Interim Report

DRAFT OF FINAL REPORT

Programmable Flight Controller

Brendon Camm
B00649176

Lucas Doucette
B00685962

Dylan Humber
B00695554

Submitted: March 27 , 2017

This document was written in LaTeX. It was compiled with pdflatex. The source of the document may be viewed at:
https://github.com/Brendoncamm/SYP/tree/master/Documents/Second%20Semester%20Report

Department of Electrical
and Computer Engineering
Dalhousie University
P.O. Box 1000
Halifax, Nova Scotia
B3J 1B6

March 26, 2017

Dr Jason Rhinelander
Reiland Systems Ltd.
PLACEHOLDER ADDRESS
Dartmouth, NS
POSTAL CODE
CC: Dr. Jason Gu
Dr José Gonzalez-Cueto
ECE Support Secretary

Dear Dr. Rhinelander:

Attached is Group 2s senior design interim report titled Programmable Flight Controller, written in fulfillment of the requirements of the Faculty of Engineering ECED 4901 Senior Design course. The report outlines the specifics of each portion of the project encounters throughout the 2016/2017 academic year.

The intention of this project is to develop a completely programmable flight controller for use on a quad-rotor drone. The controller and sensors will be implemented using an Arduino Uno Microcontroller and Raspberry Pi 3 computer. A graphical user interface was also developed to display information about the controller and sensors. The sections enclosed in the attached report discusses the simulation, implementation and testing of the controller as well as the design and testing of the graphical user interface. In addition, this report outlines the management side of the project including the Gantt Chart showing the time-lines, task division and each individual group members responsibilities. The report concludes with sections highlighting some of the challenges that were encountered over the course of the project including a new legislation regarding recreational drone usage introduced by the federal government in March of 2017 which severely disrupted our planned flight testing following this section will be recommended future improvements.

Please contact the groups nominated project manager, Dylan Humber (at dylan.humber@gmail.com) should you have any questions about the report.

Best Regards,

Dylan Humber

Brendon Camm

Lucas Doucette

Contents

Abstract

The following report summarizes the progress made throughout the Quad Rotor Drone Programmable Flight Controller project and the future recommendations based upon project findings. The goal of the project is to design a programmable quadcopter flight controller that will have the capability of responding to live control inputs and react appropriately to disturbances present. The objective of this report is to summarize the deliverables set forth for the project and to present findings, results and testing procedures. The report also outlines the design process of the project. The Quad Rotor Flight Controller project has three major components; flight and stabilizing simulations, implementation of software, and a GUI interface.

Authorship

- Abstract - Lucas Doucette
- Letter of Transmittal - Brendon Camm
- Simulations - Dylan Humber
- GUI - Brendon Camm
- Physical Implementation - Lucas Doucette
- Conclusions - Dylan Humber
- Future Recommendations - Group
- References - Group
- Appendices - Group

1 List of Acronyms

1. BLDC - Brushless Direct Current
2. DOF - Degrees of Freedom
3. FOV - Field of View
4. GUI - Graphical User Interface
5. HID - Human Interface Device
6. IMU - Inertial Measurement Unit
7. PD - Proportional Derivative
8. PI - Proportional Integral
9. PID - Proportional Integral Derivative
10. POC - Proof of Concept
11. PWM - Pulse Width Modulation
12. RF - Radio Frequency
13. RPi - Raspberry Pi
14. USB - Universal Serial Bus

2 Introduction

2.1 Background and Significance

Machine learning is defined as the science of getting computers to act without being explicitly asked. This is achieved by the development of computer programs that can teach themselves to grow and change when exposed to different sets of data. There are two traditional types of machine learning algorithms: Batch learning algorithms and on-line machine learning algorithms. Batch learning algorithms require a set of predefined training data that is shaped over the period of time to train the model that the algorithm is running on. On-line learning uses an initial guess model that forms co-variates from that initial guess then passes them through the algorithm to form an evolved model a new set of covariates are formed from the evolved model and then fed back to make a new prediction. The loop

runs continuously so that the evolved model is constantly growing and learning to adapt to certain situations. Dr. Rhinelander's research is concerned with on-line machine learning algorithms therefore the drone we are developing will configured to adapt with these algorithms.

Quad rotor drones have been on the rise in popularity in the last several years due to their simplistic mechanical design and many practical uses. The application of these drones vary from a hobbyist flying around their neighbourhood to military personnel carrying out high risk missions. A video recording device of some sort is generally attached to the drone and the video feed is relayed to a base station for the operator to gain a field of view (FOV) of an area of interest. Having the capability to have a continuous video feed allows the drone to be used for many practical applications including but not limited to: Traffic condition monitoring and surveillance missions. While these drones are very sophisticated and advanced devices they are are missing one aspect that is very important to further Dr. Rhinelanders research: They are not totally configurable.

As mentioned, Dr.Rhinelander's research is concerned with on-line machine learning algorithms and without a platform that is completely configurable his research would be limited. Before the machine learning algorithms are implemented onto the drone it must first be able to be controlled. This is where we come in, we have been tasked by Dr. Rhinelander to develop a flight controller that receives control inputs over Wi-Fi. Having a completely open source flight controller will allow for the addition of the machine learning algorithms to the flight controller software so that the drone can learn to partially, and eventually fully fly on it's own and make intelligent decisions.

2.2 Overall Objective

The main objective of the project is to develop a programmable flight controller that responds appropriately to control inputs and disturbances. The flight controller will receive control inputs over Wi-Fi from a base station. The base station can be any device that is Wi-Fi enabled and has the appropriate software installed. The software on the base station will be a graphical user interface (GUI) that allows the user to send control inputs to the drone and view statistics of the drone during operation.

2.3 Simulations

The simulations will allow us to gain an understanding of how the controller will respond to specific inputs. The simulation can then be tuned to test various PID constants for performance. We will be simulating both the flight dynamics and

controller using MATLAB and Simulink exclusively.

Additionally, the simulation will be able to take input from both predefined control sequences and HID on the operating system. As real time input from a HID is desired for the simulation, real-time rendering in 3D will be attempted.

2.4 Physical Implementation

The physical implementation of the drone software and hardware requires a communication channel between a control input, a base station host, a wireless communication receiving unit, and a host for the controlling system. The objective of the physical implementation is to provide the software and hardware design required to successfully implement a quad rotor drone flight controller. It is desired to have a controller that is capable of sensing disturbances in flight and stabilizing the system based upon the measured disturbances.

The objectives of the physical implementation for the project begin with successfully creating a Wi-Fi communication channel between a controller input and a wireless device placed on the drone hardware. The drone is to have flight sensing capabilities, interfaced with both the on board wireless device and a grounded controlling station. The flight sensing is to include yaw, pitch, roll and altitude measurements with a polling rate sufficient for real time control of the drone. The flight sensing is to be combined with a set point from the wirelessly transmitted control input by a PI or PID loop to control the flight of the drone with stabilizing features. The code written for the control input, communication interface, and finally the controller is to be well documented to ensure ease of use and simplicity for updating and modifying after handing over the final product.

2.5 Graphical User Interface

The Graphical User Interface (GUI) requires a network connection between the base station host it lives on and the Raspberry Pi 3 to receive the serialized dictionaries containing sensor data and controller information. The objective of the GUI is to provide the user with a medium to access information regarding the current flight, such as, the altitude at which the drone has flown or how the physical controller is configured. It is desired to have a real time plot of the altitude based on the sensor information and the ability to initialize the physical controller.

The objectives for the GUI for the project start with selecting an appropriate development framework that will function across Windows, Mac OS and Unix environments. The GUI is to have the ability to initialize the communications between the base station host and Raspberry Pi 3 to allow for the control inputs to be sent using the base station host as well as receiving the data that is to

be displayed on the GUI. The GUI is to be intuitive to avoid any unnecessary confusion with the end user. The code written and any other software used for the GUI will be provided and well documented to ensure that in event that the end user would like to modify anything after receiving the final product, this can be done effortlessly.

3 Methods

3.1 Simulations

In order to begin pursuing the simulation, problem areas were first identified. This is to say, the investigation or creation of blocks that we had no experience with. The PID and other control blocks used in the simulation we were already accustomed to. Other blocks such as addition, gain, (de)multiplexers, and other common blocks we were also familiar with. The solutions to these problem areas, as well as the overall design of the simulation is discussed in section **??**. The problem areas that we identified were as follows:

- HID Input
- Physical simulation
- Real-time rendering

The investigation into these issues would first begin by searching for existing blocks or packages within SimuLink. In the absence of already existing solutions, useful packages would be identified. Any packages or blocks that could be used as tools to build the desired effect. Subsequently, the amount of effort to build the result into the simulation using these packages would then be evaluated. However, in the progression of the simulation, the majority of the tools (as discussed in section **??**) were easily used to implement the problem areas.

3.2 Physical Implementation

The physical implementation of the system began with determining a method to interface a controller input with a base station host. To do this, research of available python libraries was performed. It was determined quickly that the most effective route would be to use Python's Pygame library which analyses a bluetooth or usb connected device and determines the types of control inputs applicable for the device. Initially, a PS4 bluetooth connected controller was successfully utilized, receiving all axis and button inputs through the Python interface.

The next challenge of the physical implementation was to determine the best method of Wi-Fi communication. It was determined that Python's socket library would be best suited, and a Raspberry Pi 3 with a Wi-Fi module available would act as the server in the server, client architecture. A preliminary communication server client script set was written and tested successfully between the base station laptop and the Raspberry Pi server host.

For the controller host, two possible solutions were considered. Utilizing an Arduino micro-controller as a permanent host of the control loop, communicating with the Raspberry Pi to receive control input data or using the Raspberry Pi for both communication and the controller. Using Arduino would provide a dedicated control loop and an intuitive interface to prototype with. The Raspberry Pi is capable of running the control loop, but issues with regards to a real time operating system were anticipated. Using an Arduino would require more mounting space on the drone as well as another communication channel between the RPi and Arduino to troubleshoot. Many other controllers could have been considered but both RPi and Arduino are owned by group members.

The controller required input from a 10 degree of freedom inertial measurement unit to calculate yaw, pitch, roll and altitude to poll and compare to the control set points. Eventually, alternative methods of calculating altitude were considered and discussed. The methods included infra-red distance sensing, ultrasonic sensing, multiple barometers coupled with Kalman filters, or a barometer accelerometer coupling. PI and PID loops were researched to be implemented on the controller. Several additional filtering techniques were considered for altitude measurement, including moving average filtering, averaging filtering and Kalman filtering with a feedback loop for comparison.

Communications between the RPi and the Arduino were researched, using the interface to interface bus or the serial communication channel were both considered. The data was required to be transmitted as single bytes, thus an algorithm to convert received bytes into floating point values needed research and development.

From a technical background perspective, research regarding the Python language and Python's available libraries was required. Server to client interfacing is also required technical background. Previous knowledge of the Python language assisted in directing the research and development flow throughout the project. PI and PID control loop knowledge was required and research time was used to learn filtering processes, Arduino library development, miscellaneous communication algorithms and utilization of the Adafruit 10 DOF IMU.

3.3 Graphical User Interface

3.3.1 Framework Selection

There is a multitude of frameworks available for graphical user interface development so in order to narrow down the choices the following restrictions were placed:

- Must function on Windows, Mac OS and UNIX environments
- Have the ability to display live plots
- Be compatibile with the physical controller initialization software
- Have well documented libraries for ease of development

With these restrictions placed and further research into various frameworks the selections narrowed down to Qt and PyQt5. Each of these use the Qt framework which is regarded as a reliable, cross-platform GUI development software with extremely well documented libraries which met the major requirements. Although both Qt and PyQt5 use the same framework there are a few key differences.

The most notable difference at first glance is that the two use different languages, Qt uses C++ where PyQt5 uses Python but the biggest difference is in the design suites. When developing using Qt you're able to use the QtCreator design suite which allows the user to design the look of the GUI using click and drag widgets, code the functionality of the widgets then simply compile all within the design suite. When using PyQt5 there isn't a complete design suite, instead the user must design the user interface using QtDesigner and then import the GUI file into the Python script, from here the widgets are coded is relatively the same other than the obvious differences between C++ and Python. To finalize the decision on which framework to use two basic GUI's with the same functionality were developed using Qt and then PyQt5. From a technical background perspective, research regarding Python and C++ languages and the available libraries for each was required as well as research into the capabilities of both Qt and PyQt5 development frameworks.

