Building An Autonomous Indoor Drone System

Hunter Gossett

University of Mississippi

Follow this and additional works at: https://egrove.olemiss.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
https://egrove.olemiss.edu/etd/984

This Thesis is brought to you for free and open access by the Graduate School at eGrove. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of eGrove. For more information, please contact egrove@olemiss.edu.
BUILDING AN AUTONOMOUS INDOOR DRONE SYSTEM

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by
Hunter Gossett
May 2018
This thesis presents an Indoor Autonomous Drone System using a self assembled drone which uses a companion computer as well as external sensors for autonomous flight. While autonomous drone systems have been around for some time, originating with the military, the vast majority of them are designed for outdoor use, because of their heavy reliance on GPS for their positioning systems. In order to achieve autonomous flight indoors we choose to use Simultaneous Localization and Mapping (SLAM) as our positioning system. The contributions of this thesis is an in depth guide to the hardware and assembly of a drone system, the software required for a basic drone set-up as well as additional software needed for autonomous operations, and finally the results of the project along with the difficulties and set backs that were encountered.
ACKNOWLEDGEMENTS

I would like to express my appreciation to my advisory Dr. Byunghyun Jang for his continuous support of my graduate study and research. I thank him for giving me an opportunity to be part of the HEROES research team and work under his supervision. I thank him for his great support, encouragement, suggestions and insightful comments on my research. On a personal note, Dr. Jang inspired me by his hardworking, passionate attitude and caring to all of his students. It was my honor to work with Dr. Jang.

Finally, many thanks go to the members of HEROES research team. I thank them for their continuous support and friendly suggestions. It was my great pleasure to work with them.
TABLE OF CONTENTS

ABSTRACT ........................................................................ iii

ACKNOWLEDGEMENTS ................................................... iii

LIST OF FIGURES ............................................................ vi

INTRODUCTION ............................................................... 1

TECHNICAL BACKGROUND ................................................. 4
  2.1 Basics of Quadcopter Technology ................................. 4
  2.2 Basics of SLAM ......................................................... 9

HARDWARE AND ASSEMBLY OF BASIC DRONE ............... 15
  3.1 The Frame ............................................................... 15
  3.2 The Flight Controller ................................................. 16
  3.3 Transmitter and Receiver .......................................... 17
  3.4 Batteries ................................................................. 18
  3.5 Assembly ................................................................. 18
  3.6 Additional Hardware .............................................. 19

SOFTWARE STACK ........................................................... 27
  4.1 Flight Controller Firmware ........................................ 27
  4.2 Raspberry Pi’s Operating System ................................. 29
  4.3 DroneKit-Python ..................................................... 30
  4.4 Software to Run SLAM ............................................. 31
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Quadcopter Basic Components[15]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Quadcopter Motor Rotation[5]</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Movements on Quadcopter’s Axes[5]</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>The essential SLAM problem[13]</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>DJI Flamewheel F450 ARF Kit[1]</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>3DR Pixhawk Mini Flight Controller[2]</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>RC Transmitter and Receiver</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Typical Quadcopter Wiring Layout[5]</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>Wiring between RPI and Pixhawk[6]</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>HC-SR04 Ultrasonic Range Finder Sensor[8]</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>A Basic Voltage Divider Circuit[8]</td>
<td>23</td>
</tr>
<tr>
<td>3.8</td>
<td>Equation for Voltage Divider Circuit</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>Connections between Ultrasonic Sensor and Raspberry Pi[8]</td>
<td>24</td>
</tr>
<tr>
<td>3.10</td>
<td>RPLidar A2 Laser Range Scanner[20]</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>A Map Built using Hector SLAM</td>
<td>41</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Drones, also known as unmanned aerial vehicles (UAV), are pilotless and non-crewed aircraft that are capable of flight either by remote control or through the use of on-board computers. Drones are commonly used by the military, but are also being implemented in search and rescue operations and being utilized in other civil applications, such as policing and firefighting. The technology is also allowing for hobbyists and other enthusiasts to become avid drone operators. While most hobbyists and other enthusiasts purchase drone systems for the enjoyment of flying the craft around the yard, taking nice aerial videos, or for competing in First Person View (FPV) racing, most of the civil applications such as search and rescue or other innovative applications benefit from some form of computer aided autonomous or automated flight. Current commercial UAVs are capable of some basic autonomous flight, where the operator can set a “flight plan” within a control center application and have the UAV travel from one GPS way-point to another then return home or travel to a GPS way-point and take a 360° video then return home. However all current autonomous or automated systems require the use of GPS as its positioning system and are meant for outdoor use only. Since GPS is at best error prone and at worst non-functioning indoors, we propose using SLAM as our indoor positioning system for indoor autonomous flight.

Upon research I found several approaches for resolving indoor positioning within
robotics. Many approaches used either Bluetooth or RF beacons positioned throughout the environment and used them as anchor nodes to determine location. Another approach uses motion tracking cameras positioned on the ceiling to track and locate the robot. With the exception of a few approaches, most rely on additional hardware and additional setup to implement the system, such as in the beacons and motion tracking cameras [16]. While this seems to work well in small known environments, these approaches would not scale well for large implementations or unknown environments. For this reason we choose to implement the SLAM approach. In robotic mapping and navigation, Simultaneous Localization and Mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of a robot’s location within it.

In order to achieve autonomous operations with our system, we propose using Ardupilot firmware running on the drones onboard flight controller, as well as DroneKit-Python which runs on an onboard companion computer. Ardupilot is an open source autopilot software capable of controlling many different vehicle systems, such as conventional airplanes, multirotors, helicopters, boats, and even submarines. With Ardupilot being open source there are many peripheral suppliers creating interfaces, allowing for the use of many different sensors, companion computers and communication systems. For this reason we are able to use DroneKit-Python to control Ardupilot via a low-latency link from an onboard companion computer. DroneKit-Python allows developers to create apps that can significantly enhance the autopilot, adding greater intelligence to vehicle behaviour, and performing tasks that are computationally intensive or time-sensitive. Through DroneKit-Python we will be able to perform obstacle avoidance and navigation with our system.

This thesis is organized as follows. Chapter 2 provides some necessary backgrounds into how basic quadcopter drone systems work, and the basics of SLAM. Chapter 3 presents an in depth guide on the hardware and assembly of the drone system we built. Chapter 4 presents the entire software stack that is used within our drone system. Chapter 5 discusses our approach to autonomous operations. Chapter 6 discusses the experimental results as
well as the difficulties with real world experimentation of our system. Finally we conclude the thesis and discuss future plans in Chapter 7.
2.1 Basics of Quadcopter Technology

A quadcopter is essentially a helicopter which has motors that are equally spaced, arranged on the corners of a square (X) body. Unlike helicopters, multirotors are inherently very unstable without electronic assistance because it is extremely herculean to control multiple rotors manually. The decrease in cost of modern microprocessors lead to these machines getting popular in the recent years [15].

