© 2015 Adam Burns

AUTONOMOUS GROUND-BASED ROBOTIC NAVIGATION FOR AN AGRICULTURAL ROW CROP ENVIRONMENT

BY

ADAM BURNS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Civil Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Joshua Peschel

ABSTRACT

The research presented in this thesis is focused on the navigation technique for an autonomous ground-based robotic system for use in an agricultural row crop environment. The performance of the navigational system was evaluated by measuring the offset of the robot's path to a predetermined path. It is found through a total of ten field tests that utilizing highly accurate GPS systems results in the greatest navigation accuracy, with an average offset of 3.56-inches.

During the agricultural growing season, many row crops produce a canopy that restricts the ability to observe and measure the various atmospheric and biological processes that take place beneath the canopy, affect the various growth stages of the plant, and ultimately alter the crop yield. The increasing use of unmanned aerial vehicles (UAV) for agricultural purposes has increased our ability to monitor crop growth. Mainly due to cost limitations, however, most studies on the interaction of environmental factors on plant growth are focused on end-point measurements. Ground-based robotic technologies provide a new method for obtaining measurements that give insight into the effect of environmental factors that affect plants during many different stages of a plant's growth cycle. Furthermore, much more frequent analysis and modeling of the crops can be obtained using a ground-based robotic approach. This allows for more accurate yield estimations as the great number of varying conditions make yield estimations derived from fewer measurements much more difficult and complex.

One of the greatest drawbacks to using a UAV approach to monitor and estimate crop growth and yield is the lack of sensing in the sub-canopy environment. Other drawbacks include the necessity for high-cost localization hardware used to facilitate navigation. In order to overcome the limitations of aerial-based sensing, this work proposes a low-cost ground-based solution for sub-canopy monitoring. This research focuses specifically on the rover navigation technique, which is a main aspect in the foundation of the proposed project.

The navigation method employed in this study was evaluated in both laboratory and agricultural settings. This was, in part, an effort to help simulate the various terrain and environmental conditions that may be experienced in a real life setting. Utilizing various types of navigation methods, the ability of each method to successfully navigate through the rows of a field was quantified by analyzing the deviation from the ideal path, or a straight line, as commonly seen in row crop settings.

A total of ten straight-line tests were conducted, each with slightly different navigational parameters and configurations. GPS waypoints were used to instruct the robot to drive in a straight line for 10-meter segments.

The results of this study indicate that a Real Time Kinematic (RTK) GPS system provides the greatest accuracy and ability for row crop navigation, with an average offset from the desired path of 3.56-inches. This solution also provides an opportunity for applying ground-based navigational solutions for various projects that may require frequent and detailed measurements obtained by on-board sensors. This research is important to researchers because it provides a low-cost autonomous robotic navigational system that can be used in a wide range of projects, such as the continuous monitoring of the sub-canopy environment of a row crop field.

ACKNOWLEDGMENTS

This project would not have been possible without the guidance and support of my adviser, Professor Joshua M. Peschel. His advice was invaluable to my education and research of the topics discussed in this work. Many thanks to Professor Peschel for his continuous encouragement and confidence in my knowledge throughout my master’s program at the University of Illinois at Urbana-Champaign. It was through his continuous efforts of enriching my academic and professional development that I was able to enhance my own personal abilities and skills.

I would also like to thank my family for their undying support of me throughout my academic ventures. Thank you for always being there for me. Your support has not gone unnoticed, and it continuous to be the main driving force behind my successes to this day.

Lastly, I would like to extend my greatest thanks to my fellow colleagues, Elizabeth Depwe, Meng Han, Christopher Chini, and Jeff Wallace. Thank you for providing me with endless support and encouragement, and for making my time in the Human Infrastructure Interaction Laboratory so pleasant and inviting. I will cherish the great memories that we shared together during my time here.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Research Question

Studies of the various processes that occur beneath the canopy of row crops that ultimately affect the growth stages of the plants require a means of ground-based monitoring. Current methods of monitoring are restricted by high costs and vision-based limitations. A potential solution is an autonomous ground-based vehicle equipped with sensors. In order to accomplish this task, a navigation system that is capable of autonomously traversing the rows of crop fields is required. The ability for an unmanned vehicle to navigate through the rows of crops is a challenging task. Furthermore, doing so using low-cost equipment only further complicates this task. No currently available solutions for low-cost navigation systems for use in a row crop environments are available, which would provide means for collecting the high resolution sensor data required to conduct adequate studies of the sub canopy environment. It is for these reasons that a method for autonomous ground-based robotic navigation of an agricultural sub-canopy environment is necessary.

Currently, there exist a number of methods used for autonomous navigation. Depending on its usage, each has its own advantages and disadvantages. For example, some methods may work much better for indoor settings while others may work better in outdoor settings. Although multiple methods may achieve similar results, it is important to identify the most adequate navigation method for a row crop settings.

To the best of the author’s knowledge, there currently exists no proven reliable low-cost solution for row crop navigation by ground-based autonomous vehicles performing sub-canopy monitoring. Therefore, it is necessary to conduct a study that will determine an appropriate method for facilitating ground-based robotic navigation.

This study proposes a specific method for ground-based robotic navigation of row crops using RTK GPS and 2-dimentional LIDAR. The goal is to identify the ideal method and configuration for row crop navigation. Identifying the various adequate components of this project will be aided by the quantification and comparison of navigation techniques.

1.2 Why Focus on Ground-Based Navigation of Row Crops

During the agricultural growing season, many row crops develop a canopy which prevents unmanned aerial vehicles (UAVs), such as quadcopters and fixed-wing drones, from analyzing the near-ground environment. It is of great importance to be capable of analyzing the sub-canopy environment in order to better understand the various parameters that effect crop growth and yield.

Continuous monitoring of the sub-canopy may be achieved using networks of fixed sensors. High costs associated with the installation and maintenance of large networks of sensors restricts the feasibility of this approach. Reliable monitoring data is an essential component of this area of research. The advantages to using a ground-based robotic method for sensing would be high quality monitoring data at a low cost.

1.3 Importance to Civil and Environmental Engineering

The importance of the proposed study to Civil and Environmental Engineering is the lack of ground-based navigation methods that can be used to further investigate and understand the issue of water scarcity. More specifically, it is essential to accurately analyze and sense the sub-canopy environments of row crops in order to better understand the impacts of various environmental parameters that can affect the availability of water. These tasks can be accomplished by using sensors mounted directly on an autonomous ground-based robotic system. Furthermore, a method that is low-cost has also been lacking from the Civil and Environmental Engineering discipline.

To environmental engineers, the proposed study would serve as a sufficient method to further analyze an agricultural sub-canopy environment, allowing for further discoveries to be uncovered. To civil engineers, on the other hand, the proposed study would serve as an inspiration for implementing low-cost autonomous vehicle solutions for various projects in their field of study.