3.3.2 Features

Many different features were considered when initially designing the graphical user interface such as:

- Display Live Plots
- Initialize Manual Control of the Drone

- Manipulate and display PI and PID Parameters
- Display Sensor Status
- Display Flight Time
- Initialize communications between base station host and Raspberry Pi 3

Each of these features have pros but upon futher research on how each individual feature could be implemented highlighted that some may be out of the scope of this project. It was decided that regardless of the difficulty of implementation that our GUI would at least contain the ability to have live plots, initialize the physical controller and initialize the communications between the base station host and Raspberry Pi 3 therefore further research was put into these. From a technical background perspective the research done required knowledge of the Qt and PyQt5 framework, networking knowledge and knowledge about basic PI and PID controllers.

3.3.3 GUI Layout

The layout of a GUI can make or break the end users experience, a clunky non-intuitive GUI will undoubtably cause the end user an unneeded headache. For this reason, a prototype of a graphical user interface that included some of the features mentioned above was developed and sent to our client to get an idea of what he would be comfortable with, this prototype was then optimized based on his comments. From a technical background perspective designing the two prototypes requires knowledge on how to develop using QtDesigner as well as basic GUI layout techniques.

3.3.4 Live Plotting

When thinking about how to plot the live sensor data two libraries were considered these are, matplotlib and pyqtgraph. Matplotlib is a favorite when it comes to plotting data with Python where pyqtgraph is a plotting library specifically developed for use on PyQt. Each of these libraries were used to create plots on the GUI and then the performance was examined. The performance factors that were considered were: latency to plot when data is received and accuracy of the plotted data. Research into both of these libraries and how they can be integrated into the user interface was performed.

4 Proposed Solutions

4.1 Simulations

For the control design, four separate controllers will be used. A PID controller will be used to control the altitude of the quad-rotor drone. Pitch, roll, and yaw will each be controlled respectively by a separate PI controller. These controllers will be first tested and tuned using test inputs. The physical aspect of the simulation will be defined by a SimScape MultiBody model. Once this rudimentary model is verified, the simulation will be developed into taking real-time signals from a HID and rendering the flight of the quad-rotor drone in response to user input.

4.1.1 Simulation Design

The overall architecture of the proposed design can be seen in Figure 4.1.1. The entire system is broken down into six subsystems. Their specifications are discussed below. The layout of each subsystem and the overall SimuLink simulation layout may be found in Appendix **??**.

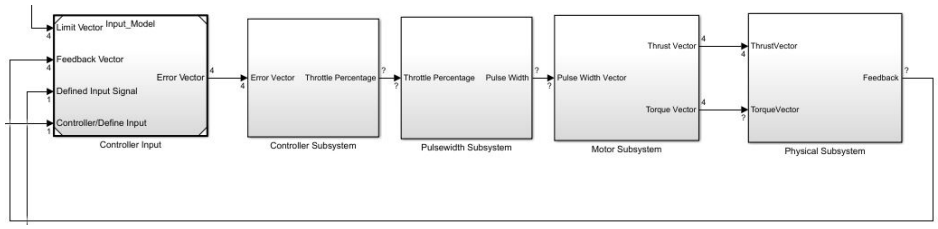


Figure 1: Proposed Simulation Architecture

1. **Input Block:** This block handles the HID input and pre-configured test signals. As well as the specifications of limit/scalar values of the input signals. As the limits will be scalar values, control signal input should be restricted between -1 and 1.

2. **Feedback Block:** This block receives the contents specified from the input block as well as the current orientation of the quad-rotor drone in the simulation. It combines all of these to output an error signal for the pitch, roll, yaw, and altitude controllers.

3. **Controller Block:** This block houses the four controllers. The output of these controllers is normalized to a throttle percentage.

4. **Pulsewidth Subsystem:** This block takes the throttle percentage and transform it to a value in microseconds that can be used to define the pulsewidth output of the microcontroller. Physically, this PWM signal controls the ESCs.

5. **Motor Characterization:** This block contains motor thrust data corresponding to characterization tests performed in the first semester. Functionally this block contains a look up table that relates pulsewidth to thrust and torque for each motor. The torque value is an approximation based on the value of thrust from the motor.

6. **Physical Subsystem:** This block contains the physical subsystem built in SimScape Multibody. This library allows for the system to be built quickly and easily, as well as providing a (although simple) 3D rendering model.

7. **3D Rendering:** This subsystem will handle the real-time rendering of the simulation for flying with input from the HID.

4.2 Physical Implementation

The proposed solution incorporates the following components for the physical system architecture.

- Joystick Controller
- Base Station Laptop
- Raspberry Pi 3 Computer
- Arduino Micro-Controller
- 10 Degree of Freedom Inertial Measurement Unit
- Raspberry Pi Camera Module (Future)

The system architecture is implemented as depicted in the block diagram shown in Figure 2.

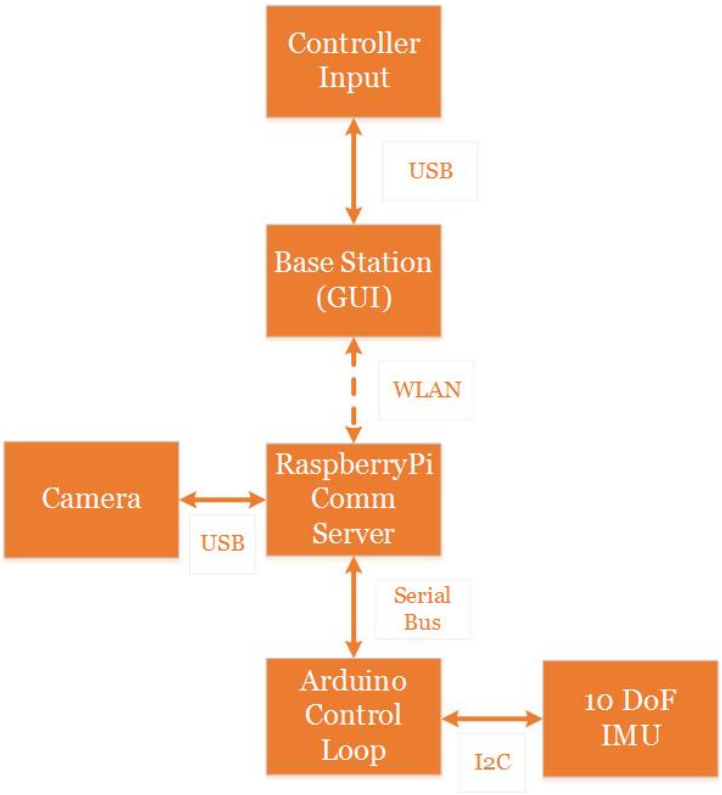


Figure 2: System Architecture

A joystick controller was used as the control input for its non-spring loaded throttle control. The drone requires an altitude set point from the control input, a spring loaded control input would make the implementation of a constant set point difficult. The base station laptop hosts the GUI and the communication channel software between the USB connected joystick control input and the Raspberry Pi Wi-Fi connection. The Python scripts that enable the communication channels use the sockets library for a server, client Wi-Fi channel and pygame library to acquire control set points from the joystick controller. The inputs from the controller are converted from floats to sets of four bytes for transmission.

The Raspberry Pi is used solely as a communication channel between the base station and the Arduino control loop. The RPi is the server while the base station is considered the client in the connection. The Raspberry Pi relays the received bytes to the Arduino control loop using a serial communication channel.

The Arduino control loop is used to host the drone function library and main script required for drone flight. The Arduino control loop outline is shown in Figure

3. The gain values for the PID loop are to be determined through simulations. The Arduino initializes all sensors, motors, and communications. The loop then gathers data from the inertial measurement unit and the serial connection to the Raspberry Pi, compares the data and performs a PID loop, finally outputting the results to the ESCs.

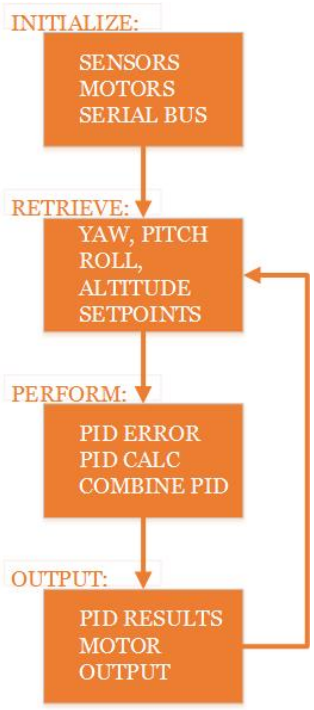


Figure 3: Arduino Control Loop

Yaw, pitch, roll and altitude are calculated using the accelerometer, gyroscope, magnetometer, barometric pressure sensor and temperature sensor available on the inertial measurement unit. The altitude measurement is completed using the barometric pressure sensor and temperature sensor. The equation to calculate altitude is given as:

$$h = \frac{(\frac{P_0}{P})^{\frac{1}{5.257}} - 1) * (T + 273.15)}{0.0065}$$

h = Difference between Starting Height and Current Height
P_0 = Initial Pressure at Starting Height
P= Current Pressure

T = Current Temperature

The barometric pressure sensor calculation was found to be highly susceptible to noise, several filtering techniques were attempted and a Kalman filter with a feedback comparison is currently in use. This filtering technique is subject to change. All written scripts for the Arduino, Raspberry Pi and the base station communication channel can be found in Appendix B.

The hardware build of the drone requires mounting the Arduino and Raspberry Pi on the drone, connecting the 10 DOF IMU to the Arduino through an I2C connection, ESC connections through the available PWM pins on the Arduino, and a serial connection to the Raspberry Pi. The ESCs are to be powered by a lithium ion battery through a power distribution board. The power distribution board will also provide power supply to the Raspberry Pi, supplying power to the Arduino through the serial USB connection.

4.3 Graphical User Interface

4.3.1 Framework Selection

The basic GUI was to have the ability to call the PS4 controller initialization script and manipulate the axis settings. Because the initialization script was written in Python it was very difficult and time consuming to do this using the C++ version of Qt compared to using PyQt5 where all that had to be done was import the script. Due to each of the scripts that the GUI needs to call are written using Python it was decided to carry on using PyQt5. On top of the ease of importing the scripts using PyQt5 it also allows us to use the vast amount of Python libraries that are available such as matplotlib and pyqtgraph and as discussed, these libraries will be used to plot the sensor data.

4.3.2 Features

As mentioned in section 3.3.2 how we could go about implementing each of the listed features was researched. Through this research is was revealed that some of the features we had originally considered were out of the scope of this project because it would take too much time to implement for little benefit. For this reason the following features were removed from our consideration at this time, but they will be considered potentional future improvements:

- Manipulate and Display PI and PID Parameters
- Display Sensor Status

- Display Flight Time

Therefore the key features that will be implemented into the GUI at this time are as follows:

- Initialize Manual Control of the Drone
- Initialize communications between Base Station Host and Raspberry Pi 3
- Display Live Plots

A detail explanation of how these features will be implemented into the GUI will be explained in sections 4.3.4-4.3.6.

4.3.3 GUI Layout

As explained in section 3.3.3 an initial GUI was developed and presented to our client this initial prototype can be viewed in APPENDIX X. As you can see from the figures in APPENDIX X the prototype presented to our client included most of the features we later deemed unnecessary therefore we could disregard these extra widgets. With the decision to not include the features and taking our clients comments into consideration a new GUI layout was developed. The new GUI layout includes all of the key features we have decided to implement and presents them in a clean and intuitive manner. This new layout was presented to each group member and a consensus was reached to carry on with development using this layout. The final GUI layout can be viewed in Figure 4 and Figure 5.