2.1.1 Basic Components of a Multirotor

A visual representation of the basic components is shown in Figure 2.1.

- The Body/Frame
  The frame or body is what holds everything together. They are generally designed to be strong and lightweight and consist of a center plate where the main flight controller chip and sensors are mounted and arms where the motors are mounted.

- Motors
  The motivation behind using motors is to turn the propellers, which are responsible
for providing thrust for countering gravity and drag. Each motor, depending on the style of frame, should be controlled separately by a electronic speed controller. The brushless motors, which are the style motors most quadcopters use, are rated using kilovolts. The $kV$ rating in a motor demonstrates how various RPMs (Revolutions per Minute) the motor will do per volt. The higher the $kV$ rating is the quicker the motors rotate at a steady voltage.

- **Electronic Speed Controllers (ESCs)**

  Since each motor on the quadcopter must spin at various speeds, each motor has its own ESC. The ESC is what tells the motor how to spin or more specifically how fast or slow to spin. The ESCs are typically connected to the batteries via a power distribution board within the multirotor’s frame, and receive an input signal from the flight controller.

- **Flight Controller**

  The flight controller is the mind or ‘brains’ of the multirotor. This board is what sits at the center, controlling the firmware within the ESCs which controls the speed at which each motor spins. The flight controller takes the inputs from a receiver or an onboard
companion computer and adjusts the motor RPM accordingly. Flight controller systems also house additional sensors to enhance control and stability of the craft. Most flight controllers contain a gyroscope, an accelerometer, and a barometer. A GPS module is usually added externally to provide coordinate and altitude information.

- **Transmitter and Receiver**
  The radio transmitter and receiver are used to control the quadcopter. At a minimum for a quadcopter to work, four channels (Throttle, Yaw, Roll, Pitch) are required. However, most transmitters and receivers have more than the required four channels allowing for additional functionality, such as switching between flight modes or controlling an onboard camera gimbal.

- **Batteries**
  Lithium Polymer (LiPo) batteries seem to be the most preferred power source for powering multirotors currently. However, for larger quadcopters even the LiPo batteries don’t allow for extended flight times.

### 2.1.2 How Quadcopters Fly

In a quadcopter there are four motors placed at the edge of four arms of the frame. The direction of each motor’s rotation is such that it counteracts the torque generated by the motor that is placed at the opposite side as shown in Figure 2.2. This is how the quadcopter keeps from spinning due to torque effect. Since the motors are spinning the propellers in two different directions, there are two different types of propellers. For the motors that spin clockwise there are pusher propellers, which push the air down in order to generate thrust. Likewise for the counterclockwise motors there are puller propellers, which pull the air down in order to generate thrust.

- **Hover**
  In order for a quadcopter to hover in place, it requires that:
Figure 2.2. Quadcopter Motor Rotation[5]

Figure 2.3. Movements on Quadcopter’s Axes[5]
1. All the motors rotate at the same speed.
2. The rotation speed must be sufficient enough to generate lift which counteracts the weight of the quadcopter.
3. The torque effect acting on the body of the quadcopter by each motor should cancel out.

• **Gaining and Losing Altitude**
  In order for the quadcopter to gain altitude, all four of the motors must increase the speed of rotation **simultaneously**. Likewise to lose altitude all four of the motors must decrease speed of rotation **simultaneously**.

• **Pitch**
  The pitch control tells the quadcopter to fly forward or backward. In order to pitch forward, the speed of the motors at the rear of the quadcopter must increase relative to the speed of the motors on the front. This pitches the nose of the quadcopter down making the quadcopter move forward. Likewise to pitch backwards, the speed of the front motors must increase relative to the speed of the back motors.

• **Roll**
  The roll control tells the quadcopter to move side to side. In order to roll the quadcopter to the right, the speed of the motors at the left must increase relative to the motors on the right. This rolls the quadcopter to the right resulting in side-ways movement to the right. Likewise to roll the quadcopter to the left, the motors on the right of the quadcopter must increase relative to the motors on the left. This rolls the quadcopter to the left, resulting in a side-ways movement to the left.

• **Yaw**
  The yaw is the rotational movement of the quadcopter along the z-axis. This is accomplished by increasing or decreasing two motors that spin in the same direction.
This results in the quadcopter rotating in the direction of the increased torque. An illustration of pitch, roll, and yaw can be seen in Figure 2.3.

2.2 Basics of SLAM

The Simultaneous Localization and Mapping (SLAM) problem asks if it is possible for a mobile robot to be placed at an unknown location in an unknown environment and for the robot to incrementally build a consistent map of this environment while simultaneously determining its location within this map. A solution to the SLAM problem provides the means to make a robot truly autonomous. The solution of the SLAM problem has been one of the notable successes of the robotics community. SLAM has been formulated and solved as a theoretical problem in a number of different forms. SLAM has also been implemented in a number of different domains from indoor robots, to outdoor, underwater, and airborne systems [13].

2.2.1 Brief History of SLAM

The beginning of the probabilistic SLAM problem happened at the 1986 IEEE Robotics and Automation Conference. At this time probabilistic methods were just beginning to be implemented in robotics and AI systems. Early researchers into applying estimation methods to the mapping and localization problems included Peter Cheeseman, Jim Crowley, and Hugh Durrant-Whyte. Early work by Smith and Cheeseman [18] and Durrant-Whyte [11] established a statistical basis for describing relationships between landmarks and manipulation geometric uncertainty. Smith, Self, and Cheeseman [19] published a landmark paper in 1990 that showed that as a mobile robot moves through an unknown environment taking relative observations of landmarks, the estimates of these landmarks are all necessarily correlated with each other because of the common error in estimated vehicle location. The acronym 'SLAM' was first presented at the 1995 International Symposium on Robotics Research [14].
2.2.2 Definition of the SLAM Problem

Consider a mobile robot moving through an environment taking relative observations of a number of unknown landmarks using a sensor located on the robot as shown in Figure 2.4. At a time instant $k$, the following quantities are defined:

- $\mathbf{x}_k$: the state vector describing the location and orientation of the vehicle.
- $\mathbf{u}_k$: the control vector, applied at time $k - 1$ to drive the vehicle to a state $\mathbf{x}_k$ at time $k$.
- $\mathbf{m}_i$: a vector describing the location of the $i$th landmark whose true location is assumed time invariant.
- $\mathbf{z}_{ik}$: an observation taken from the vehicle of the location of the $i$th landmark at time $k$. When there are multiple landmark observations at any one time or when the specific landmark is not relevant to the discussion, the observation will be written simply as $\mathbf{z}_k$. 