To hydrologists, the proposed study may provide means for new and emerging monitoring technologies to improve the quality of models. Sensor data obtained from ground vehicles navigating through row crops may provide superior measurements of various model parameters, such as evapotranspiration. The mobility of a ground-basic robotic sensing system increases the coverage of sensing, therefore increasing the amount of high resolution data available for studying water resources.

1.4 Contributions

The contributions of this study include the analysis of autonomous navigation methods that would be suitable for agricultural sub-canopy environments of row crops. A robotic solution that can autonomously traverse the rows of a crop field is presented, utilizing low-cost components. Highly accurate GPS systems increase the navigational accuracy of a robotic system, and 2-D LIDAR systems can be used to obtain localization data with high resolution. Contributions into the tuning of a PID controller used for the steering of a ground robot is also investigated in detail.

Additionally, this study opens up the possibility of using ground-based robotic navigation methods for various projects that fall in the category of Civil and Environmental Engineering studies. The aspects of continuous and mobile monitoring at a low cost make the work presented very adaptable to a wide range of potential projects and research studies.

1.5 Organization of This Thesis

The organization of this thesis is as follows. Chapter 1 includes the research questions of ground-based robotic navigation of agricultural row-crop settings, the importance of such a method, and the contributions of this study. Chapter 2 details the prior research conducted on similar projects, the current limitations and issues, and the various hardware solutions available. Chapter 3 presents the various autonomous navigation methods and their applicability to this thesis. Chapter 4 describes the implementation steps and procedures taken in this study to determine the ideal navigation method. Chapter 5 discusses the results the tests and the limitations observed. Chapter 6 includes the conclusions and findings of the research presented in this study, as well as any necessary future work.

CHAPTER 2
LITERATURE REVIEW

The goal of this study is to determine the ideal autonomous ground-based navigation system for use in agricultural row crop environments. The literature review section will discuss the abilities and limitations of current methods used for agricultural navigation by autonomous robotic systems. A review of the critical navigational sensing components most commonly utilized for autonomous navigation is additionally conducted.

2.1 Current Methods

In order to achieve autonomous robotic navigation, many methods have been employed, most utilizing some form of localization or robotic mapping. In some cases, both methods are used simultaneously. In the localization approach, the robot is able to determine its current heading and its desired path or target heading. Alternatively, by mapping the environment, a robotic system can locate and navigate around obstacles. When these approaches are combined, often referred to as Simultaneous localization and mapping (SLAM), obstacle avoidance and robotic mapping can be achieved simultaneously.

In recent years, significant research and documentation has been focused on autonomous path following. Comparisons of various path following techniques have been conducted, analyzing the performance of such techniques as Non-Linear Guidance Law, Pure Pursuit with Line of Sight, Vector Field Pursuit, and Linear-Quadratic Follower in [1], [3], and [9].

The use of offset from a determined path as an input to a PID controller has been explored, with inconclusive results [4]. The complexity of this approach is discussed in greater detail in the methodology section of this thesis.

Other approaches to autonomous navigation include the use of artificial intelligence for the purposes of path planning. Research conducted utilizing both learning and non-learning systems.

A non-learning neural network system that was capable of achieving collision free path navigation has been successfully implemented in [10]. Furthermore, it is important to note the investigation into artificial potential field algorithms designed to navigate along crop rows. This approach had significant drawbacks, such as trapping caused by local minima in measured sensor readings [11].

2.2 Issues with Robotic Mapping

The acquisition of a robot's environment, specifically for use in navigation, is a process that combines the use of various sensors that allow for the creation of a spatial model of the environment [8]. Sensors that are commonly used for this purpose include cameras, LIDAR, sonar sensors, GPS, compasses and magnetometers. It is important to note the various drawbacks and limitations of this approach. Like most sensors, there exists a certain range of measurement error which decreases the accuracy of a robotic navigation system. Additionally, limitations on sensing through objects or obstructions, such as walls, may limit the ability to map the environment. As a result, robotic mapping has been shown to be a key challenge to robotic navigation as sensor errors can accumulate over time [6].

Another key challenge to robot mapping is changing environments. In the specific case of a robot navigating through an outdoor field, issues related to the dynamic nature of the environment as the wind may cause swaying and movement in the vegetation may be encountered. This, in effect, can produce inconsistent sensor measurements, especially in rapidly changing environments. Since computational modeling of such dynamic environments is extremely difficult due to factors that are outside of the scope of this research, navigation methods must find alternative methods to account for these situations.

2.3 Sensors

The use of sensors is a critical component of autonomous robotic navigation. Due to the combination of both a wide range of currently available technologies and the current rate of sensor development, this section is focused on the discussion of relevant examples of proven sensor integration, as determined to be appropriate for this study.

2.3.1 IMU

The inertial measurement unit, IMU, is used to measure sensor data that is directly used by the flight controller in order to determine the attitude of the vehicle. The sensors involved in this process includes accelerometers, magnetometers, and gyroscopes. As opposed to an attitude and heading reference system, AHRS, an IMU requires off-board processing of sensor data in order to calculate vehicle attitude. Non-linear estimations are typically used in conjunction with sensor data to calculate the attitude [2].

Using the process of dead reckoning, it is possible to calculate the current position of a vehicle using only the last determined GPS location, the calculated velocity, and the time elapsed [12]. The advantages of this process include reduced processing time due to the relatively small amount of required sensor data, the capability to conduct underground navigation when technologies such as GPS are unavailable, and reduced cost. One of the largest drawbacks of this method is the potential for accumulated error that can result in miscalculations of current location on the scale of several meters due to inaccurate sensor readings [5]. The cause of the inaccurate sensor readings can have a wide impact and is discussed in more detail in following sections.

2.3.2 Camera

Object recognition, which is possible with the use of cameras, has the capability of greatly enhancing the accuracy of robotic navigation. Continuous analysis of camera images as an autonomous vehicle travels has been shown to be capable of correcting for errors in target path trajectory [3]. This can be done through a variety of approaches. One potential application of a camera-based navigation system uses multiple sequential images captured from the camera, and compares the changing of the scenery to determine the current vehicle heading and speed. Further uses of this technology include the use of object recognition for obstacle avoidance purposes [7].

2.3.3 LIDAR

The proper operation of a LIDAR device can produce highly accurate distance measurements through the use of pulses of light emitted from the module [16]. This is accomplished through the measurement of the amount of time that has elapsed between the time that light pulse has been emitted and the time that the returned light pulse is detected, along with the knowledge of the speed of light. Continuous and rapid development of LIDAR devices has resulted in a wide range of available technologies. For the purposes of this research, certain LIDAR modules are investigated in greater detail, as detailed in upcoming sections.

2.3.4 Encoder

A common technique employed to measure the velocity, acceleration, and angular position of a vehicle relies heavily on encoders. In combination with a light detector, an encoder typically operates with the assistance of a disk that consists of black and white segments that are attached to the axel of a vehicle. As the wheels rotate, the light detector is able to measure the pulses of each color of the disk. Using displacement measurements, the current velocity, acceleration, and angular position of the vehicle can be calculated using the information obtained by the encoder [16]. The various types and operation methods of encoders vary greatly and are outside of the scope of this thesis, and will therefore be omitted. It is important to note, however, that this technique is a cost-effective method for determining vehicle positioning information.