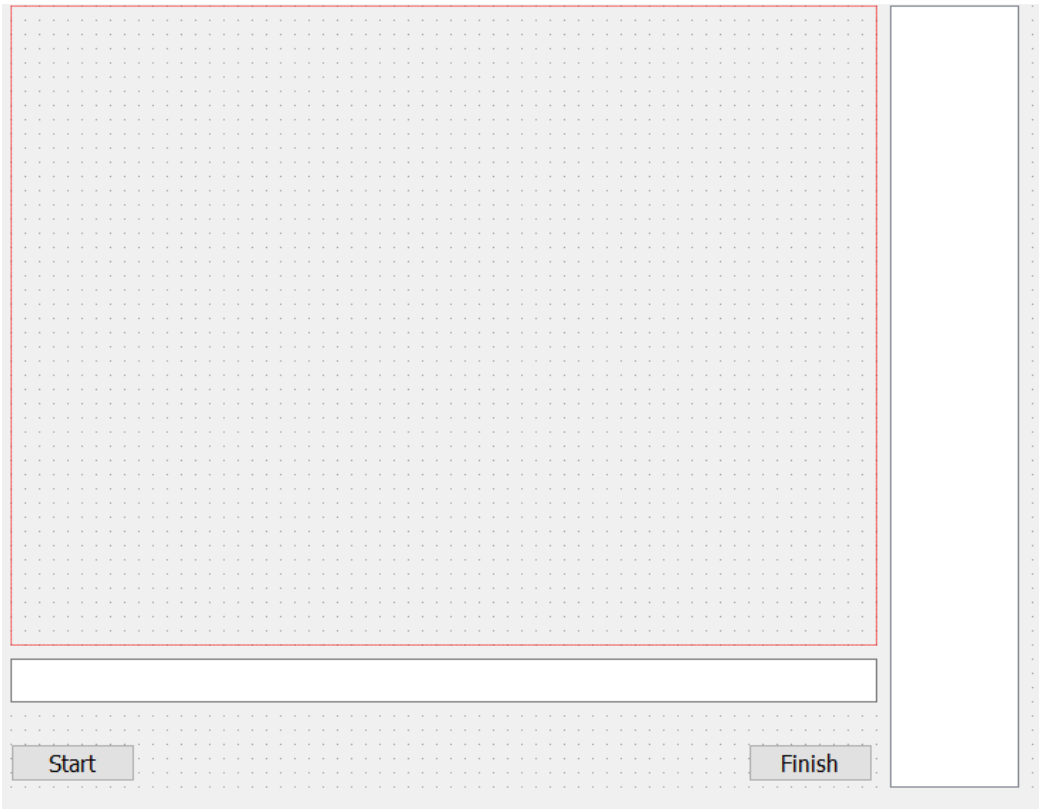


Figure 4: Final GUI Home Page

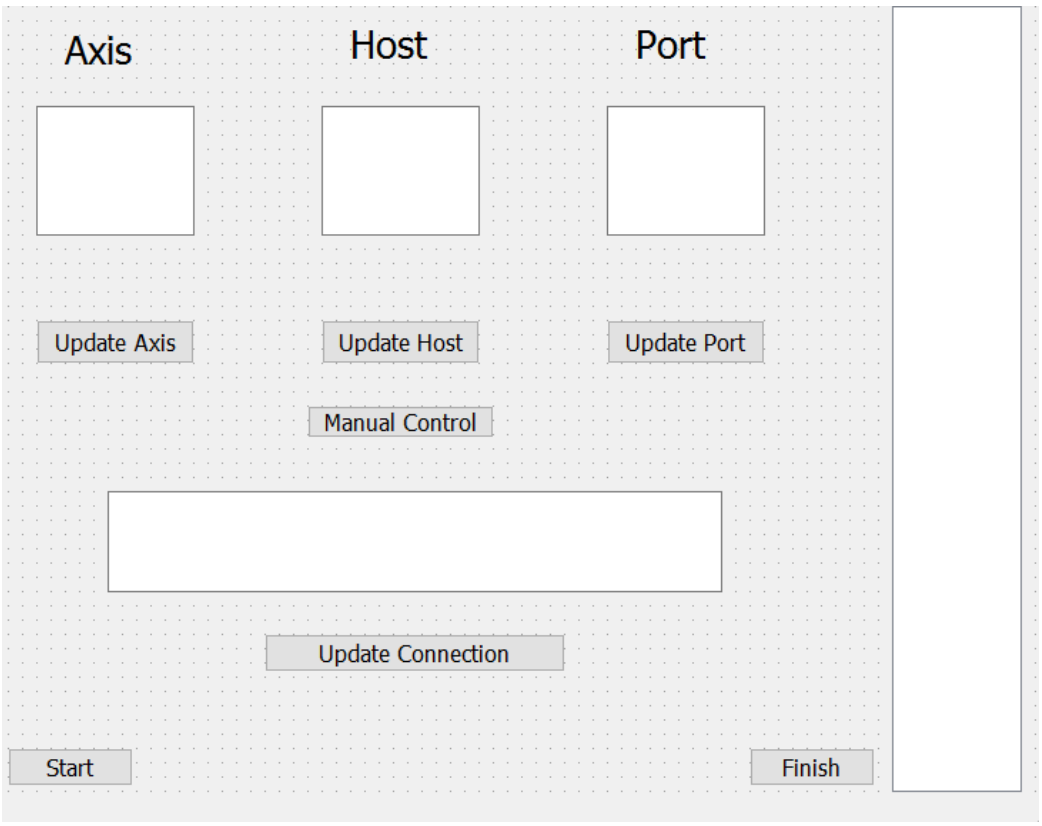


Figure 5: Final GUI Controller and Network Settings page

The red outlined box is the field that the live plot will populate when the GUI is compiled and the blank vertical rectangle is a list widget that will be populated with the names different pages of the GUI and will allow the user to access them by simply clicking on the name. An example of how the list widget will look can be viewed in Figure 6 and how the live plot will look can be viewed in section 4.3.5. An explanation of what purpose each of the other widgets and push buttons seen in Figures 4 and 5 will serve is described in the following sections.

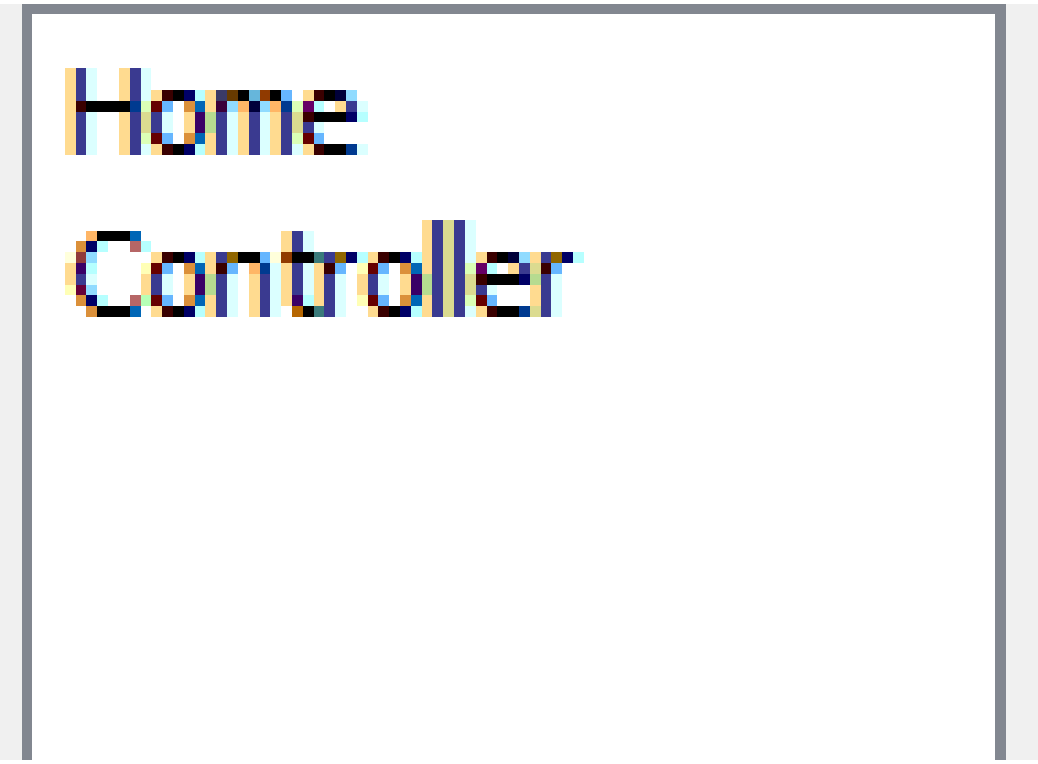


Figure 6: List Widget Example

4.3.4 Initializing Communications and Manual Control

In order for the GUI to receive sensor data or the controller to receive manual control inputs over WiFi a socket connection must be made. Instead of having to open and run the socket script on the client (base station host in our case) the Start button on the GUI calls the socket script and makes the connection to the socket. Once the Start button has been pressed the horizontal white fields found in Figures 4 and 5 will tell the users whether or not communications have been successfully made how this push is coded can be found in the connection() definition of the GUI code in APPENDIX. When the GUI is initially compiled the white fields under Axis and Port seen in 5 will be populated with the default values that are found in the PS4 Controller script for the Axis values and the socket script for the Port value. The white field under Host will be populated with the name of the host computer's IP or name.

The Axis values will determine which direction the drone will go depending

on which way to joystick is moved therefore if the user would prefer the drone move differently they have the option to change the joystick settings by hitting the "Update Axis" push button. How this push button is coded to achieve this functionality can be found in the updateAxis() method in the GUI code in APPENDIX

In the event that the client script is altered and a new host and port number is to be used for the socket the "Update Connection" push button will allow the user to enter the new host and port values and then the connection to the new socket will be made. If only one value has changed the user can select either the "Update Host" or "Update Port" push buttons and then hit "Update Connection" and the new connection will be made. If either of these cases occur the horizontal white field found in Figure 5 will be populated letting the user know whether or not the values have been updated and the connection has been successfully established. How the Axis value, Host value, Port value and Connection Status will be displayed when the GUI is compiled and the Start button is pushed using the wrong port value can be viewed in Figure 7 and how the text field that pops up when you click any of the push buttons mentioned looks can be viewed in Figure 8