Figure 2.4. The essential SLAM problem[13]
In addition, the following sets are also defined:

- \( X_{0:k} = \{x_0, x_1, \ldots, x_k\} \): the history of vehicle locations.
- \( U_{0:k} = \{u_1, u_2, \ldots, u_k\} \): the history of control inputs.
- \( m = \{m_1, m_2, \ldots, m_n\} \): the set of all landmarks.
- \( Z_{0:k} = \{z_1, z_2, \ldots, z_k\} \): the set of all landmark observations.

### 2.2.3 Probabilistic SLAM

In probabilistic form, the Simultaneous Localization and Mapping problem requires that the probability distribution

\[
P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0)
\]

be computed for all times \( k \). This probability distribution describes the joint posterior density of the landmark locations and vehicle state (at time \( k \)) given the recorded observations and control inputs up to and including time \( k \) together with the initial state of the vehicle. Starting with an estimate for the distribution \( P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1}) \) at time \( k - 1 \), the joint posterior, following a control \( u_k \) and observation \( z_k \), is computed using Bayes Theorem. This computation requires that a motion model and observation model be defined.

The **observation model** describes the probability of making an observation \( z_k \) when the vehicle location and landmark locations are known, and is defined as

\[
P(z_k \mid x_k, m)
\]

The **motion model** for the vehicle can be described in terms of a probability distri-
bution on state transitions as follows

\[ P(x_k \mid x_{k-1}, u_k) \]  

(2.3)

The SLAM algorithm is now implemented in a standard two-step recursive (sequential) prediction (time-update) correction (measurement-update) form:

**Time-update**

\[ P(x_k, m \mid Z_{0:k-1}, U_{0:k}, x_0) = \int P(x_k \mid x_{k-1}, u_k) \times P(x_{k-1} \mid Z_{0:k-1}, U_{0:k-1}, x_0) \, dx_{k-1} \]  

(2.4)

**Measurement Update**

\[ P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k \mid x_k, m)P(x_k, m \mid Z_{0:k-1}, U_{0:k-1}, x_0)}{P(z_k \mid Z_{0:k-1}, U_{0:k})} \]  

(2.5)

Equations 2.4 and 2.5 provide a recursive procedure for calculation the joint posterior \( P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) \) for the robot state \( x_k \) and map \( m \) at a time \( k \) based on all observations \( Z_{0:k} \) and all control inputs \( U_{0:k} \) up to and including time \( k \).

### 2.2.4 Most Popular Solution to the SLAM Problem

Solutions to the probabilistic SLAM problem involve finding an appropriate representation for the observation model Equation 2.2 and motion model Equation 2.3 which allows efficient and consistent computation of the prior and posterior distributions in Equations 2.4 and 2.5. By far the most common representation is in the form of a state-space model with additive Gaussian noise, leading to the use of the extended Kalman filter (EKF) to solve the SLAM problem. While EKF-SLAM is the most popular there are alternatives including FastSLAM which uses the Rao-Blackwellised particle filter.
The basis for the EKF-SLAM method is to describe the vehicle motion in the form

\[ P(x_k \mid x_{k-1}, u_k) \iff x_k = f(x_{k-1}, u_k) + w_k \]  

(2.6)

where \( f(\ldots) \) models vehicle kinematics and where \( w_k \) are additive, zero mean uncorrelated Gaussian motion disturbances with covariance \( Q_k \). The observation model is described in the form

\[ P(z_k \mid x_k, m) \iff z(k) = h(x_k, m) + v_k, \]  

(2.7)

where \( h(\ldots) \) describes the geometry of the observation and where \( v_k \) are additive, zero mean uncorrelated Gaussian observation errors with covariance \( R_k \).

With these definitions the standard EKF method[17], [12] can be applied to compute the mean

\[
\begin{bmatrix}
\hat{x}_{k|k} \\
\hat{m}_{k}
\end{bmatrix} = E \begin{bmatrix} x_k & Z_{0:k} \end{bmatrix},
\]  

(2.8)

and covariance

\[
P_{k|k} = \begin{bmatrix} P_{xx} & P_{xm} \\ P_{xm}^T & P_{mm} \end{bmatrix} = E \begin{bmatrix} (x_k - \hat{x}_k) & (x_k - \hat{x}_k)^T \\ (m - \hat{m}_k) & (m - \hat{m}_k)^T \end{bmatrix}.
\]  

(2.9)

of the joint posterior distribution \( P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) \) from:

**Time-update**

\[
\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)
\]  

(2.10)

\[
P_{xx,k|k-1} = \nabla f P_{xx,k-1|k-1} \nabla f^T + Q_k
\]  

(2.11)

where \( \nabla f \) is the Jacobian of \( f \) evaluated at the estimate \( \hat{x}_{k-1|k-1} \). There is generally
no need to perform a time-update for stationary landmarks.

**Observation-update**

\[
\begin{bmatrix}
\hat{x}_{k|k} \\
\hat{m}_{k}
\end{bmatrix}
= \begin{bmatrix}
\hat{x}_{k|k-1} \\
\hat{m}_{k-1}
\end{bmatrix} + W_k[z(k) - h(\hat{x}_{k|k-1}, \hat{m}_{k-1})]
\]  

(2.12)

\[
P_{k|k} = P_{k|k-1} - W_k S_k W_k^T
\]  

(2.13)

where

\[
S_k = \nabla h P_{k|k-1} \nabla h^T + R_k
\]  

(2.14)

\[
W_k = P_{k|k-1} \nabla h^T S_k^{-1}
\]  

(2.15)

and where \(\nabla h\) is the Jacobian of \(h\) evaluated at \(\hat{x}_{k|k-1}\) and \(\hat{m}_{k-1}\). This EKF-SLAM solution is very well known and inherits many of the same benefits and problems as the standard EKF solutions to navigation or tracking problems.
CHAPTER 3

HARDWARE AND ASSEMBLY OF BASIC DRONE

Instead of buying and using a high end pre-built quadcopter, we opted to build a quadcopter from the ground up. Although it would have been much simpler and faster to buy a pre-built drone, by building our own we had more control over which parts the drone was made up of as well as the functionality of the finished drone itself. In this portion of the document I will explain which parts the drone is composed of, and how everything is assembled.

3.1 The Frame

First of all we started with a frame kit from the popular drone company DJI. The kit we settled on was the DJI Flamewheel F450 ARF Kit. As seen in Figure 3.1. The DJI Flamewheel F450 ARF Kit includes a quadcopter frame, ESCs (Electronic Speed Controllers), propellers, and motors. After reading the user manual [7], the kit was straightforward to assemble. The frame itself shipped in 6 pieces and was assembled using the supplied hardware. Next it was just a matter of installing the motors, ESCs and soldering the connections to the power distribution board. The power distribution board was integrated into the bottom portion of
the frame making it somewhat difficult to get the soldering iron in to make a good solder joint. A few personal opinions about the DJI kit, the frame itself is made very well out of good quality material, even after a hard crash the frame never had any problems. The diagonal “wheelbase” of the frame is 450mm which is a little large for indoor use, and the frame’s weight is 282g by itself which could easily break things if crashed into. Well after we purchased the kit we found out that the DJI ESCs are pre-calibrated from the factory and do not have the ability to be re-calibrated, which caused some problems for us that I will address later in the document.

