't account for scenarios that never even crossed our minds. We would also need to implement a secondary positioning check. With a single point of failure being the ultrasonics sensors, there is a high possibility the we can lose track of where we are.

Execution Plan

	October 15	**October 29**	**November 12**	**November 26**	**Dec 3**
Drone Subsystem	Drone parts put together	Drone flies	Sensors added to drone and coded	Drone avoids collisions	Drone flight tested in Zachry
Navigation Subsystem	Raspberry Pi connects to beacons	All first floor dimensions are obtained	Algorithm for obtaining checkpoint is made	Model is made to simulate hallway	Simulation GUI is made to show drone
Tour Group Tracker	Get Pi camera feed	Recognize people	Create boundary update logic	React to distance by changing speed	Drone keeps distance
Mobile Application	Draft application layout	Start coding GUI	Connect to drone	Route selection	Send tour destinations to drone

Task	**Completion Date**
Combine App & Navigation	January 24th, 2019
No GPS Dronekit Flight	January 31st, 2019
Tracking Loss Contingency	February 7th, 2019
Integrate PX4Flow with Pixhawk	February 7th, 2019
Tour Destination Audio Recorded	February 14th, 2019
3D Printing Order Placed	February 14th, 2019
Camera Integrated	February 14th, 2019
Tour Tracker Completed	February 21st, 2019

Fully Integrated Tour Tracker with Navigation	February 28th, 2019
Sensors Fully Integrated with Navigation	February 28th, 2019
Tour Destination Audio Fully Integrated with Navigation	February 28th, 2019
Drone Flight of First Floor Hallway	March 7th, 2019
Stairway & Second Floor Hallway added to Tour	March 14th, 2019
All subsystems integrated together	March 21st, 2019
Drone Completes Tour on one Battery Charge	March 28th, 2019
Drone Completes Tour without Collisions	April 4th, 2019
Drone Completes Tour by Staying with Group	April 11th, 2019
Project Validated	April 15th, 2019
Complete Final Report	May 1st, 2019

Validation Plan

	Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
Drone Subsystem	Confirm drone and components power on	Confirm drone hovers in place	Confirm sensors give accurate readings	Confirm drone doesn't collide with anything	Confirm drone can fly a test path
Navigation Subsystem	Confirm Raspberry Pi connects to beacons	Confirm all wall changes and checkpoint lengths are found	Confirm algorithm outputs correct checkpoint when fed sensor data	Confirm model can create fake sensor readings and algorithm will do correct action	Confirm drone and map data match with the GUI output
Tour Group Tracker	Confirm video feed output	Confirm people in frame through boundary box	Confirm boundary update logic executes properly	Confirm range conditions execute (too far, too close, right)	Confirm real response is appropriate to distance measured
Mobile Application	Confirm group all approve of application layout	Confirm application can be loaded on iPhone and Android	Confirm application connects to drone	Confirm selection of tour is sent to drone	Confirm application tells drone the list of destinations for the selected tour

Validation	Completion Date
App used to start tour navigation	January 24th, 2019
Drone able to takeoff and land	January 31st, 2019
Tracker reset after tracking loss	February 7th, 2019
Have Drone move short distance	February 7th, 2019
Able to play tour audio on PI	February 14th, 2019
Confirmed order placed	February 14th, 2019
Read and manipulate camera	February 14th, 2019
Runs off camera and holds tracking	February 21st, 2019
Drone flies tour and reacts to crowd	February 28th, 2019
Drone says when new checkpoint reached	February 28th, 2019
Have drone give speech during test runs	February 28th, 2019
Drone flies through first floor	March 7th, 2019
Drone flies through desired locations	March 14th, 2019
First full tour given without errors	March 21st, 2019
Drone completes full tour without low battery warning	March 28th, 2019
Drone completes full tour without collisions	April 4th, 2019
Drone completes full tour without losing tour group	April 11th, 2019
Drone tour completed	April 15th, 2019
Final report submitted	May 1st, 2019