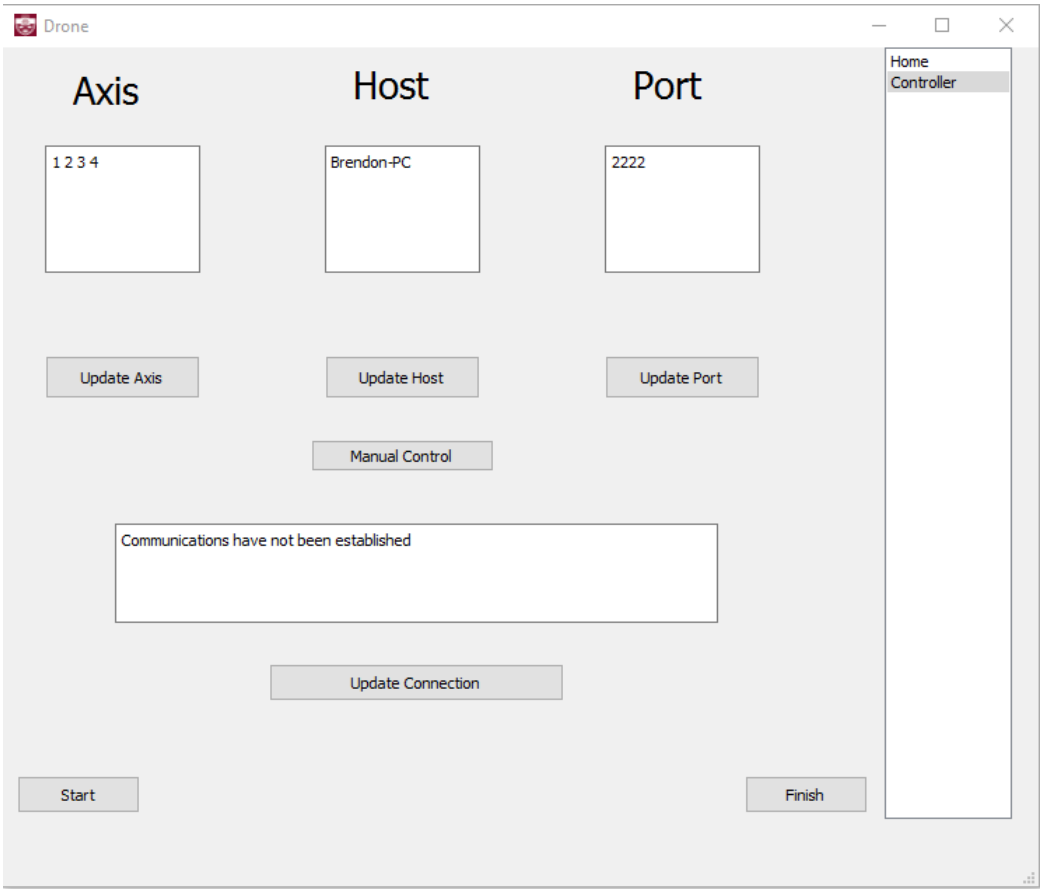


Figure 7: Compiled GUI with no active communications

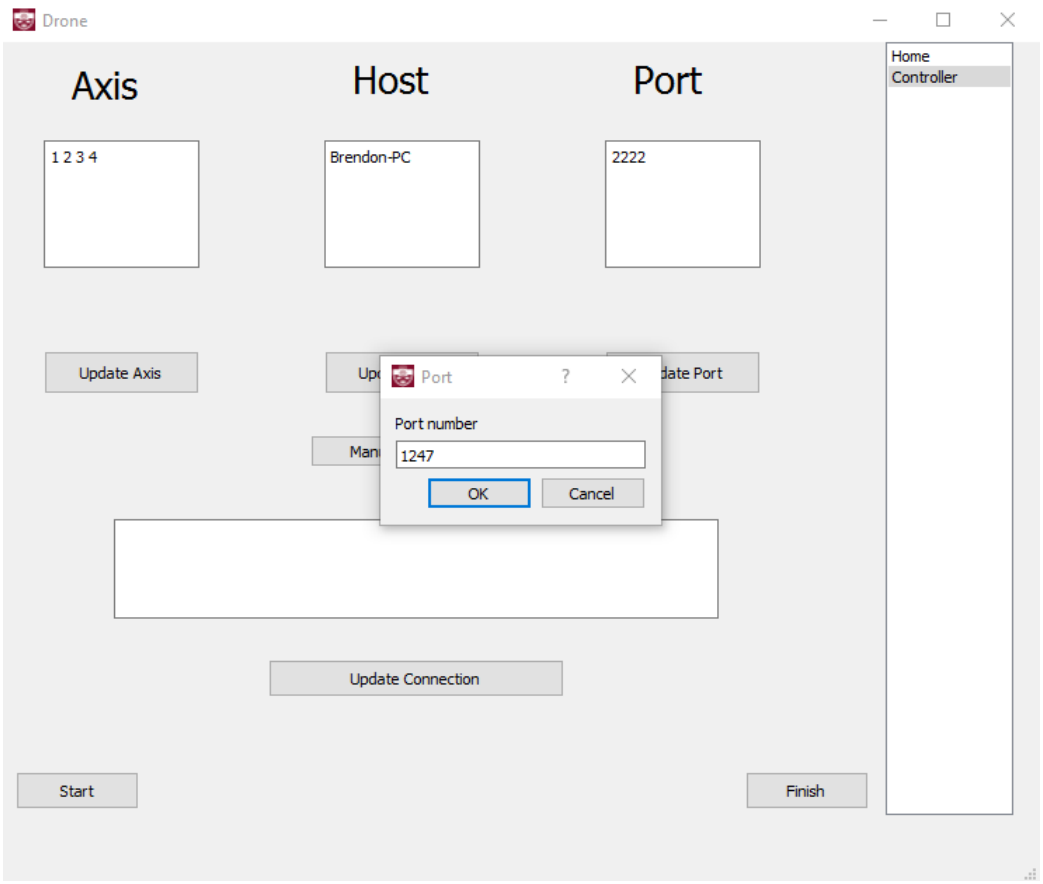


Figure 8: Pop up to input new values on GUI

As mentioned in section 2.1 Dr. Rhinelander has aspirations for the drone to partially or fully fly on it's own but in the event that something goes wrong there must be a way to manually control the drone. This is where the manual control button seen in Figure 5 comes into play. This push button calls the listen() definition of the PS4 Controller script found in APPENDIX. This initializes the PS4 Controller settings and then makes the connection to the socket on the Raspberry Pi 3 this will allow for control inputs to be sent. In order for this to be performed the Server script found in APPENDIX must be running on the Raspberry Pi 3 and the port numbers must match. The code for how this push button performs it's functions can be found in the connectController() definition on the GUI code found in APPENDIX.

4.3.5 Live Plotting

As discussed in section 3.3.4 the libraries matplotlib and pyqtgraph were considered when trying to determine how to display the live plots on the GUI. When it came to just displayed a simple graph matplotlib integrated seemlessly into the GUI and looked clean. However, when it came time to update the plot a big problem was encountered this was whenever an update timer was connected to the plotting definition the instance of the plot window would repeat itself rather than pulling the data and updating the plot with the new values. Upon researching solutions to this issue it was discovered that when dealing with real time plots matplotlib has a real weakness but that pyqtgraph excels in this aspect. With these reasons combined it was decided that moving forward all live plotting will be performed using pyqtgraph.

When using pyqtgraph live plotting became much easier to achieve. To achieve live plotting three definitions are required to set up the initial plot, what to do when the plot updates and which direction to move the plot. These definitions can be found in the initplt(), update1() and move() definitions in the GUI code in APPENDIX. Once these definitions are defined a QTimer must be connected to the move() definitions and an update interval will be defined the external dependancies and QTimer setup can be found under "Live Plotting" in the GUI code in APPENDIX. How the plot will look on the GUI can be seen in Figure 9.

On top of pyqtgraph being easier to implement it also includes some useful features. The user is able to click on the plot window and drag in any direction. For example, if the user was interested in data that occurred at time = 0 seconds but they were 100 seconds into data collection they would be able to click and drag back to the desired point to see the data of interest without disrupting data collection. If the user left clicks they are also to instantly obtain the maximum and minimum X and Y values, or in our case maximum time and amplitude values an example of the can be viewed in Figure 10

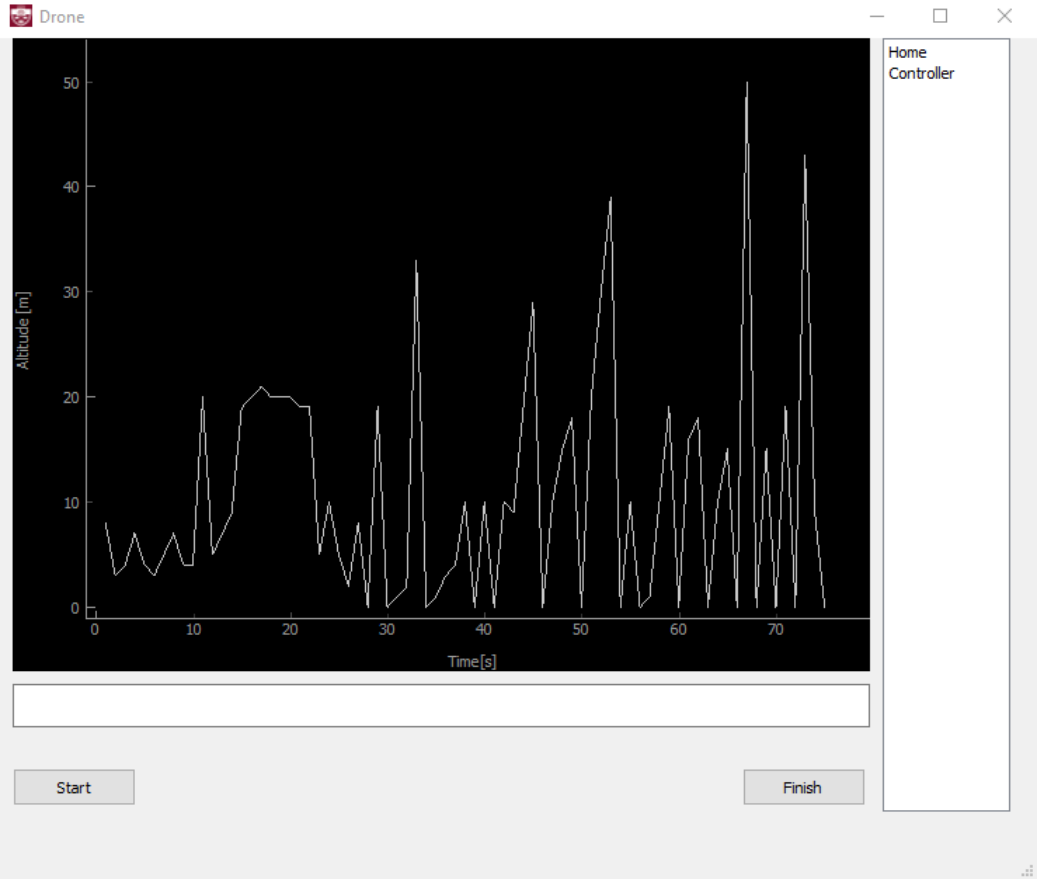


Figure 9: Live Plot

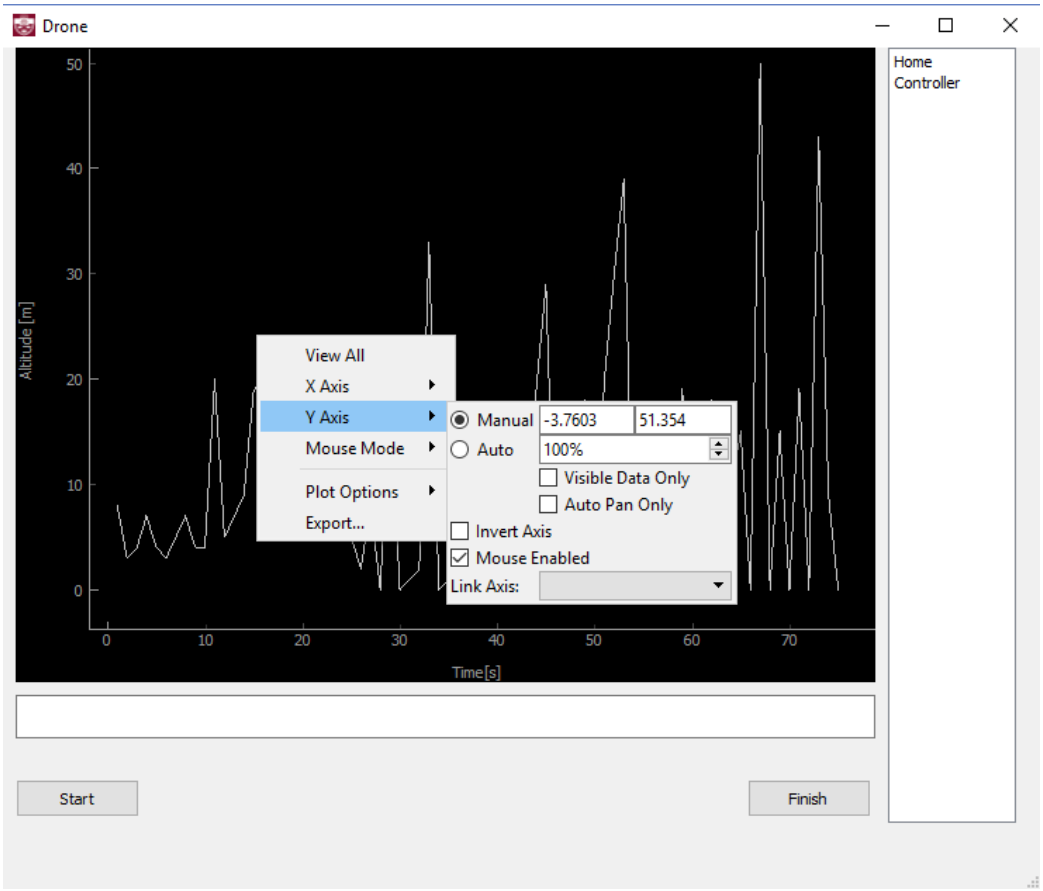


Figure 10: Display Y Values on Live Plot

5 Project Results

5.1 Simulations

5.2 Hover Test

Using the models and systems discussed in the previous subsection, our preliminary flight simulation was constructed. The SimuLink model Dynamic_Simulation.slx was created. The altitude controller was tested by providing a step input. After some tuning of the controller, we were able to obtain the following step response.