3.2 The Flight Controller

The next part for the quadcopter that was needed was a flight controller. A flight controller is a small circuit board of varying complexity. Its function is to direct the RPM of each motor in response to input. A command from the pilot for the multi-rotor to move forward is fed into the flight controller, which determines how to manipulate the motors accordingly. Flight controllers also employ sensors to supplement their calculations. These range from simple gyroscopes for orientation to barometers for automatically holding altitudes. The
flight controller we settled on was the 3DR Pixhawk Mini, which can be seen in Figure 3.2. The Pixhawk Mini comes with a built in gyroscope, accelerometer, barometer, as well as an external GPS/Compass unit. Along with all the sensors, the flight controller came with a quad power distribution board as well as all the wiring and connectors to connect the Pixhawk to the quadcopter. I will describe in more detail later how all the wiring for the quadcopter is assembled.

3.3 Transmitter and Receiver

Now that we had the basic frame and the flight controller for the quadcopter, we needed a way for an operator to control the drone for basic flying purposes. For this we purchased the FrSky Taranis X9D Radio Transmitter and the FrSky Delta 8 2.4Ghz 8CH Multi-Brand Receiver as shown in Figure 3.3. The FrSky Taranis is a very good all purpose RC Transmitter running the open source ‘Open TX’ software and comes with many configurable switches and dials. The FrSky Delta 8 is a good all purpose RC Receiver that is PPM-Sum compatible. PPM-Sum is needed by the Pixhawk and just means that all of the channels from the receiver can be sent to the Pixhawk along a single connection.
3.4 Batteries

The last part that is needed for a basic functioning quadcopter that can be controlled by an operator using the RC transmitter is a power source. For this we purchased 3 Lumenier 1800mAh 4s 75c Lipo batteries. This size battery is what the DJI kit called for and offers around 15 minutes of flight time before needing to be recharged.

3.5 Assembly

Refer to Figure 3.4 for a visual representation of how a typical quadcopter is wired. Although the diagram does not show exactly how our setup is wired, it is a good visual of how a typical quadcopter should be wired. From the diagram you can see that each motor should be connected to its own ESC. The ESCs in turn are supplied power directly from the Lipo battery, with a third wire connected to the flight controller. This third wire that connects the flight controller to the ESCs is the signal wire. The flight controller, depending on input from the operator or sensor input, will send a PWM (Pulse Width Modulation) signal to the ESC telling it what RPM to turn the motor. The motors themselves must be connected to the flight controller in a certain way, if you look back at Figure 2.2 the numbers on each propeller corresponds to the output number on the flight controller. This makes sense
because the flight controller must know the position of each motor it is controlling as well as the direction that the motor is spinning.

Not shown in the diagram is how the flight controller is powered. In our setup the Pixhawk came with its own power distribution board. This board is connected to the battery by soldering power and ground wires to the power distribution board integrated into the frame, it then supplies the Pixhawk with its required 5V. Finally the GPS and RC receiver are connected to the flight controller by simply plugging the connectors into their corresponding sockets which are labeled on the flight controller.

3.6 Additional Hardware

In order to give the quadcopter the ability to fly autonomously indoors we needed to add some additional hardware.
3.6.1 Companion Computer

For starters we needed a “brain” to be able to instruct the quadcopter to perform actions within certain circumstances. While the flight controller does have the ability to do this to some extent, in more extreme circumstances like when reading additional sensor data and constructing a map of the environment the flight controller doesn’t have the processing power needed. To fix this we added a companion computer. The companion computer we settled on was a Raspberry Pi 3. The Raspberry Pi is connected to the flight controller directly and communicates with it via UART serial communication. UART stands for Universal Asynchronous Receiver/Transmitter. A UART serial connection only requires a 4 wire connection and doesn’t require a shared clock. The Raspberry Pi has a UART connection built into its normal GPIO (General Purpose Input Output) and connects to the flight controller on the flight controller’s telemetry port. While the flight controller’s telemetry port is not meant to be used in this fashion, peripheral suppliers have been able to use the UART connection of the telemetry port to create interfaces to communicate with the flight controller directly. The wiring between the Raspberry Pi and the Pixhawk is shown in Figure 3.5. The original Pixhawk had 2 telemetry ports and it was standard to connect a companion computer via Figure 3.5. Wiring between RPI and Pixhawk[6]
port Telem2. However the Pixhawk Mini only has one telemetry port, so the UART connection has been made using the Pixhawk Mini’s only telemetry port. It is stated in the documentation [6] that the 5V from the Pixhawk’s telemetry port is sufficient to power the Raspberry Pi. However in our experimentation we found that the Raspberry Pi would not boot because of low voltage when only receiving power from the Pixhawk’s telemetry port. Instead of powering the Raspberry Pi via its GPIO, we were able to pull 5V from the flight controller’s power distribution board. Since that was possible we soldered a micro USB cable to the power distribution board and can power the Raspberry Pi via micro USB, which is how the Raspberry Pi is normally powered since that is where all the protective circuitry is located.

3.6.2 Ultra Sonic Range Sensor

While the Pixhawk flight controller does have a built-in barometer, which is an instrument that measures atmospheric pressure in order to determine altitude, we had problems with getting an accurate reading for the altitude when performing experimental tests, a problem I will discuss in more detail later. For this reason we added an HC-SR04 ultrasonic range finder sensor that sits on the bottom of the quadcopter pointing straight down to give us an accurate measurement for the distance to the ground. The ultrasonic range finder sensor plugs directly into the Raspberry Pi’s GPIO ports, however the signal it outputs needs to be converted from 5V to 3.3V so as not to damage the Raspberry Pi.

Sound consists of oscillating waves through a medium (such as air) with the pitch being determined by the closeness of those waves to each other, defined as the frequency. Only some of the sound spectrum is audible to the human ear, defined as the “Acoustic” range. Very low frequency sound below Acoustic is defined as “Infrasound”, with high frequency sounds above, called “Ultrasound.” Ultrasonic sensors are designed to sense object proximity or range using ultrasound reflection, similar to radar, to calculate the time it takes to reflect ultrasound waves between the sensor and a solid object. Ultrasound is mainly used because
it’s inaudible to the human ear and is relatively accurate within short distances.