2.3.5 RTK GPS

Real-time kinematic (RTK) GPS systems can provide highly accurate location data with typical accuracy in the range of centimeters [13]. When compared to the accuracy of typical GPS systems, which is on the scale of meters, the RTK GPS method has an obvious advantage for navigation and localization purposes. With the assistance of a fixed ground station, position data is calculated with the measurements of GPS satellite carrier wave phase. This is done by referencing the carrier phase of the ground station with that detected by an on-board RTK GPS module [14]. In the following sections of this thesis, the operation of a RTK GPS system is expanded upon further, along with a discussion of the advantages of this technology in the context of an autonomous ground rover for use in an agricultural setting.

CHAPTER 3
METHODOLOGY

The approach described within this section details the techniques employed to achieve autonomous robotic navigation through row crops in an agricultural setting. Beginning with a discussion on the navigation technique, this thesis outlines the proposed solution, followed by a detailed description of the software utilized in this thesis.

3.1 Navigation Control Architecture

Through the combination of an IMU, RTK GPS unit, LIDAR module, and a Pixhawk flight controller, it is proposed that autonomous robotic navigation of an agricultural row crop environment can be achieved. As shown in Figure 1, the navigation control architecture consists of three main sources of sensor data acquisition and processing. First, the on-board IMU provides the Pixhawk flight controller with sensor data that is used to calculate the attitude of the vehicle. Second, an RTK GPS module provides centimeter-accurate GPS information to the flight controller. Lastly, a LIDAR module was implemented in order to assist in obstacle avoidance. This last step required the preprocessing of the raw LIDAR measurements, and was achieved using a Raspberry Pi, which calculated the lateral offset of the vehicle, and used this offset as the error for a PID controller. Drawbacks and limitations caused by the LIDAR system were significant and will be discussed in greater detail in upcoming sections.

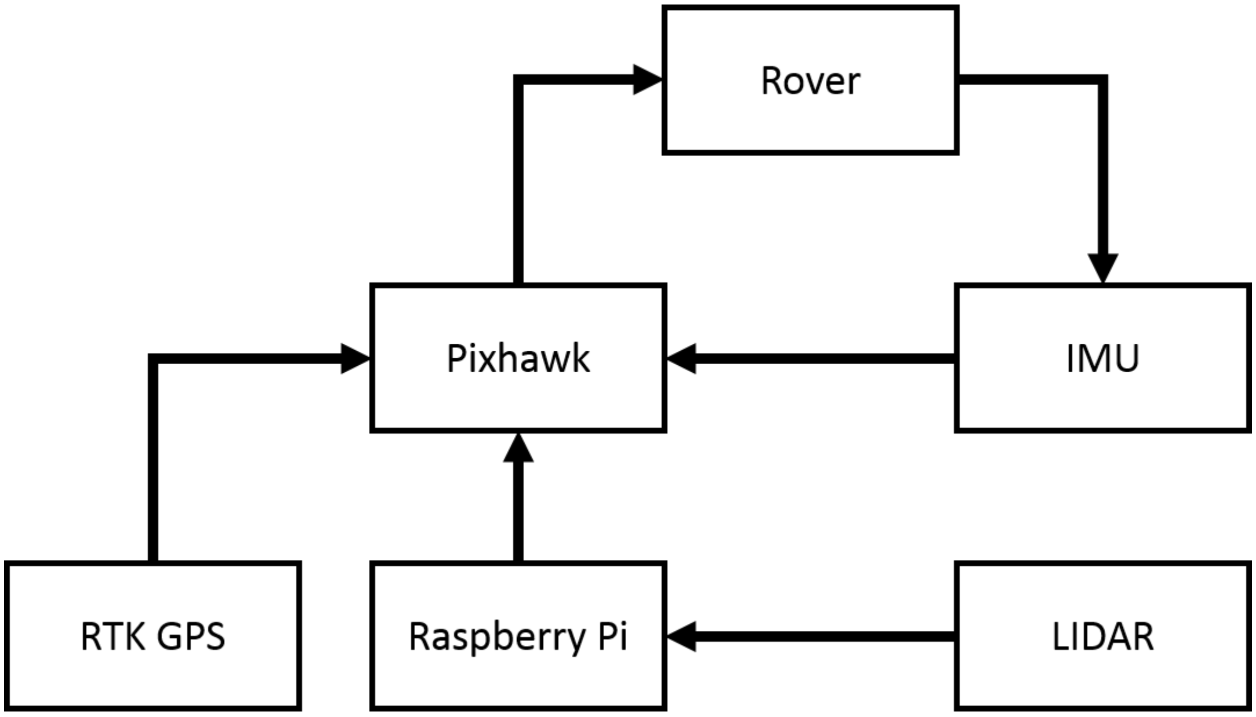


Figure 1 – A diagram of the navigation control architecture utilized by the rover. Shown on the bottom left-hand side of the figure, the RTK GPS measurements are directly sent to the Pixhawk autopilot system. Additionally, LIDAR readings, as shown on the bottom right-hand side of the figure, as well as IMU readings, shown directly above, are also sent directly to the Pixhawk module. The LIDAR readings are sent to the Pixhawk through a Raspberry Pi module. The Pixhawk autopilot system uses the readings from all three modules to steer the rover, shown in the top of the figure.

3.2 Pixhawk

The Pixhawk autopilot system is used as the primary control device for the autonomous ground rover. All control commands to the motors of the rover are directly sent from the Pixhawk. This module consists of a Cortex-M4 core, PWM outputs, and various ports for communication including UART and I2C. Additionally, an ST Micro L3GD20 3-axis 16-bit gyroscope, an ST Micro LSM303D 3-axis 14-bit accelerometer / magnetometer, an Invensense MPU 6000 3-axis accelerometer / gyroscope, and a MEAS MS5611 barometer are included in the system, providing means for calculating vehicle attitude. The Pixhawk system is also capable of supporting multiple external sensors, such as the RTK GPS module that is discussed in the following sections.

3.3 Raspberry Pi

A Raspberry Pi B was used for preprocessing of sensor data for use with the Pixhawk flight controller, as well as for running an additional PID controller that used the offset information obtained from the LIDAR module. The primary objective of using such a system is to increase the computing ability of the navigation system, beyond that of the Pixhawk flight controller alone. The MAVLink protocol is utilized to facilitate communication between the Raspberry Pi and the Pixhawk over a serial connection. In order to support a PID controller written in Python, the MAVProxy package was used in conjunction with the MAVLink protocol to serve as a ground control station (GCS).