When this test was initially performed, the SimScape MultiBody physical sim-

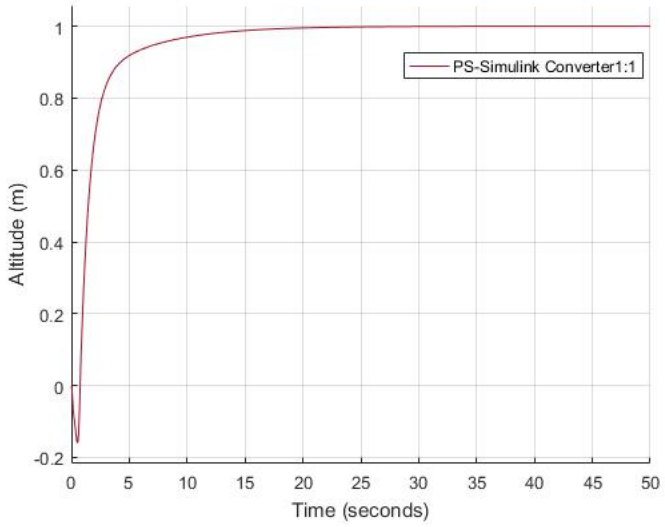


Figure 11: Hovering Step Response, for 1 metre altitude.

ulation did not yet include a model for a hard stop at the origin of the Z axis. This is to say that there was no representation of ground for the quad-rotor drone to rest upon. This is why an initial decline in position can be seen in the graph. This was subsequently added this term.

Further tests to be performed during the week will outlined in Section 6.1.

5.3 Physical Implementation

To date, several tests and proof of concepts have been performed to determine methods of controlling hardware, limitations of the hardware or software and to determine feasibility of communication protocols. The test subjects include:

- Electronic Speed Controller
- Motor Lift Characteristics
- Wi-Fi Range
- Communication Channel
- Inertial Measurement Unit
- Miscellaneous Code Testing

5.3.1 Electronic Speed Controller

The Afro ESC 12Amp BEC UltraLite Multirotor ESC V3 was tested at St. Mary's University with the assistance of Dr. Rhinelander. An Arduino running a script to map a potentiometer to a PWM duty cycle was used as an attempt to control the motor through the ESC. A 12V power supply fed the ESC while the Arduino controlled the duty cycle of the speed controller producing a voltage output to control the motor speed.

The ESC testing was successful, the testing proved the Arduino's capability to control the motor with a variable input. The testing had flaws as a PWM duty cycle was used instead of a timed pulse width input. The duty cycle had potential of operating correctly as the range of times could have been calibrated to a range in the duty cycle although this method proved to be difficult due to low values causing the ESC to enter calibration mode. The script used to operate the ESC was re-written as a timed pulse width to ensure complete compatibility and ease of future integration.The pulse width script was tested using the ESC and was successful. The provided motor was successfully driven under no load conditions for the full range of pulse width values.

5.3.2 Motor Lift Characteristics

Using a 12V power supply, the provided ESC and the Multi-Star Elite motor the characteristics of the motor's lifting capacity were tested. A weight was attached to a support system with the motor and blade seated on top. The apparatus was placed on a scale and the scale's reading was zeroed. As the rotor speed increased, the reduction in weight read by the scale was considered the lift capacity.

The test was performed beginning at a pulse width of 1127μs which was found through experimentation to be the cut in pulse width for motor operation. The test was performed at increments of 25μs. The resultant current draw values and lift values were documented. The lift values were used to create a simulink lookup table to characterize the motor's available force for simulation purposes.

During the load testing, it was noticed that the current draw from the individual motor was high. The power supply being used had a current limit of 3A therefore the maximum current draw allowed during testing was 2.95A to ensure no brown out due to lack of supply. The current limiting factor resulted in the test ending at a pulse width of 1525μs and a lift value of 137.5g.The resulting plot of the load testing is found in Figure 12.

H

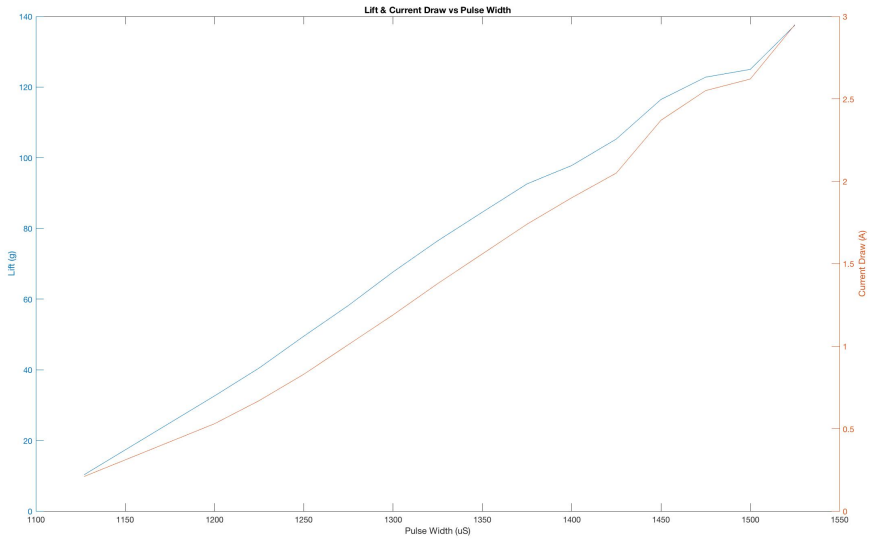


Figure 12: Motor Characterization Testing

5.3.3 Wi-Fi and Bluetooth Range

A simple proof of concept regarding the range of Wi-Fi communications was performed. The test incorporated a a Wi-Fi communicating camera tethered to a Wi-Fi output from a cell phone. A user walked down Spring Garden holding the cell phone and found the approximate distance at which the phone and the camera lost communication. It was found that the range was approximately 100 ft with line of sight available with no Wi-Fi boosting technology. Bluetooth communications were also tested using the Raspberry Pi 3 connected to a Playstation 4 controller although the communication channel held a strong connection for only approximately 10 ft, this distance was considered insufficient for the scope of the project.

5.3.4 Communication Channel Testing

Several tests to ensure the communication channels were operating correctly were performed. Testing of each communicating device were performed while scripting but an overall test was performed to be sure data was being transmitted from the joystick control input, through the Wi-Fi connection, over serial to the Arduino and finally writing to the ESCs. The testing of the channel was performed by use of an LCD screen connected to the Arduino, with the values received by the Arduino being written to the screen. Most recent testing produced slightly unexpected results, as the joystick controller seemed to have the control inputs mapped incorrectly. The test was considered successful as the altitude input performed as expected. Remapping of control inputs is required.

5.3.5 Inertial Measurement Unit Testing

The IMU was implemented and tested thoroughly for yaw, pitch, roll and altitude values. Altitude calculations were determined to be approximately +/- 50 cm in error without any filtering implemented. With an averaging filter implemented for the initial barometric pressure reading and a Kalman filter with feedback, the error was reduced to +/- 30 cm. Further filtering and testing of the altitude readings is to be performed. At low altitudes, +/- 30 cm error is considered too large as the stabilization PID loop will have difficulties when attempting to take off and land. Other measurement devices for low altitudes are going to be implemented and tested.

5.3.6 Miscellaneous Testing

All functions and scripts were tested to ensure compilation was possible and the program behaves as expected. Several tests and adjustments will be performed to ensure the device operates as expected prior to the submission of the final report.

5.4 Graphical User Interface

The main validation for the GUI was to ensure that each of the push buttons do the proper thing as well as when a page is selected from the list widget that the proper page is displayed. The testing involved with this was to select the desired push button or page and ensure the expected outcome happens. Along this process many software bugs were encountered and fixed, these included invalid syntax, improper use of libraries and just wrong implementation. Through these tests much was learned and the result was a GUI that provides all of the required functionality and is intuitive. After these were validated testing the live plotting capability was next.

5.4.1 Live Plotting Testing

Due to the new drone regulations proper flight tests to receive sensor data to live plot the altitude was unable to be performed. Instead, to validate that the plot on the seen in Figure 10 would update when new sensor data was received a test was developed. The test was to pull in data from a text file that was given in the form of x and y coordinates and then add new data to the text file as the GUI was running and make sure that the plot updated only when new data was availble. The definition to pull in the data from the file can be found in the liveData() definition in the GUI code in APPENDIX.

The result of this test was positive, whenever new data when entered into the text file plot would update to include to new data point accurately and swiftly. A flaw with this test could be that when the sensor data is receieved it would not come in this form, in this case the script would have to be altered to allow for a different data form. This test proved that the plot can be updated in real time.