A basic ultrasonic sensor consists of one or more ultrasonic transmitters (basically speakers), a receiver, and a control circuit. The transmitters emit a high frequency ultrasonic sound, which bounce off any nearby solid objects. Some of that ultrasonic noise is reflected and detected by the receiver on the sensor. That return signal is then processed by the control circuit to calculate the time difference between the signal being transmitted and received. This time can subsequently be used, along with some clever math, to calculate the distance between the sensor and the reflecting object.

The HC-SR04 ultrasonic sensor has four pins: ground (GND), Echo Pulse Output (ECHO), Trigger Pulse Input (TRIG), and 5V Supply (Vcc). As can be seen in Figure 3.6. We power the module using Vcc, ground it using GND, and use our Raspberry Pi to send an input signal to TRIG, which triggers the sensor to send an ultrasonic pulse. The pulse waves bounce off any nearby objects and some are reflected back to the sensor. The sensor detects these return waves and measures the time between the trigger and returned pulse,
ECHO will be “low” (0V) until the sensor is triggered when it receives the echo pulse. Once a return pulse has been located ECHO is set “high” (5V) for the duration of that pulse. Pulse duration is the full time between the sensor outputting an ultrasonic pulse, and the return pulse being detected by the sensor receiver. We used a Python script to measure the pulse duration and then calculate distance from this. The sensor output signal (ECHO) on the HC-SR04 is rated at 5V. However, the input pin on the Raspberry Pi GPIO is rated at 3.3V. Sending a 5V signal into that unprotected 3.3V input port could damage our GPIO pins, which is something we want to avoid. For this reason we used a small voltage divider circuit, consisting of two resistors, to lower the sensor output voltage to something the Raspberry Pi can handle.

A voltage divider consists of two resistors (R1 and R2) in series connected to an input voltage (Vin), which needs to be reduced to our output voltage (Vout). In our circuit, Vin will be ECHO, which needs to be decreased from 5V to our Vout of 3.3V. A basic voltage divider circuit is shown in Figure 3.7. The circuit in Figure 3.7 and the simple equation in Figure 3.8 can be applied to many applications where a voltage needs to be reduced. We know our input voltage (5V), and our required output voltage (3.3V), and we could have used any combination of resistors to achieve the reduction. I happened to have a bunch of extra 1kΩ resistors, so we decided to use one of these in the circuit as R1. After some simple
math we got $R_1$ as 1kΩ and a 2kΩ resistor for $R_2$.

In order to connect the ultrasonic sensor to the Raspberry Pi we used four pins on the Raspberry Pi GPIO. GPIO 5V [Pin 2]: Vcc (5v Power), GPIO GND [Pin 6]: GND (0V Ground), GPIO 23 [Pin 16]: TRIG (GPIO Output) and GPIO 24 [Pin 18]: ECHO (GPIO Input). The connects to the Raspberry Pi including the voltage divider can be seen in Figure 3.9.

As stated earlier we created a Python module on the Raspberry Pi that calculates the measured distance to the ground. This module can be imported into any other Python
script allowing it to trigger a measurement and retrieve the measured distance.

3.6.3 Laser Range Scanner

The last piece of hardware that was incorporated into the quadcopter was a laser range scanner. The laser range scanner is needed for SLAM. In order to construct a map of the environment and determine the quadcopter’s location within the environment, SLAM needs a way to sense the environment. While this can be done with a number of different sensors, from ultrasonic range finders, fixed position laser range finders, and even cameras in the case of visual SLAM, the most commonly used sensor to implement SLAM is a laser range scanner or LiDAR.

The principle behind LiDAR is really quite simple. Shine a small light at a surface and measure the time it takes to return to its source. Light travels very fast, about 300,000 kilometres per second or 0.3 metres per nanosecond, so equipment required to measure this needs to operate extremely fast. Only with the advancements in modern computing technology has this become possible. The LiDAR instrument fires rapid pulses of laser light at a surface, some at up to 150,000 pulses per second. A sensor on the instrument measures the amount of time it takes for each pulse to bounce back. Light moves at a constant and known speed so the LiDAR instrument can calculate distance between itself and the target with high accuracy. By repeating this in quick succession the instrument builds up a complex map of the surface it is measuring. As the sensor is moving location and orientation of the instrument must be included to determine the position of the laser pulse at the time of sending and the time of return. The laser range scanner we chose to use was the RPLidar A2 pictured in Figure 3.10.

The core of RPLidar A2 runs clockwise to perform a 360 degree omnidirectional laser range scanning for its surrounding environment and then generate an outline map for the environment. The sample rate of LIDAR directly decides whether the robot can map quickly and accurately. With a rotational frequency of 10 Hertz RPLidar A2 is able to provide
4000 samples per second and has a scan range accurate up to 6 meters. The RPLidar A2 laser range finder connects to the Raspberry Pi via a USB connection and SLAMTEC, the manufacturers of the scanner, provide a downloadable SDK which makes getting the scanner running quick and easy. The SDK as well as a user manual can be downloaded from [20].
CHAPTER 4

SOFTWARE STACK

4.1 Flight Controller Firmware

Within the flight controller market there are two main firmwares that are used within the
controller, APM and PX4. Ardupilot Mega (APM) is a professional quality IMU autopilot
that is based on the Arduino Mega platform. This autopilot can control fixed-wing aircraft,
multi-rotor helicopters, as well as traditional helicopters. It is a full autopilot capable for
autonomous stabilization, way-point based navigation and two way telemetry. Although
APM was originally built on the Arduino Mega platform it has moved to an open source
firmware stack that can be installed and used on a large number of different flight controllers,
including Pixhawk. PX4 is much like APM, except PX4 is part of the Dronecode Project, a
shared and collaborative open source project to deliver a complete end-to-end platform for
unmanned aerial vehicles.

While Pixhawk flight controllers are developed to be used with PX4 we decided to
use APM because DroneKit-Python can be interfaced with APM (more on DroneKit-Python
later). This is where an issue arose. Although APM works well with the original Pixhawk
hardware, the Pixhawk Mini is a fairly new hardware platform and the current stable version
of APM, version 3.4, is not compatible with the Pixhawk Mini. The work around we used
to resolve this issue was to use the “master” version of APM, version 3.6, which is still in beta as of this time. After installing APM version 3.6 onto the Pixhawk Mini and running some basic test it seems to work well enough for our experimentation, although some of the functionality doesn’t work very well such as monitoring the battery level and landing the craft when the battery becomes too low.

4.1.1 Installing and Configuring APM Firmware

In order to install and configure the APM firmware it requires a Ground Control Station (GCS), that is to be installed onto your desktop or laptop. There are two GCS programs that are officially supported by APM. If your machine runs Windows it is recommended to use Mission Planner, while if your machine is running a Linux distro or MacOS it is recommended to use APM Planner 2. Mission Planner was the original GCS software to install and configure APM but was only available on Windows. APM Planner 2 is a later version of Mission Planner that is cross-platform. After experimenting with both we’ve found there is little difference between the two applications. From the GCS software you can install different APM firmwares. It even offers a wizard after a new firmware installation that will walk you through the configuration process.