3.3.1 MAVLink

Communication between the Raspberry Pi and the Pixhawk autopilot system is facilitated with the use of the MAVLink protocol. All communication data is transmitted with an FTDI cable, which connectes the two modules. A baud rate of 1.5Mbit proves to be a stable rate of communication. Compiling of the MAVLink library is not needed as this is a header-only library. Software packages that are required to run the MAVProxy module are python 2.8, numpy, libxml, libxslt, and pymavlink. Other necessary configuration steps for the Raspberry Pi include disabling the OS control of the serial port and creating a script that automatically initializes the MAVProxy module whenever the Raspberry Pi is powered on. The MAVProxy module is then easily implemented.

The motivation for using the MAVProxy module is to create a method for continuous monitoring of IMU sensor data for use with the PID controller designed for use with the LIDAR module. The most important sensor information obtained with the use of the MAVProxy module is the rover heading and GPS coordinates, as these gave the best information relating to the current trajectory and the desired path of the rover. Predetermined GPS waypoints are used to instruct the rover where the beginning and end coordinates of each row of the field are located. With the current location of and heading of the rover, it is reasonable to assume that an adequately accurate estimate of the current rover trajectory is possible to be calculated. Using these estimates, along with the LIDAR readings, commands for correcting the steering are sent to the Pixhawk autopilot system. The corrections are in the format of heading, in degrees. The

Pixhawk automatically translates these corrections into motor commands, which results in the rover turning.

3.3.2 LIDAR

A Robopeak RPLIDAR 360 degree Laser Scanner Module is used to obtain measurements of the distance between the rover and the rows of the crop field. The LIDAR module is advertised as having a distance resolution of 0.2cm and an angular resolution of 1 degree. With a sample rate of 5.5Hz and a range of 6 meters, it is assumed that this setup will provide sufficient readings necessary for the avoidance of collision with the crops. The LIDAR module is mounted on the top of the rover, at a height of approximately 12-inches above ground level (agl). 5 volts is provided to the LIDAR module directly through the USB connection from the Raspberry Pi. This voltage was used to operate the scanner system, as well as the motor system. An on-board voltage regulator maintains a 3.3V signal for use with the UART serial port communication interface. The configuration of this system is outlined in Figure 1.

The RPLIDAR module operates by emitting an infrared laser pulse and sampling the returning signal with a vision acquisition system. Distance and angle data is calculated by an on-board digital signal processor (DSP). A representation of this process is shown in Figure 2.

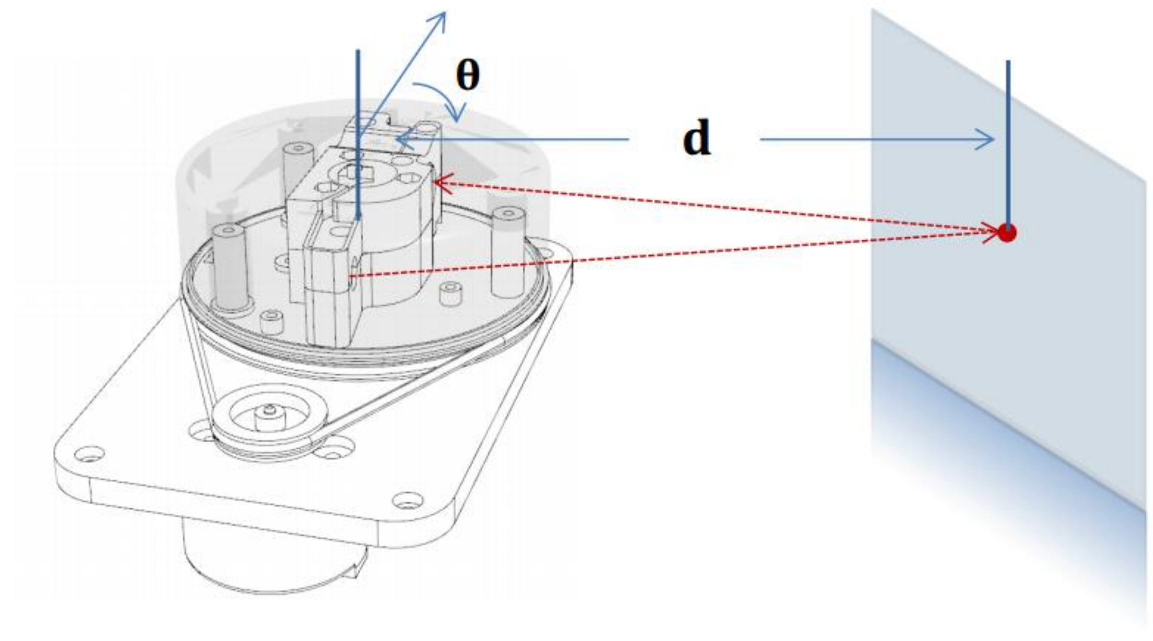


Figure 2 – RPLIDAR distance and angle measurement diagram [11]. The RPLIDAR module, shown on the left, emits a laser beam at an angle, θ. The beam is then reflected off of a surface, shown on the right. The delay between the time that the laser is emitted to the time that it is received is used to calculate the distance, d.

3.3.3 PID Controller

Attempts were made to incorporate a custom Python PID controller that ran on the Raspberry PI and used the offset, determined by a RPLIDAR module, as the error. Due to various complications, including electromagnetic interference and noise caused by occlusions in the agricultural row crop environment, results are inconclusive. A discussion of this aspect of the thesis is included in Chapter 5. Despite the lack of decisive results from the implementation of the PID controller for use with the RPLIDAR module, it is important to review the PID controller concept as it plays a central role in the control system employed by the Pixhawk autopilot system.

A block diagram of a typical PID system is shown in Figure 3. Mainly consisting of a control loop feedback circuit, PID controllers are often used for holding a system at a predetermined setpoint. The output of the controller, as shown in Figure 3, is the sum of the proportional control (P), the integral control (I), and the derivative control (D). This can be denoted as:

$$output(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \qquad (3.1)$$

Where $e(t) = setpoint - process\ variable$, K_p is the proportional gain, K_i is the integral gain, and K_d is the derivative gain.

The derivative control component of the PID controller produces a response to the rate of change of the system. This is done by multiplying the derivative gain, K_d, by the derivative of the error, e. In effect, the derivative control will act to slow the response of the controller, which is helpful in situations that involve overshooting.

The integral control component generates a response to the magnitude and the duration of the error. The second term on the right hand side of equation 3.1 describes this process, as the integral of the error over time t is multiplied by the integral gain, K_i. Exceedingly high integral gain values can lead to oscillation of the system due to the overaggressive nature of the response.

The proportional control component, as shown as $K_p e(t)$ in equation 3.1, creates a response that is proportional to the error of the system. The correct tuning of the proportional gain, K_p, is essential for stable performance. A gain value that is too large will produce oscillations, while a value that is too low may not produce enough response to adequately correct for the error. Although many various tuning strategies exist, many techniques used for tuning begin with the proportional gain, as it often the primary indicator for system stability [17].