6 Discussion

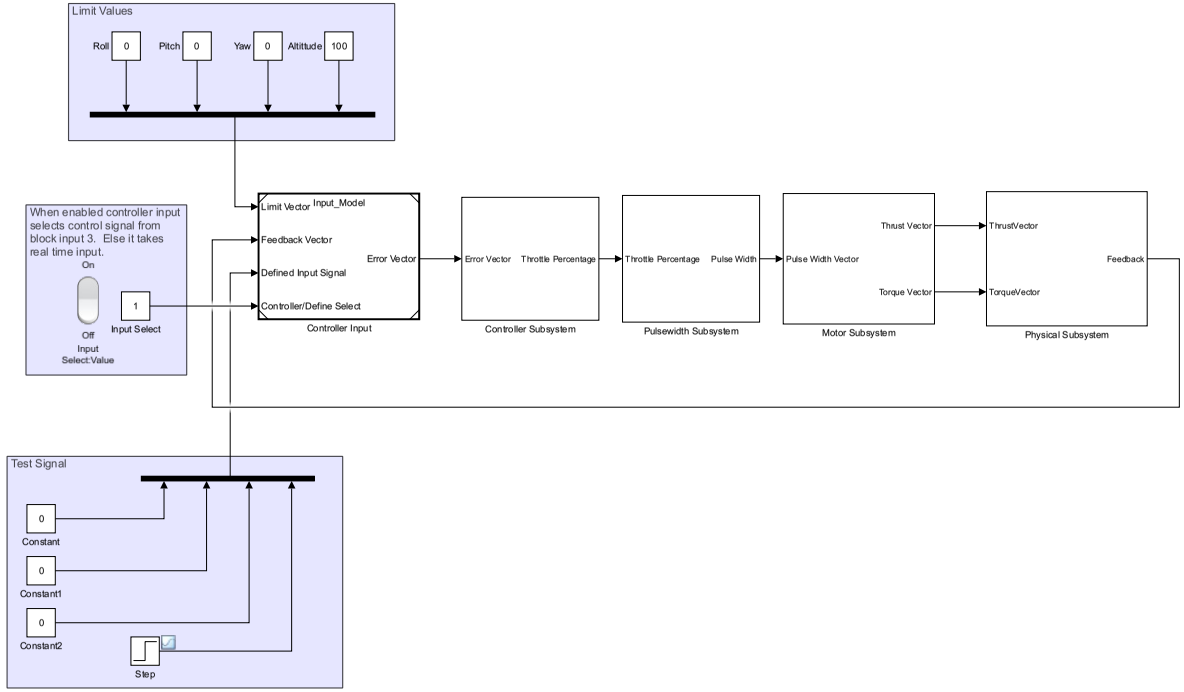
6.1 Simulations

6.2 Physical Implementation

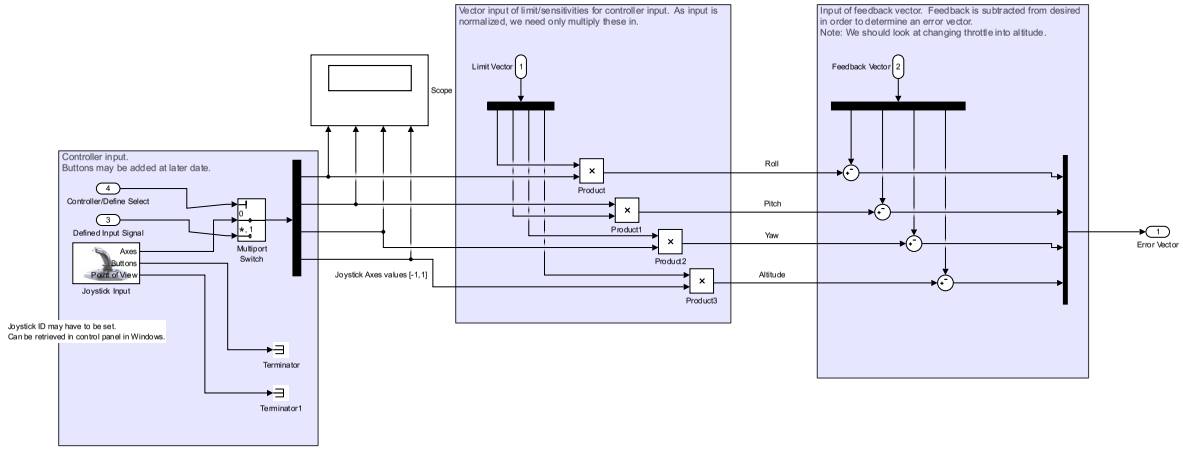
6.3 Graphical User Interface

A SimuLink Models

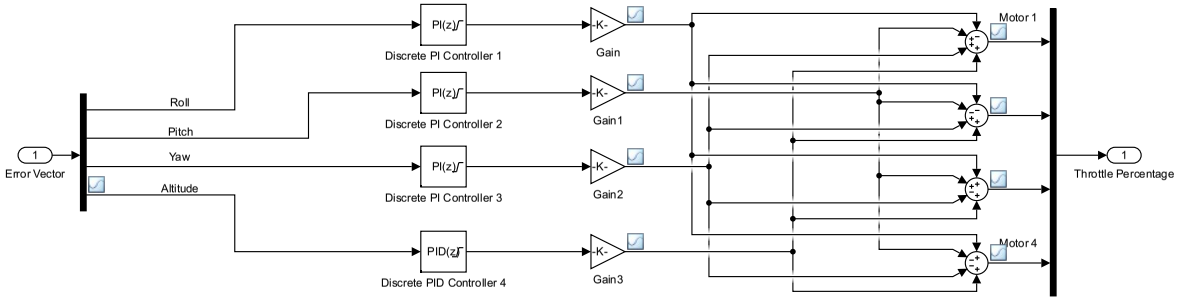
A.1 Dynamic Simulation



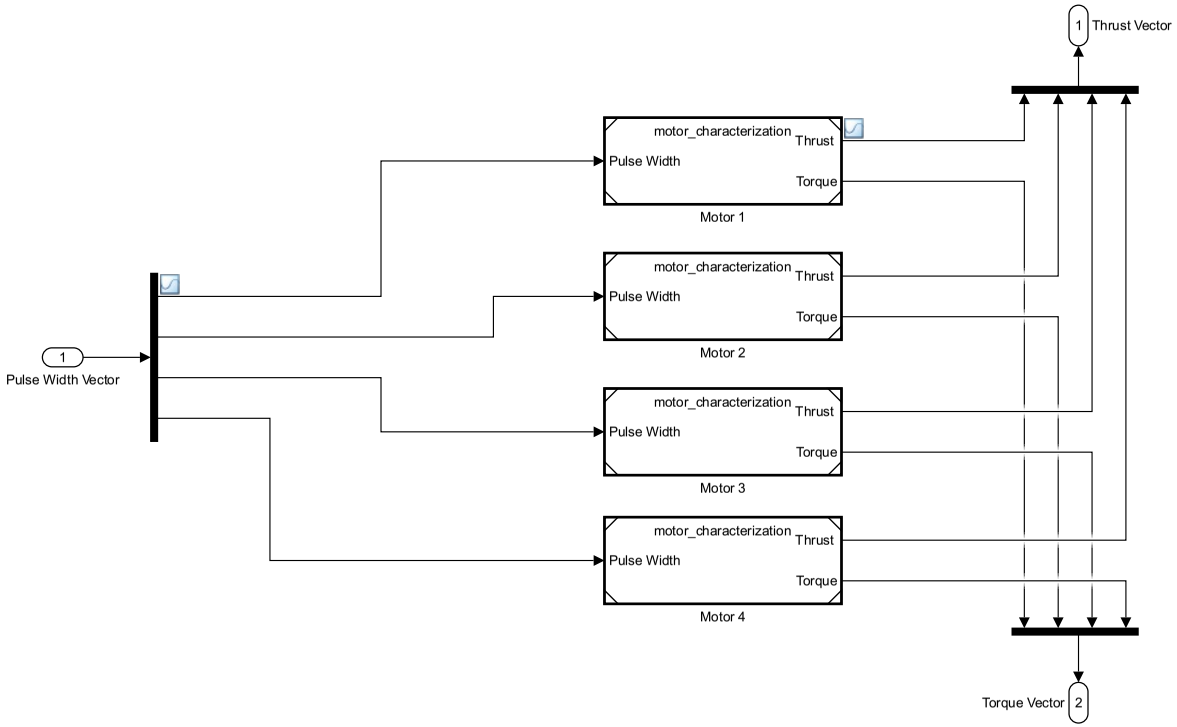
A.2 Input Model Subsystem



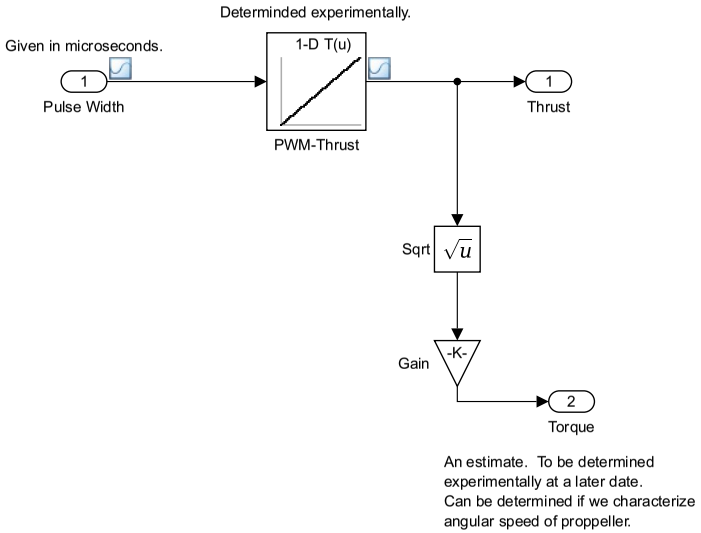
A.3 Controller Subsystem



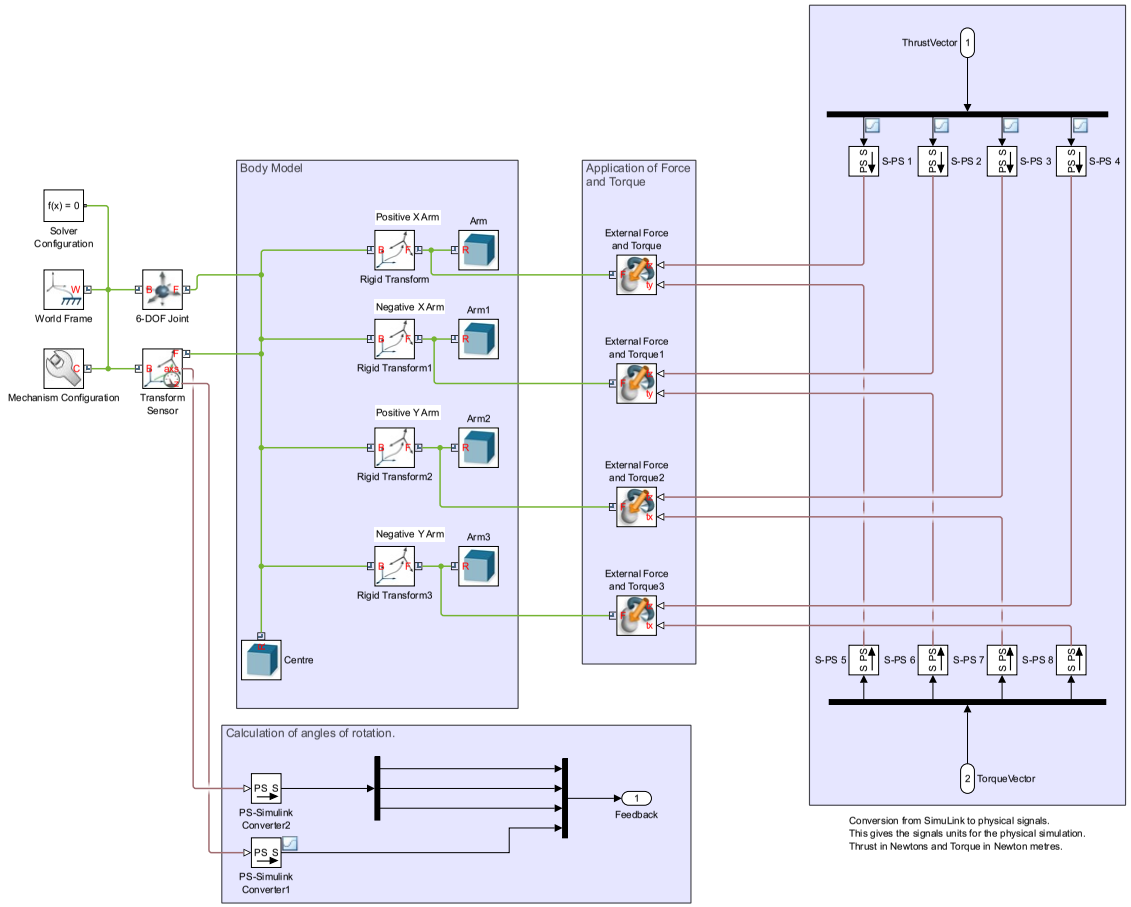
A.4 Motor Subsystem



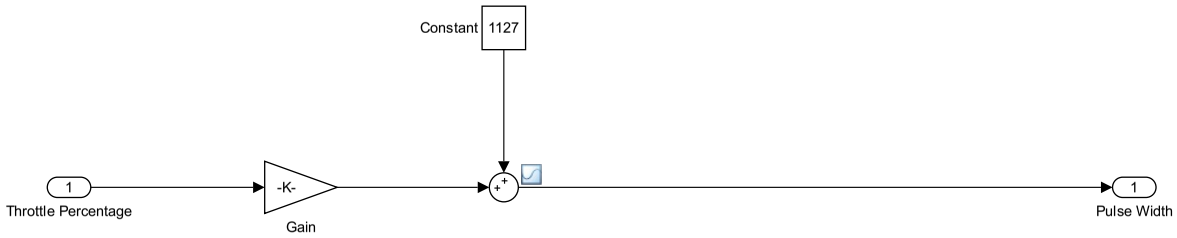
A.5 Motor Characterization Model



A.6 Physical Subsystem



A.7 Pulsewidth Subsystem



B Software Packages

B.1 Arduino Library

Listing 1: Drone.h Arduino Header File

```
/*
  Drone.h - Library for Drone Controller.
  Created by Lucas Doucette, February 17, 2017.
*/
#ifndef Drone_h
#define Drone_h

#include "Arduino.h"
#include <Wire.h>
#include <Servo.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_LSM303_U.h>
#include <Adafruit_BMP085_U.h>
#include <Adafruit_L3GD20_U.h>
#include <Adafruit_10DOF.h>

class Drone
{
  public:
    Drone();
    float get_sensorRoll();
    float get_sensorPitch();
    float get_sensorYaw();
    float get_sensorAltitude(float startingPressure);
    float get_currentPressure();
    float PID_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki );
    void read_PS4Setpoints();
    void initSensors();
    void initESCs(int MotorPin1, int MotorPin2,int MotorPin3,int MotorPin4);
    float read_float();
    void send_float();

};

#endif
```