After a fresh firmware installation there are generally several things that are mandatory to configure before the drone is ready for basic flight. These mandatory configurations include: frame type, compass calibration, accelerometer calibration, radio calibration, flight mode setup, and failsafe setup. Frame type, compass calibration, and accelerometer calibration are all simple to configure and the GCS software walks you through the whole process. However the radio calibration is a little more difficult. During this step you are setting up the RC transmitter and letting the flight controller know which channels will be used for which commands as well as the thresholds for each of those channels. For our setup we can control which flight mode we are currently in using a 3 position switch on the front right of the controller, corresponding to channel 5. The three flight modes that are currently setup
are center position (STABILIZE), down position (ALT_HOLD), and up position (LAND). STABILIZE flight mode is like a manual flight mode. In this mode you have the most control over the copter, the flight controller just tries to keep the copter as stable as possible. From my experience this is the hardest mode to control the copter and have had some pretty hard landings while flying in STABILIZE flight mode. The mode I personally like flying the copter in is ALT_HOLD mode. ALT_HOLD attempts to hold the copter at the current altitude but leaves you in control of everything else. While ALT_HOLD attempts to hold the current altitude there is still some “drift” in altitude, especially when there is wind, when this occurs you can still tell the flight controller to increase or decrease the altitude using the throttle joystick, it will just increase or decrease very slowly. The last flight mode setup to be used is the LAND flight mode. When LAND flight mode is invoked, the throttle joystick is ignored and the the copter levels itself out and slowly descends until it touches the ground. This was setup as a manual failsafe, if the operator loses control of the copter then you can switch it to LAND and not worry about damaging the copter. The last mandatory configuration to be setup is the failsafe configuration. The failsafe configuration tells the flight controller what to do when certain circumstances occur, such as when the battery voltage becomes too low, or if the flight controller loses the signal from the RC transmitter. In our setup all failsafes have been set to land at current position. Once the APM firmware has been installed and configured you can take the quadcopter out and fly it using the RC transmitter. Flying the quadcopter is a lot of fun and I personally have spent a lot of time just flying our setup around the yard.

4.2 Raspberry Pi’s Operating System

While there are many different operating systems that can be ran on a Raspberry Pi including, Raspbian, Debian, Ubuntu MATE, Arch Linux, and Gentoo Linux to name a few, the most popular seems to be Raspbian. The operating system currently running on our Rasp-
The Raspberry Pi 3 is Raspbian Jessie. Raspbian Jessie is the current stable version of the popular Raspbian operating system. The Raspbian operating system is based on Debian Linux, and all of the software we need to run on the Raspberry Pi works well with Raspbian.

### 4.3 DroneKit-Python

In order to control the quadcopter autonomously we needed a way to create applications on a companion computer, in this case the Raspberry Pi, and then send commands to the flight controller to control the quadcopters movements. For this we chose to use DroneKit-Python. DroneKit-Python allows developers to create apps that run on an onboard companion computer and communicate with the ArduPilot (APM) flight controller using a low-latency link. The API communicates with vehicles over MAVLink, or Micro Air Vehicle Link. It provides programmatic access to a connected vehicle’s telemetry, state and parameter information, and enables both mission management and direct control over vehicle movement and operations. DroneKit-Python’s API provides classes and methods to:

- Connect to a vehicle (or multiple vehicles) from a script.
- Get and set vehicle state/telemetry and parameter information.
- Receive asynchronous notification of state changes.
- Guide a UAV to specified postion (GUIDED mode).
- Send arbitrary custom messages to control UAV movement and other hardware (GUIDED mode).
- Create and manage waypoint missions (AUTO mode).
- Override RC channel settings.
Anyone with programming experience in Python has the ability to pick up DroneKit-Python very quickly. The API is also very well documented, with quick start guides and examples for all of the built-in functionality. The documentation can be found at [10].

4.4 Software to Run SLAM

While we could implement our own version of SLAM, there is no need in reinventing the wheel as they say. Remember SLAM has been around since the late 80s, so there are many open source SLAM algorithms out there as well as many implementations. Most SLAM approaches require odometry input in order to complete the problem. With wheeled robots this is not a problem, since you can acquire odometry from how much the wheels have turned, but with a quadcopter it is very difficult to get odometry information. With this in mind we chose to implement Hector SLAM. Hector SLAM can build up a map and predict location without the need for odometry. Since the lidar sensor outputs a distance versus polar angle readings, when the sensor moves, the readings change. Hector SLAM uses this change in sensor readings in place of odometry input.

In order to implement Hector SLAM in our system, we decided to use ROS. The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. One of the packages that ROS provides is an implementation of Hector SLAM. This implementation of Hector SLAM in ROS not only provides the core functionality of constructing the map and predicting the robot’s location within it but also provides a means of saving the map to be reloaded and updated later. ROS is known to be very complex and to have a steep learning curve, but luckily there is a lot of documentation on how to install the system and get it set up and working[9]. Once the ROS system as well as the Hector SLAM package was installed onto our system we needed a way for our LiDAR sensor to talk to ROS. Luckily a group
known as “robopeak” has already written a ROS package that allows our RPLidar sensor to work with the Hector SLAM package. After modifying some of the configurations we were able to use the ROS system to build a map and predict the quadcopter’s localization within the map. The results of the SLAM implementation is presented in Chapter 6.
As stated earlier, the software we chose to use in order to programatically control the movements of the quadcopter is DroneKit-Python. When first researching this API it seemed to be exactly what we needed in order to perform our autonomous controls, since the API has built-in methods for commands like move forward one meter or rotate orientation clockwise by 30 degrees. However, once we started real world test, we realised these functions are only available when the quadcopter was in GUIDED flight mode. Under normal conditions this would not be a problem as the APM firmware supports GUIDED flight mode, the problem arose because we were testing and developing the system to work indoors. Within the APM firmware, GUIDED flight mode has a failsafe that will not let the quadcopter arm its motors unless it has a strong GPS signal, and since we were testing indoors the GPS signal was weak or not present at all. After reading through the documentation we found there is no way of turning this failsafe off and because of this we were unable to use most of the built-in functionality DroneKit-Python has to offer. In order to control the quadcopter’s movements we had to resort to using RC channel overrides. RC channel overrides, which is supported by DroneKit-Python, does exactly what the name implies, it overrides the RC channel inputs from the RC transmitter. While DroneKit-Python does offer this functionality it is not recommended, they even give the following warning about using RC
channel overrides, “WARNING: Channel overrides (a.k.a. RC overrides) are highly discour-aged (they are primarily intended for simulating user input and when implementing certain types of joystick control). Instead use the appropriate methods to set the desired position or direction/speed.[10]” Even though using RC overrides was discouraged this was the only option we could use to control the movements of the quadcopter indoors.