The input for a PID controller is the error. In the specific case of this project, the error is the offset, or the difference in distance between the row crops to the left and to the right of the rover. Overshooting was minimized through derivative control and steady state performance was improved with integral control. This general method can be a suitable solution for other terrain-following objectives, including those that utilize various range finding techniques. Using a GPS unit as the primary direction controller, a PID control method can be implemented to improve the driving accuracy of a robot traveling the rows of a crop field. In this scenario, the error detected by the control method will be supplemented with the GPS coordinate error. Proper implementation may allow for a more reliable and robust steering control. In theory, it would be reasonable to assume that an improvement in the quality of the control system could be obtained by using more advanced localization technologies. Uncertainty exists in the expected amount of improvement in the PID control method from using a LIDAR system for the range finding component, the improvement in GPS accuracy with the use of GCP (ground control points), and the applicability of the method to more noisy environments such as crop rows. Further investigation is required to adequately analyze the effectiveness of a PID control method without the use of GPS coordinates. For detailed reference of the PID controller designed for this application, the full code is provided in Appendix A.2.

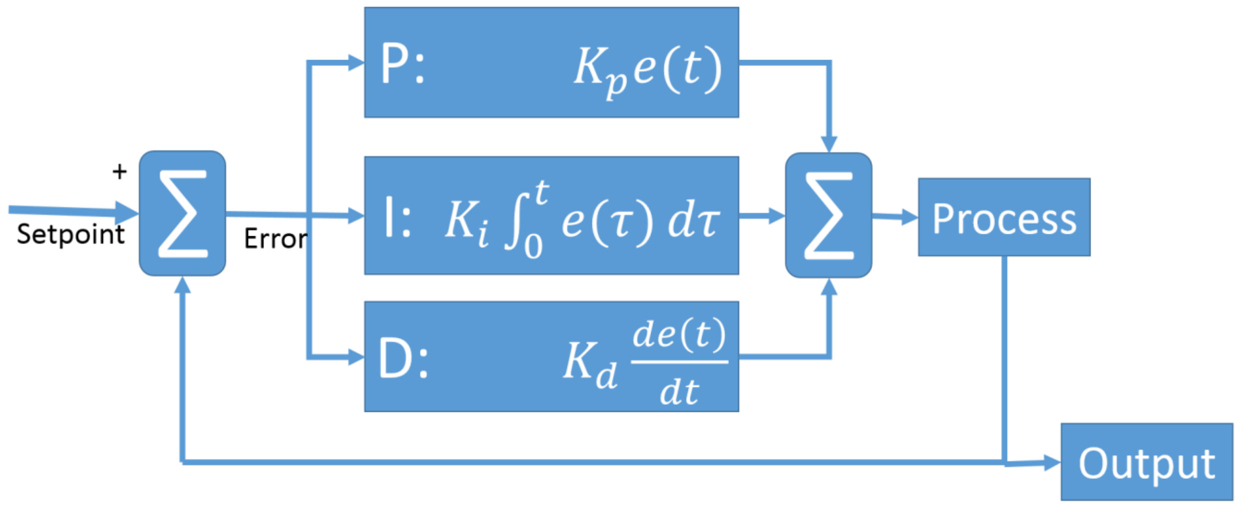


Figure 3 – A PID controller block diagram. Starting from the left-hand side of the diagram and moving to the right, a setpoint is referenced with an actual measurement. This is classified as the error. Next, the error is input into three separate components. The error is used in the proportional control, P, the integral control, I, and the derivative control, D, to calculate the three values of error. These three error values are summed together and sent to a process, which results in the total error being sent back to the beginning of the system where it is summed together with the setpoint error.

3.4 RTK GPS

One of the most accurate GPS technologies currently available for use in autonomous navigation systems is the real-time kinematic (RTK) GPS. With an accuracy on the scale of centimeters, this technology allows for extremely high precision localization [13]. The Piksi GPS receiver system produced by Swift Navigation is utilized in this thesis. This system provides position and velocity calculations at a rate of 50 Hz. As shown in Figure 4, the system architecture for the Piksi receiver consists of the RF front-end, which uses a Maxim MAX22769 down-converter along with a 3-bit analog to digital converter (ADC) to downconvert and digitize the received radio frequency signal. Controlled by an ARM Cortex-M4 DSP core, the SwiftNAP component performs the filtering and correlation operations on the signal, which is required to calculate position, velocity, and time solutions.

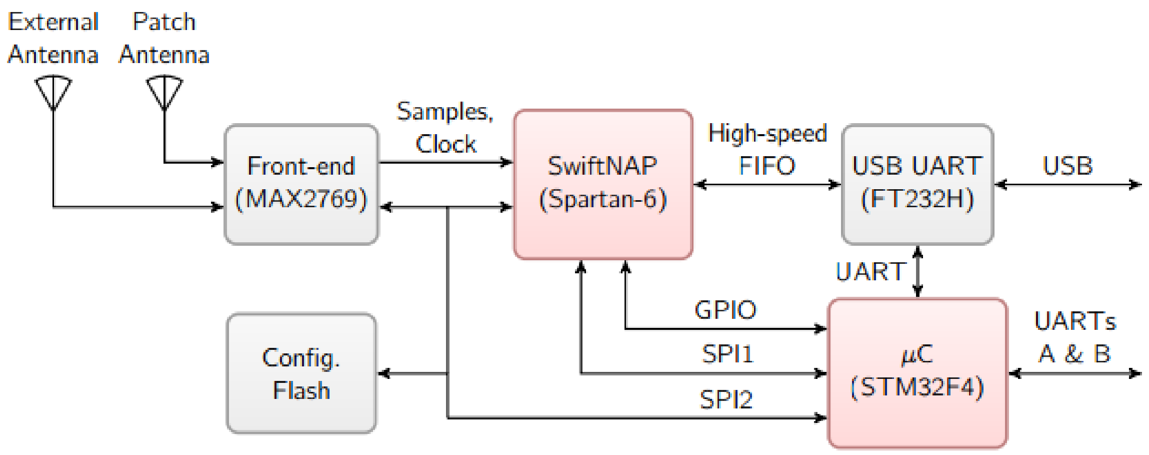


Figure 4 – A block diagram of the Piksi system [19]. A radio frequency front-end, shown on the left, consists of the patch and external antennas and the Maxim MAX22769 module. The radio signal is downconverted and sent to the SwiftNAP core, shown in the center, where filtering and other correlation operations are conducted. The results of these operations are sent to the USB and STM32F4 modules, which utilize the UART communication protocol to transmit the data, as shown on the right-hand side of the figure. Lastly, a flash memory module is used to store configuration parameters.

3.4.1 Rover RTK Module

A Piksi RTK GPS receiver is attached to the vehicle such that the receiver has an unobstructed view of the sky at all times. This is an essential component of the overall autonomous navigation system as the quality of the RTK GPS measurements is directly related to the number of satellites that the receiver can detect. Obstructions between satellites and the receiver can cause the loss of signal packets, which results in significantly decreased localization accuracy.