Listing 2: Drone.cpp Arduino Library File

```
/*
  Drone.h - Library for Drone Controller.
  Created by Lucas Doucette, February 17, 2017.
*/

#include "Arduino.h"
#include "Drone.h"
#include <Wire.h>
#include <Servo.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_LSM303_U.h>
#include <Adafruit_LSM303.h>
#include <Adafruit_BMP085_U.h>
#include <Adafruit_L3GD20_U.h>
#include <Adafruit_10DOF.h>
#include <LiquidCrystal.h>


  /* Assign a unique ID to the sensors */
Adafruit_10DOF dof   = Adafruit_10DOF();
Adafruit_LSM303_Accel_Unified accel = Adafruit_LSM303_Accel_Unified(30301);
Adafruit_LSM303_Mag_Unified   mag   = Adafruit_LSM303_Mag_Unified(30302);
Adafruit_BMP085_Unified       bmp   = Adafruit_BMP085_Unified(18001);


sensors_event_t accel_event;
sensors_event_t mag_event;
sensors_event_t bmp_event;
sensors_vec_t   orientation;

float temperature;

Drone::Drone()
{

}

float Drone::get_sensorRoll()
{

    float sensorRoll;


  // Calculate pitch and roll from the raw accelerometer data

  accel.getEvent(&accel_event);
  if (dof.accelGetOrientation(&accel_event, &orientation))
  {
    // 'orientation' should have valid .roll and .pitch fields
    sensorRoll=orientation.roll;
    return sensorRoll;


  }
}

float Drone::get_sensorPitch()
{

  float sensorPitch;

  // Calculate pitch and roll from the raw accelerometer data

  accel.getEvent(&accel_event);
  if (dof.accelGetOrientation(&accel_event, &orientation))
  {
    // 'orientation' should have valid .roll and .pitch fields

     sensorPitch=orientation.pitch;
     return sensorPitch;

  }
}
```

```
float Drone::get_sensorYaw()
{

  float sensorYaw;

  // Calculate the heading using the magnetometer
  mag.getEvent(&mag_event);
  if (dof.magGetOrientation(SENSOR_AXIS_Z, &mag_event, &orientation))
  {
    // 'orientation' should have valid .heading data now
    sensorYaw=orientation.heading;
    return sensorYaw;

  }
}

float Drone::get_sensorAltitude(float startingPressure)
{

    float sensorAltitude, NowPressure;

    // kalman filtering variables

    float Variance=0.108923435;
    float varProcess = 1e-12;
    float Pc = 0.0;
    float G = 0.0;
    float P = 1.0;
    float Xp = 0.0;
    float Zp = 0.0;
    float Xe = 0.0;

    // Calculate the altitude using the barometric pressure sensor
    bmp.getEvent(&bmp_event);
    NowPressure=bmp_event.pressure;

    // Get ambient temperature in C
    bmp.getTemperature(&temperature);


    // Convert atmospheric pressure, SLP and temp to altitude
    sensorAltitude=bmp.pressureToAltitude(startingPressure,NowPressure,temperature);


    // kalman filter;
    Pc=P+varProcess;
    G=Pc/(Pc+Variance);
    P=(1-G)*Pc;
    Xp=Xe;
    Zp=Xp;
    Xe=G*(sensorAltitude-Zp)+Xp;


    //Create Deadzone (+/- 30cm)

     // if(Xe<0.3 && Xe>-0.3)
    //{
      //  Xe=0;
    //}

    return Xe;


}

//must be called after initSensors();

float Drone::get_currentPressure()
{
    float currentPressure;

   //Initializing Pressure, Filtered for accuracy
   //Smooting Constant

   const int numReadings=100;
   float SmoothingVariable=0;
   float NowPressure=0;
```

```
    bmp.getEvent(&bmp_event);
    for(int i=0; i<numReadings; i++)
    {
        SmoothingVariable+=bmp_event.pressure;
        //delay(1);

    }

    currentPressure = SmoothingVariable/numReadings;
    SmoothingVariable=0;

    return currentPressure;

}

float Drone::PID_Calculate(float Setpoint, float SenseRead, float kp, float kd, float ki)
{
    //Define PID Variables
    float Error, DError, IError, LastTime, LastError, Output;

    //find current time
    unsigned long NowTime = millis();

    //calculate PID Error values
    Error = Setpoint-SenseRead;
    DError = (Error-LastError)/(NowTime-LastTime);
    IError += (Error*(NowTime-LastTime));

    //Calculate Output
    Output = kp*Error+ki*IError+kd*DError;

    //Save LastTime and LastError
    LastTime=NowTime;
    LastError=Error;


    //Requirement based on Simulink

    Output=(Output/9000)*(1860-1127);

    if(Output>=1860){

        Output = 1860;

    }

    return Output;
}


void Drone::read_PS4Setpoints(float *PS4Yaw, float *PS4Pitch, float *PS4Roll, float *PS4Altitude)

{

    //Write serially to Pi to begin transmission.

    byte XMIT = 00000001;

    if (Serial.available())
    {
       Serial.write(XMIT);
    }



    while (Serial.available())
    {
        for(int i=0; i<4; i++)
        {
            *PS4Yaw=read_float();

        }

        for(int i=4; i<8; i++)
        {
            *PS4Pitch=read_float();
```

```
        }

        for(int i=8; i<12; i++)
        {
            *PS4Roll=read_float();

        }

        for(int i=12; i<16; i++)
        {
           *PS4Altitude=read_float();

        }

    }



}


void Drone::initSensors()
{

  if(!accel.begin())
  {
    /* There was a problem detecting the LSM303 ... check your connections */
    Serial.println(F("Ooops, no LSM303 detected ... Check your wiring!"));
    while(1);
  }
  if(!mag.begin())
  {
    /* There was a problem detecting the LSM303 ... check your connections */
    Serial.println("Ooops, no LSM303 detected ... Check your wiring!");
    while(1);
  }
  if(!bmp.begin())
  {
    /* There was a problem detecting the BMP180 ... check your connections */
    Serial.println("Ooops, no BMP180 detected ... Check your wiring!");
    while(1);
  }

}

void Drone::initESCs(Servo esc_1, Servo esc_2, Servo esc_3, Servo esc_4)
{


  //Initialization of the ESCs
  esc_1.writeMicroseconds(1860);
  esc_2.writeMicroseconds(1860);
  esc_3.writeMicroseconds(1860);
  esc_4.writeMicroseconds(1860);
  delay(3000);
  esc_1.writeMicroseconds(1060);
  esc_2.writeMicroseconds(1060);
  esc_3.writeMicroseconds(1060);
  esc_4.writeMicroseconds(1060);
  delay(3000);


}

void Drone::send_float (float arg) {
  byte * data = (byte *) &arg;
  Serial.write(data, sizeof(arg));
}

float Drone::read_float () {
 union{
   float a;
   unsigned char bytes[4];
 } data;
 for (int i=0; i<4; i++) {
```

```
    data.bytes[i] = Serial.read();
  }
  float test = data.a;
  return(test);
}
```

B.2 Raspberry Pi Scripts

Listing 3: Server.py Raspberry Pi Communication Server

```
import socketserver
import sys
import arduino
import threading
import queue
from subprocess import check_output

#TODO:
#   Control Side
#       Write Server w/ logging
#       Test
#   Backchannel
#       Write Handler
#       Test
#   Read
#       TCP Binding

class SerialRequestHandler(threading.Thread):
    def __init__(self, stateq):
        super(self.__class__, self).__init__()
        self.stateq = stateq

    def run(self):
        #TODO: Add poison pill to exit thread
        sbus = arduino.Arduino_Controller(9600)
        while True:
            ready = sbus.ready()
            if ready:
                sbus.serial_bus.read(ready)
                sbus.serial_bus.write(self.stateq.get())


class QuadControlHandler(socketserver.BaseRequestHandler):
    def __init__(self, request, client_address, server, stateq):
        self.stateq = stateq
        super(self.__class__, self).__init__(request, client_address, server)
        return

    def handle(self):
        if self.stateq.full():
            self.stateq.get() #Queue has size of 1, if full clear for new state
        recv = self.request.recv(16)
        self.stateq.put(recv)
        print('Received: ' + str(recv))
        return


class QuadControlServer(socketserver.TCPServer):
    def __init__(self, server_address, RequestHandlerClass):
        super(self.__class__, self).__init__(server_address, RequestHandlerClass)
        self.controller = arduino.Arduino_Controller(9600)
        self.stateq = queue.Queue(1)
        return

    def serve_forever(self, poll_interval=0.5):
        thread = SerialRequestHandler(self.stateq)
        thread.start()
        super(self.__class__, self).serve_forever(poll_interval)
        return

    def finish_request(self, request, client_address):
        self.RequestHandlerClass(request, client_address, self, self.stateq)



if __name__ == '__main__':
    if sys.version_info[0] < 3:
        raise Exception('Version Error', 'Not compatible with Python version 2')

    HOST = check_output(['hostname', '-I']).strip()
```

```
CONTROL_PORT = 2222
COMMS_PORT = 4444

test_serv = QuadControlServer((HOST, CONTROL_PORT), QuadControlHandler)
test_serv.serve_forever()
```

Listing 4: arduino.py Raspberry Pi Serial Communication Library

```
# 2017-01-13 Auth: Dylan

import serial

class Arduino_Controller(object):
    #  Provides a wrapper for communicating with the Arduino over I2C/SMBUS.
    def __init__(self, baud):
        self.baudrate = baud
        self.serial_bus = serial.Serial('/dev/ttyACM0', self.baudrate)

    def write_axes(self, axes):
        self.serial_bus.write(axes)

    def ready(self):
        return self.serial_bus.inWaiting()

    def write_button(self, button):
        pass
```

B.3 Base Station Communication Script

Listing 5: Control Signal and Client Script

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
#
# This file presents an interface for interacting with the Playstation 4 Controller
# in Python. Simply plug your PS4 controller into your computer using USB and run this
# script!
#
# NOTE: I assume in this script that the only joystick plugged in is the PS4 controller.
#       if this is not the case, you will need to change the class accordingly.
#
# Copyright   2015 Clay L. McLeod <clay.l.mcleod@gmail.com>
#
# Distributed under terms of the MIT license.

#TODO:
#   rewrite connection for new server
#   test

# import os
# import pprint
import pygame
import socket
import struct
import sys

if sys.version_info[0] < 3:
    raise Exception('Lucas', 'not compatible with Python version 2')


class PS4Controller(object):
    """Class representing the PS4 controller. Pretty straightforward functionality."""

    controller = None
    axis_data = None
    button_data = None
    hat_data = None

    def __init__(self, axis_order=[1, 2, 3, 4], hostname='raspberrypi', port=2222):
        """Initialize the joystick components"""

        pygame.init()
        pygame.joystick.init()
        self.controller = pygame.joystick.Joystick(0)
        self.controller.init()
        self.hostname = hostname
        self.port = port
        if isinstance(axis_order, list):
            self.axis_order = axis_order  # For changing how controller axes are bound
        else:
            raise Exception(TypeError, 'axis_order must be list.')

    def update_axes(self, axis_order):
        self.axis_order = axis_order

    def listen(self):
        """Listen for events to happen"""

        if not self.axis_data:
            self.axis_data = {0: float(0),
                              1: float(0),
                              2: float(0),
                              3: float(0),
                              4: float(-1),
                              5: float(-1)}  # Added explicity number of axes to avoid waiting for input

        if not self.button_data:
            self.button_data = {}
            for i in range(self.controller.get_numbuttons()):
                self.button_data[i] = False

        if not self.hat_data:
```

```
        self.hat_data = {}
        for i in range(self.controller.get_numhats()):
            self.hat_data[i] = (0, 0)


    # host = '192.168.2.19' #ip of Server (PI)
    host = socket.gethostbyname(self.hostname)  # if fails install samba on pi and reboot

    while True:
        for event in pygame.event.get():
            if event.type == pygame.JOYAXISMOTION:
                self.axis_data[event.axis] = round(event.value, 2)
            elif event.type == pygame.JOYBUTTONDOWN:
                self.button_data[event.button] = True
            elif event.type == pygame.JOYBUTTONUP:
                self.button_data[event.button] = False
            elif event.type == pygame.JOYHATMOTION:
                self.hat_data[event.hat] = event.value

            # Insert your code on what you would like to happen for each event here!
            # In the current setup, I have the state simply printing out to the screen.