5.1 Taking off to Target Altitude

In order to get the quadcopter to ascend a target altitude we wrote a function in DroneKit-Python that accepts the target altitude in centimeters as an argument, it then slowly increases the output of RC channel 3, which is the throttle channel, while continuously monitoring the quadcopter’s altitude, once the target altitude is reached the function returns. This function is presented in Algorithm 1.

<table>
<thead>
<tr>
<th>Algorithm 1 Take Off to Target Altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A Target Altitude ( \text{target} )</td>
</tr>
<tr>
<td><strong>Output:</strong> NA</td>
</tr>
<tr>
<td><strong>while</strong> 1 do</td>
</tr>
<tr>
<td>( \text{channel}[3].\text{override} \leftarrow \text{channel}[3] + 3 )</td>
</tr>
<tr>
<td>( \text{if } \text{Altitude} \geq \text{target} \times 0.95 ) then <strong>return</strong></td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td>time.sleep(1)</td>
</tr>
<tr>
<td><strong>end while</strong></td>
</tr>
</tbody>
</table>

While this function seems quite simple, which it is, using RC overrides can produce some weird results from time to time. This is because you can’t be sure what the RC channel will be when the method is called and there is no way that I have found to preset a RC channel to a desired value, the override is dependent on the current channel value. While this approach works it is slow as the propellers slowly increase to be able to generate enough thrust to lift the quadcopter.
5.2 Holding the Quadcopter at Current Altitude

Once the quadcopter has reached a desired altitude we want it to hold this altitude while all other functionality is being performed, until we invoke a landing function. The most straightforward approach to achieve would be to switch the APM firmware into ALT_HOLD mode and let the flight controller do the heavy lifting for you. However because of RC overrides this is not possible because you cannot be sure as to what the throttle channel value is, also even if the value is within a desired range, as stated earlier ALT_HOLD mode will still allow the quadcopter to drift in altitude. This drift is usually minimum but indoors it could cause problems. Because of this we wrote another function in DroneKit-Python that continuously monitors the quadcopter’s altitude and applies slight adjustments to the throttle accordingly. This function is invoked after reaching a target altitude and is executed in a separate thread so that it is always attempting to hold the target altitude. This function is presented in Algorithm 2.

\begin{algorithm}
\caption{Holding Target Altitude}
\begin{algorithmic}
\State \textbf{Input:} A Target Altitude \texttt{target}
\State \textbf{Output:} NA
\While{1}
\If{\texttt{Altitude} > \texttt{target} * 1.1}
\State \texttt{channel[3].override} \Leftarrow \texttt{channel[3]} - 1
\EndIf
\If{\texttt{Altitude} < \texttt{target} * 0.9}
\State \texttt{channel[3].override} \Leftarrow \texttt{channel[3]} + 1
\EndIf
\State \texttt{time.sleep(0.5)}
\EndWhile
\end{algorithmic}
\end{algorithm}

As stated earlier this function is continuously running in its own thread, once it is time to land the craft, the thread is killed and the land gently function is invoked.
5.3 Moving Quadcopter Forwards, Backwards, Left, Right, and Orientation

In order to move the quadcopter forwards, backwards, left, right, and orientation, we wrote another function in DroneKit-Python using RC overrides except in this case we are overriding the corresponding channel, channel 2: pitch, channel 1: roll, channel 4: yaw. This function takes 3 parameters, the first parameter is a 1 or -1 corresponding to direction. The second parameter is the duration the movement should occur. The third parameter is the channel to override. The function is presented in Algorithm 3.

Algorithm 3 Moving Pitch, Roll, and Yaw

<table>
<thead>
<tr>
<th>Input:</th>
<th>Direction(1 or -1) Direction, Duration in seconds Duration, Channel to Override Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>NA</td>
</tr>
</tbody>
</table>

```python
channel[Channel].override ← channel[Channel] + (Direction * 5)

for i to Duration do
    time.sleep(1)
end for

channel[Channel].override ← channel[Channel] - (Direction * 5)
```

As can be seen from Algorithm 3 the function overrides the corresponding channel by 5 units in the direction that was passed as an argument, this off-setting by 5 was chosen from trail-and-error which moves the quadcopter at a slow steady pace, it then waits for the duration of time that was passed as an argument, then finally resets the corresponding channel back to its starting value. For clarification direction 1 on channel 2 corresponds to the forward pitch direction, while direction -1 on channel 2 corresponds to the backwards pitch direction, direction 1 on channel 1 corresponds to the right roll direction, while -1 on channel 1 corresponds to the left roll direction, direction 1 on channel 4 corresponds to clockwise rotation of yaw, and direction -1 on channel 4 corresponds to counterclockwise rotation of yaw.
5.4 Landing the Quadcopter

In order to land the quadcopter, we first would just switch the flight controller into LAND mode using the corresponding DroneKit-Python command and let the flight controller land the quadcopter for us. However for reasons I will discuss in the next chapter, this would result in the motors of the quadcopter to stop while it was still a good ways off of the ground and the quadcopter would free fall to the ground. For this reason we wrote our own land function in DroneKit-Python that monitors the altitude of the quadcopter and slowly decreases the throttle till the quadcopter was safely on the ground. The function is presented in Algorithm 4.

**Algorithm 4** Landing the Quadcopter

<table>
<thead>
<tr>
<th>Input:</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>NA</td>
</tr>
</tbody>
</table>

while Age > 13cm do
  time.sleep(1)
end while

VehicleMode ← "LAND"
CHAPTER 6

EXPERIMENTAL RESULTS

When it came time to start real world experimentation within an indoor environment, it quickly became clear that we had overlooked an important part of the project, a place to test the quadcopter. Since the quadcopter we built is quite large and with all of the additional hardware required for this project it is also quite heavy, having a safe place in order to test it was essential. The environment had to be void of other people as well as pets, believe me from first hand experience getting hit by the moving propellers doesn’t feel very good and will break the skin if hit hard enough. The environment also needed to be relatively empty, it would be a shame for the quadcopter to shatter an expensive big screen tv. For much of the testing of basic functionality I was attempting to test the quadcopter in my living room, but in doing so I was always very nervous and would kill the quadcopter at the first sign that it was not working correctly. Because of this reason early testing was very slow to say the least. Finally I was able to find a suitable environment to test. A buddy had recently purchased a new home that had a large two car garage that was pretty much empty when they were not home. Luckily he agreed I could use this space as my testing environment.