3.4.2 Base Station

A second Piksi receiver is mounted to a tripod with a height of 6-feet above ground level and is placed in a fixed location with a surveyed latitude and longitude information. For reference, a photo of the basestation setup is provided in Figure 5. The Piksi system used this information, along with the carrier wave phase comparison, to determine the precise distance between the ground rover and the base station. This was a critical component to the navigation system, as the Pixhawk was able to interpret this distance data and translate it into useful location data. One of the greatest advantages to using this system has to do with the extremely high accuracy, which allows for an increased quality of vehicle attitude and velocity calculations.

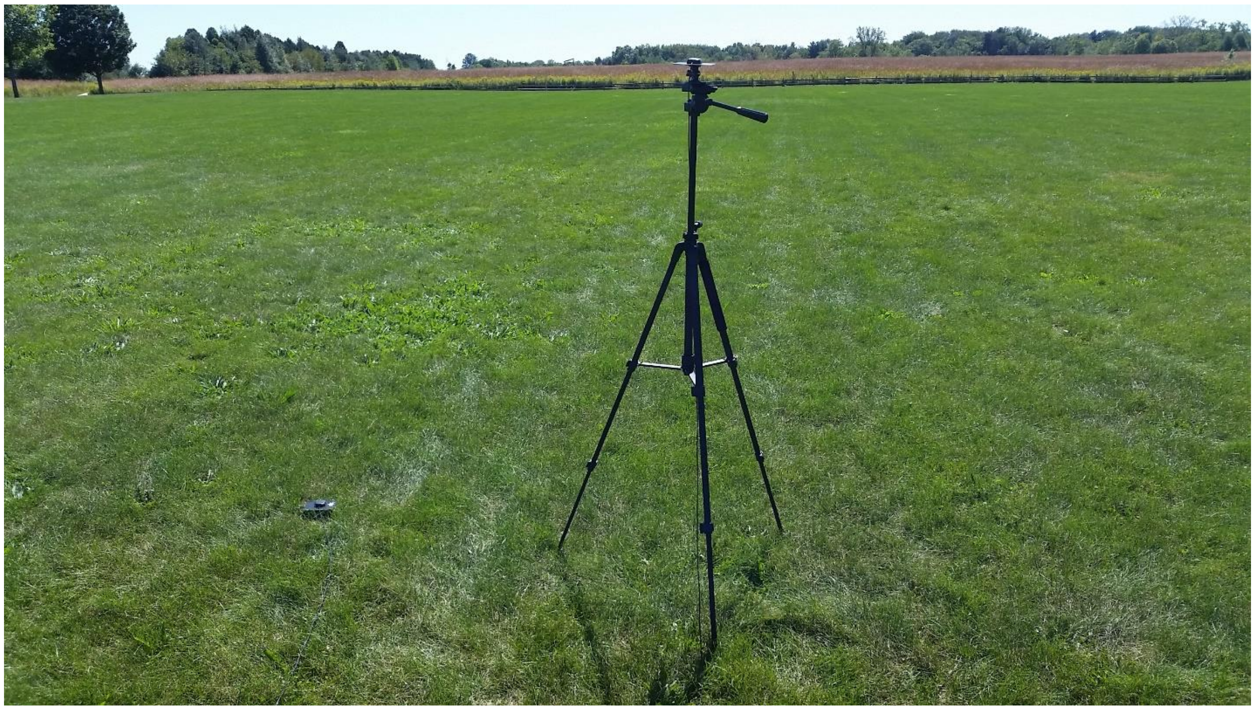


Figure 5- A Piksi basestation mounted on tripod in an open field. This was used as the primary GPS sensor for the ground control software. A tripod was required to minimize obstructions in order to enhance the reception quality of the GPS module, which improved the overall accuracy of the system. Stakes were used to secure the legs of the tripod into the ground in order to prevent wind gusts from knocking the tripod over.

3.4.3 GPS Injection

The last required step in order to successfully implement the RTK GPS into the navigation system is the use of GPS injection with the Pixhawk. This is achieved through the use of RF transmitters and a ground station running the Mission Planner software created by Michael Oborne, a developer based in Australia, along with the Piksi Console software developed by Swift Navigation.

SBP Observation Messages are obtained by the ground station using the Piksi Console, which is running on a laptop with Windows 8.1. In order to forward these messages along to the Pixhawk, the Mission Planner software is used to inject the GPS information. Lastly, this information is sent from the ground station computer to the ground rover using RF transmitters provided by 3DR Robotics.

3.5 Straight-Line Tests

Straight-line tests are performed in order to evaluate the accuracy of navigation. These tests are conducted by first creating a path for the rover to navigate. Using predetermined GPS waypoints, a 10 meter path is created. The rover is instructed to drive between the waypoints, with the objective being to reach the destination while minimizing the total offset from the desired path. As discussed in the implementation section of this thesis, a rope was anchored between the two GPS waypoints in order to aid in visual evaluation of the rover’s deviation from the path. In addition, log files were created during all testing scenarios, which consists of all sensor readings, including the RTK GPS data. Processing of these log files is used to validate the visual observations.

CHAPTER 4
IMPLEMENTATION

The implementation of the methodology discussed in Chapter 3 is the primary focus of discussion for this chapter, beginning with an overview of the specifications of the ground rover that is used for testing in this thesis. Furthermore, the execution of the control system architecture is explored, in addition to the various tests employed to measure the performance of the system.

4.1 Robot Specifications

The ground robot primarily utilized for testing and development of this project is a tank-tread style rover. Two treads are independently operated by separate motors. The width of the rover is 12-inches. Depending on the size of the payload attached to the chassis of the robot, the overall weight can vary. On average, the total weight is approximately 10-pounds. Diagrams of the ground rover are provided in Figure 6, Figure 7, and Figure 8. More detailed specifications of the rover is provided below.