            # Defining Variables to send through the socket to the RPi, need to be strings

            # axis_data=str(self.axis_data)
            # button_data = str(self.button_data)
            # hat_data = str(self.hat_data)

            # Sending Data over a socket to the RPi
            # print(str(self.axis_data))
            # Isolate desired Axes

            axes_data = [self.axis_data[self.axis_order[0]],
                         self.axis_data[self.axis_order[1]],
                         self.axis_data[self.axis_order[2]],
                         self.axis_data[self.axis_order[3]]]
            byte_data = []  # To hold the axes data serialized to bytes
            for axis in axes_data:
                byte_data.append(struct.pack("f", axis))  # F for float

            xmission_bytes = bytes().join(byte_data)
            connection = socket.socket()
            connection.connect((host, self.port))
            connection.send(xmission_bytes)  # sending the controller data over the port
            connection.close()
            # print(xmission_bytes)

            # os.system('cls')
            # break
            # s.send(button_data)
            # s.send(hat_data)
            # s.close()


if __name__ == "__main__":
    ps4 = PS4Controller()
    # ps4.init()
    ps4.listen()
```

C GUI

C.1 GUI Code

Listing 6: GUI.py

```
#Written by Brendon Camm, last updated March 27th, 2017

import sys
from PyQt5 import QtCore, QtGui, uic, QtWidgets
import matplotlib
from matplotlib.backends.backend_qt5agg import (FigureCanvasQTAgg as FigureCanvas,
NavigationToolbar2QT as NavigationToolBar)
from matplotlib.figure import Figure
import numpy as np
import time
import struct
import socket
from timeit import default_timer as timer
from PS4_Controller import PS4Controller as PS4
import threading
import logging
import matplotlib.animation as animation
import pyqtgraph as pg


qtCreatorFile = "GUI.ui" # Enter file here.
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)



class GUI(QtWidgets.QMainWindow, Ui_MainWindow, QtWidgets.QMenu):
def __init__(self):

super(GUI, self).__init__()
#Qt initialization
QtWidgets.QMainWindow.__init__(self)
Ui_MainWindow.__init__(self)
self.page = QtWidgets.QStackedWidget()
self.setCentralWidget(self.page)
self.setupUi(self)
self.setWindowTitle("Drone")
self.setWindowIcon(QtGui.QIcon('smu.png'))

#Networking
self.getHost = socket.gethostname()
self.staticPort = '1247'
#Main Page buttons
self.start.clicked.connect(self.connection)
self.end.clicked.connect(self.stop)

#Listing widget
self.list.insertItem(0, 'Home')
self.list.insertItem(1, 'Controller')
self.list.currentRowChanged.connect(self.display)

#Controller Page
self.axisVal.setText('1_2_3_4')
self.hostVal.setText(self.getHost)
self.portVal.setText('2222')
self.axisMenu.clicked.connect(self.axisSettings)
self.hostMenu.clicked.connect(self.hostSettings)
self.portMenu.clicked.connect(self.portSettings)
self.updateConnect.clicked.connect(self.updateConnection)
self.connectPS4.clicked.connect(self.connectController)

#Live Plotting
self.initplt()
self.plotcurve = pg.PlotDataItem()
self.plotwidget.addItem(self.plotcurve)
self.t = 0
self.amplitude = 10
self.update1()
```

```
self.timer = pg.QtCore.QTimer()
self.timer.timeout.connect(self.move)
self.timer.start(1000)

def stop(self):
sys.exit(app.exec_())
def connection(self):
s = socket.socket()
host = self.getHost
port = int(self.staticPort)
status = s.connect_ex((host,port)) #Returns 0 if true
if status: # Status = errno
self.thisworks.setText("Connection Unsuccessful")
self.connectionStat.setText("Communications have not been established")
else: # Status = 0
print(status)
self.thisworks.setText("Connection Successful")
self.connectionStat.setText("Communications are active")
def axisSettings(self):
cont = PS4()
text, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[0]', 'No Spaces')
text1, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[1]', 'No Spaces')
text2, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[2]', 'No Spaces')
text3, ok = QtWidgets.QInputDialog.getText(self, 'Axis Value[3]', 'No Spaces')
axis = [int(text), int(text1), int(text2), int(text3)]
self.axisVal.setText(str(axis))
cont.axis_order = axis
return cont.axis_order
def display(self,i):
self.home.setCurrentIndex(i)
def addData(self,name,fig):
self.fig_duct[name] = fig
self.list.addItem(name)
def hostSettings(self):
text, ok = QtWidgets.QInputDialog.getText(self,'Host', 'Host name or IP address')
newHost = str(text)
if newHost == '':
self.hostVal.setText(self.getHost)
else:
self.hostVal.setText(newHost)
return newHost
def portSettings(self):
text, ok = QtWidgets.QInputDialog.getText(self,'Port','Port number')
if ok:
print('success')
newPort = str(text)
if newPort == '':
self.portVal.setText(self.staticPort)
else:
self.portVal.setText(newPort)
else:
self.display(1)

return int(newPort)
def connectController(self):
new = PS4()
new.listen()
def updateConnection(self):
host1 = self.hostSettings()
port1 = self.portSettings()
s = socket.socket()
status = s.connect_ex((host1,port1))

if status:
self.connectionStat.setText("Update and Connection Unsuccessful")
self.thisworks.setText("Update and Connection Unsuccessful")
#socket.send('Success')
else:
self.connectionStat.setText("Update and Connection Successful")
self.thisworks.setText("Update and Connection Successful")

def liveData(self):
graph_data = open('test.txt', 'r').read()
lines = graph_data.split('\n')
xs = []
ys = []
for line in lines:
```

```
if len(line)>1:
x, y = line.split(',')
xs.append(int(x))
ys.append(int(y))
return xs,ys


def initplt(self):
self.plotwidget = pg.PlotWidget()
self.mplvl.addWidget(self.plotwidget)
self.plotwidget.setLabel('left', 'Altitude [m]')
self.plotwidget.setLabel('bottom','Time[s]')
self.show()
def update1(self):
list1,list2 = self.liveData()
self.plotcurve.setData(list1,list2)
def move(self):
self.t+=1
self.update1()


if __name__ == '__main__':
app = QtWidgets.QApplication(sys.argv)
main = GUI()
main.show()
#cont = PS4()
#print(str(cont.axis_order))
QtWidgets.QApplication.processEvents()
sys.exit(app.exec_())
```

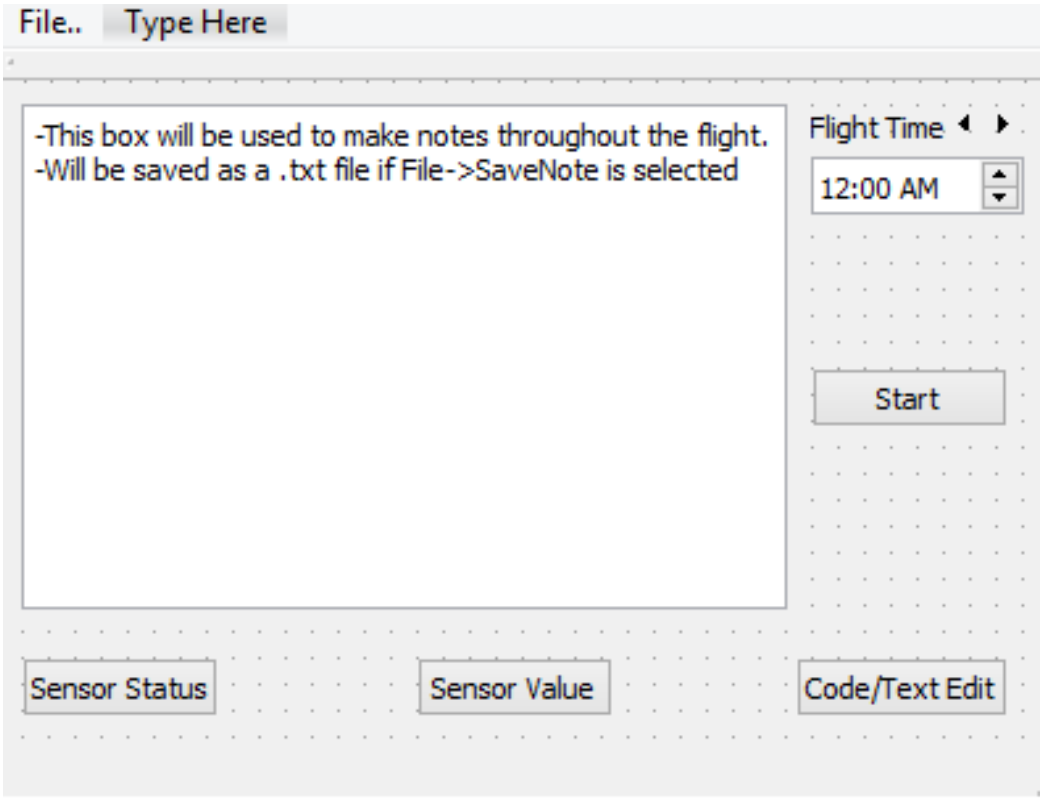


Figure C.1: Original GUI Prototype Home Page

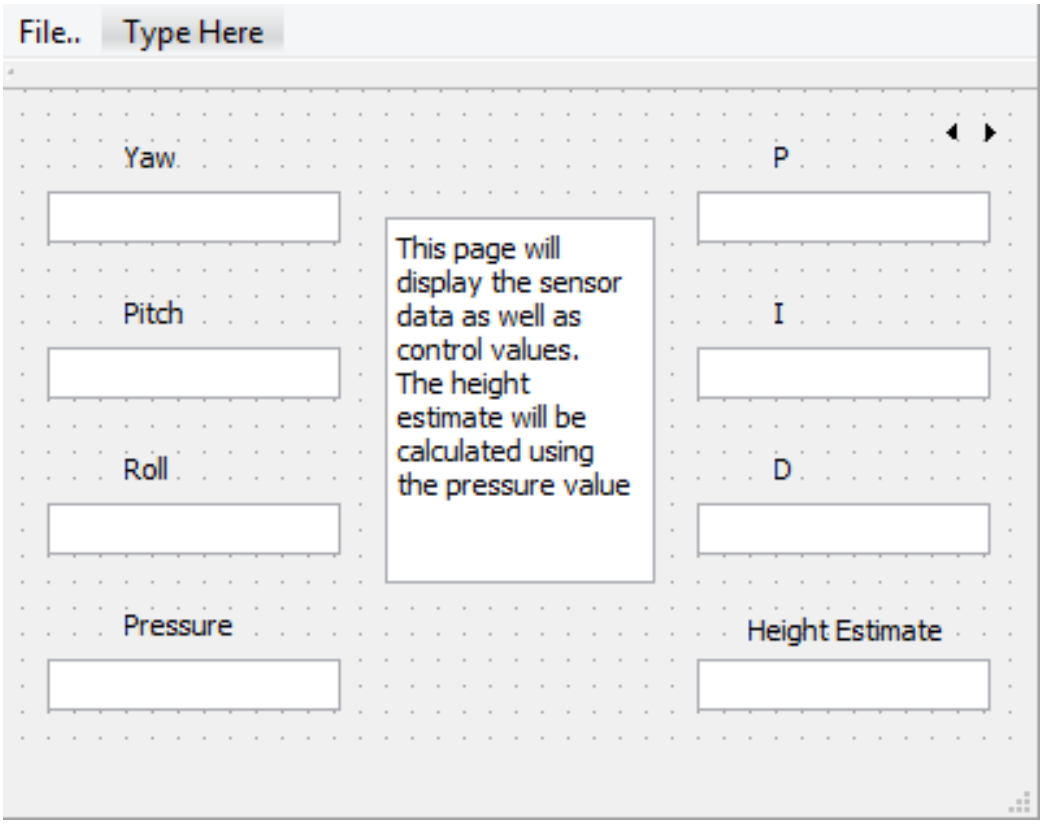


Figure C.2: Original GUI Prototype 2nd Page