One of the early issues we ran into was actually getting the DroneKit-Python API to communicate with the APM firmware on the flight controller. As I had done the soldering between the Raspberry Pi and the flight controller myself, my first instinct was that it was
a hardware problem. However after checking and double checking the connections between the Raspberry Pi and the flight controller we decided it was not a hardware problem but in fact a software problem. With my confidence in my soldering skill back, I started to read the DroneKit-Python documentation. The issue was finally found with how the DroneKit-Python API was connecting to the Raspberry Pi’s serial port. The DroneKit-Python API invokes a $\text{connect()}$ function that takes the connection string as well as the baud rate of the serial port you are attempting to connect to. In the API documentation the connection string to use in order to connect to a Raspberry Pi’s serial port, or UART port, was “/dev/ttyAMA0” with a baud rate of 57600. However, since the documentation was released Raspberry Pi’s Raspbian operating system had renamed its device ports, we finally discovered that the UART serial port on the Raspberry Pi had been renamed to “dev/ttyS0.” After changing this connection string in the connect function, we were able to communicate between the DroneKit-Python API on the Raspberry Pi and the APM firmware running on the Pixhawk flight controller.

The next issue we ran into while testing the quadcopter was its ability to accurately read its altitude indoors. As stated earlier the Pixhawk flight controller has an internal barometer that uses the atmospheric pressure in order to determine its altitude. Within DroneKit-Python we have access to the barometer’s readings, and we were trying to use this altitude reading in order to take off to a target altitude as well as hold a target altitude. However when testing the results were very inconsistent. There were times when the quadcopter had not even lifted off of the floor and the program would say it had reached its target altitude, other times the quadcopter would continue climbing in altitude until it slammed into the ceiling, and every once in a while it would work exactly how we had programmed it to. The problem was finally found to be how the barometer determines altitude. Since we were testing in an enclosed space that was climate controlled, the air pressure of the environment doesn’t change much from one altitude to another, and with that being the case we would never be able to get accurate measures for altitude using the embedded barometer in
the flight controller. This led to using the ultrasonic range sensor placed on the bottom of the quadcopter that would tell us the exact distance to the ground. The use of the ultrasonic range sensor on the quadcopter works very well. It is accurate to within 1-2cm, and since we wrote the range sensor code in a python module, we can import its functions directly into our DroneKit-Python scripts. Anytime we needed to check the quadcopters current altitude, we could just trigger the ultrasonic sensor and retrieve the distance to the ground. While this was a nice fix to our problem, it isn’t without its faults. For starters the ultrasonic sensor doesn’t work on carpet, at least not the carpet in my home which is pretty thick. Since the range sensor uses soundwaves it is my guess that the carpet is absorbing the soundwaves not allowing them to return to the sensor. The other problem with using the range sensor in place of the barometer is when the quadcopter flies over something the program thinks it has dropped in altitude when it reality it is at the same height. While this is an annoyance, this issue could be fixed programmatically by ignoring any sudden drops or rises in altitude.

While the previous problems were able to be resolved, this next problem could not be and it makes any testing of the quadcopters autonomous controls very difficult, not impossible but difficult. It has to do with the DJI ESCs being pre-calibrated from the factory. The DJI ESCs we purchased for our system are meant to be used by a DJI flight controller. However since we are using a Pixhawk flight controller running APM firmware the ESCs aren’t calibrated for it. From my own research into the problem it seems that DJI ESCs are the only ESCs that cannot be re-calibrated once leaving the factory and this causes us a problem because one of the ESCs spins its motor noticeably slower than the other 3. To be more specific it is the back right ESC that needs to be re-calibrated. What is strange is the flight controller seems to resolve the issue after the quadcopter is up in the air and has been flying around for a bit. However the flight controller does nothing to resolve the issue when trying to take off. When attempting to take off the quadcopter moves fairly quickly backwards and to the right until the flight controller resolves the issue. There has even been a couple of occasions when attempting to take off the quadcopter will flip over
onto its back. The only workaround we were able to use in order to continue testing was for an operator to use the RC transmitter to try to keep it steady while the DroneKit-Python script attempts to take off. While this workaround works most of the time, it is still very difficult to keep the quadcopter in one place without hitting anything on its take off, even in an empty garage.

Even with these setbacks we were still able to get the quadcopter to perform some very basic autonomous commands such as taking off to a target altitude and holding the target altitude with a little help from an operator on the RC transmitter, moving forward, backward, left, right, and changing the orientation. I had hoped to get more autonomous functionality out of our system but with all of the setbacks quite frankly I ran out of time.

Results from the Hector SLAM running in ROS was a different story. We had good results with getting a map built as well as determining the location within the map using the RPLidar A2 laser scanner. Figure 6.1 shows a map that was constructed of my home with me walking the drone around simulating the drone exploring the environment.

As can be seen from Figure 6.1, the walls of the rooms and hallways are well defined.
and the neon green trace is the route that was taken while exploring the environment. Although it cannot be seen from the finished map the localization of the Hector SLAM algorithm is very good, I was almost always at the exact spot in the real environment as it was predicting me to be in the map. The only real problem I can see with using LiDAR to implement SLAM on the quadcopter is that the LiDAR doesn’t detect windows very well. As can be seen from the Figure, when a window is encountered it perceives it as an open path, which is bad, if you were deploying this system in a real world environment the quadcopter could be crashing through a lot of windows. However this issue could be resolved by placing another ultrasonic range sensor on the front of the quadcopter to help it detect windows better.
CHAPTER 7

CONCLUSION AND FUTURE WORK

This thesis presents the building and developing of an autonomous indoor drone system. It provides the theory behind how quadcopter technology works as well as the theory behind the indoor positioning system SLAM. The thesis continues by giving an in-depth explanation of all the hardware and software that make up our drone system as well as how it all fits together. It then presents our proposed solution to autonomous control of a quadcopter system as well as all the difficulties and setbacks that were encountered along the way.

While I would like to be able to get more autonomous control out of this system, future works on this project include: using the data from the RPLidar laser scanner unit for object avoidance, using the map that is constructed from Hector SLAM to instruct the quadcopter to travel from one point to another, as well as adding point-to-point local path planning to the system. With advancements in computer technology and the emergence of drone systems in recent years it will be exciting to see what this technology brings in years to come.
BIBLIOGRAPHY
BIBLIOGRAPHY

[1] https://www.getfpv.com/dji-flamewheel-f450-arf-kit.html?gclid=CPfFsS7chtQCFQwZgQod90s1cg.


VITA

HUNTER GOSSETT

Huntergossett10@gmail.com

EDUCATION

B.S, Computer Science, University of Mississippi, May 2016

TEACHING EXPERIENCE

Teaching Assistant, 2016 - 2018
University of Mississippi
Courses: Java 3, Databases, Matlab

Research Assistant, Summer 2017
University of Mississippi
High Performance Computing

HONORS and FELLOWSHIPS

Upsilon Pi Epsilon, 2016 - present
Graduate Student Council Senator, 2017 - 2018