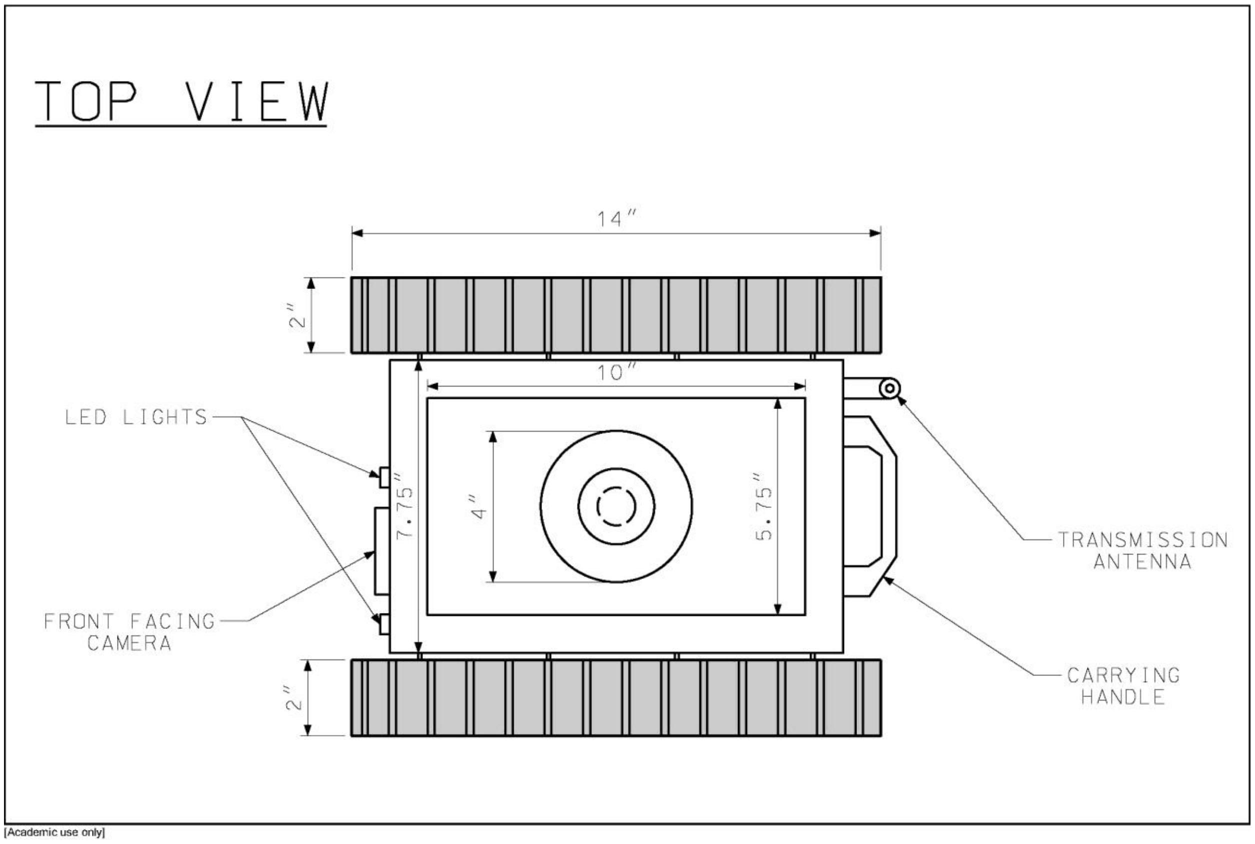
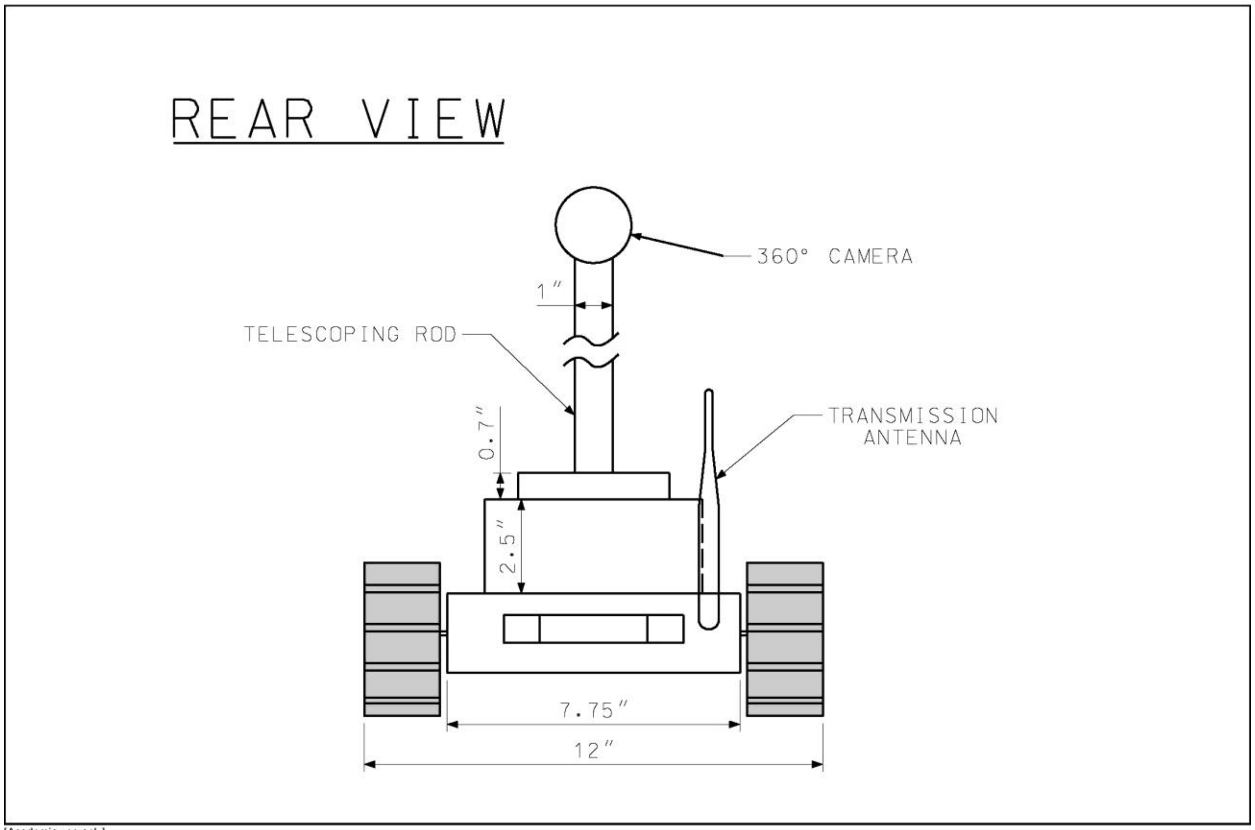


Figure 6 – Top down view of ground rover. The front of the rover is pointing to the left. Attached to the front of the ground rover is a front facing camera along with a pair of LED lights. The rear of the rover, shown on the right hand side, has the transmission antenna and a transportation handle. The width of the tracks are 2-inches, and the length of the tracks are 14-inches.



[Academic use only]

Figure 7 – Rear view of the ground rover. The back of the rover is shown, along with the dimensions of the chassis and the overall rover. These dimensions are 7.75-inches and 12-inches, respectively.

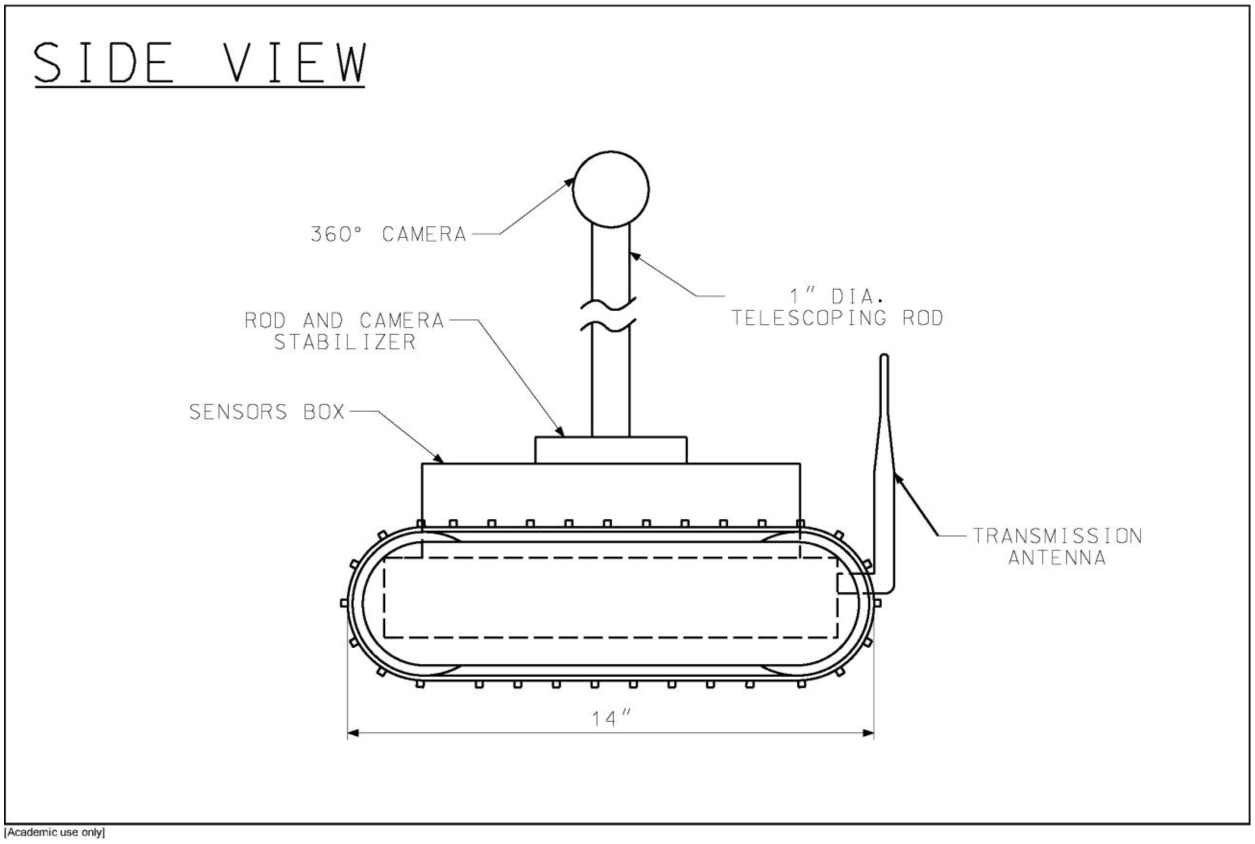


Figure 8 – Side view of the ground rover. The length of the treads are shown as 14-inches long. Also shown is the transmission antenna, the sensor box, the rod and camera stabilizer, and the 360-degree camera. The rover is facing such that forward travel of the vehicle would be observed as moving to the left.

4.1.1 Chassis

As the main goal of the rover is to safely navigate through a row crop environment, compact chassis dimensions are of great importance. Entanglement in plant life is a major issue for robotic navigation of agricultural environments [12]. It is for these reasons that the vertical profile of the chassis remain as small as possible. By integrating the mechanical elements of the rover into the chassis in a compact fashion, the height is approximately 4-inches. The frame of the rover is constructed of steel and is in a flat rectangular box configuration, providing ample room for electronic and mechanical components.

4.1.2 Motors

The two motors used to drive the rover are RS550VC-7527 and are produced by Mabuchi. The specifications of the motors are shown below in Table 1. For agricultural robotic navigation purposes, these motors proved to possess sufficient torque to avoid issues such as becoming stuck due to uneven terrain. Each motor independently drove a tread on the rover, requiring two separate motor signal inputs.

Table 1 – RS550VC-7527 motor specifications. The current, measured in Amperes, the speed, calculated as rotations per minute, and the Torque, measure in gram-centimeters, are provided for the three cases of no load, maximum efficiency, and stall.

	Parameter	Value	Unit
No Load	Current	1.3	Amperes
	Speed	19800	rotations/minute
Maximum Efficiency	Torque	660	gram-centimeters
	Current	10.5	Amperes
	Output	119	Watts
	Speed	17620	rotations/minute
Stall	Torque	5994	gram-centimeters
	Current	85	Amperes

4.1.3 Motor Driver

In order for the Pixhawk autopilot system to control the motors, a motor driver is necessary. This module acts as an encoder, translating raw PWM values from the Pixhawk to the appropriate voltage levels which are sent to the motors. The motor driver used in this study is a Sabertooth 2x12 v1.00, made by Dimension Engineering. This driver can supply 12 Amps to two DC motors simultaneously, with peak instantaneous currents of 25A. The synchronous regenerative capability of the motor driver allows for automatic battery recharging during vehicle braking.

4.1.4 Power Source

As the power source of any robotic system is one of the most critical components, redundant power systems are employed. This aids in system operation stability and overall operation lifetime. Two 30C 2200mAh 11.1V LIPO batteries are connected to the rover, one providing electricity to the motor driver and the second providing power to the navigation system and peripherals.

During the development process of implementing the LIDAR module, it was necessary to increase the battery capacity as the overall energy demand increased. This is caused by the introduction of both the Raspberry Pi and RPLIDAR modules into the navigation system, both of which requires continuous power in order to operate correctly. In order to address the issue of increased electrical demands, the navigation system’s power supply is replaced with a 40C 5300mAh LIPO battery.

4.1.5 Radio and Receiver

The radio used for manual operation of the rover in this project is a Spektrum DX6i. The corresponding Spektrum AR610 receiver is replaced by a 2.4 GHz DSMX Remote Receiver, also manufactured by Spektrum. This modification was necessary for allowing the Pixhawk autopilot system to interface directly with the Sabertooth motor driver. An alternative approach would be implementing an additional PPM encoder between the AR610 receiver and the Pixhawk.

4.2 LIDAR and PID

Initial implementation of a LIDAR module into the navigation system provided means for obtaining highly accurate distance measurements. Attached to the rover at a height of approximately 12-inches agl, a 360 degree 2D scan is acquired. The Raspberry Pi processes the raw distance data from the LIDAR module and determines the offset with respect to the center of the row. The offset is then used as the error in the PID controller. The PID controller is written in Python and is run on the Raspberry Pi. For detailed reference, the full code is provided in Appendix A.2.

For testing purposes, plungers are used to simulate the stalks of the row crops. Separated horizontally by 24-inches, the plungers are aligned in two rows. The rover is instructed to drive through the rows of plungers and use the LIDAR data for the PID controller. Results of these tests are provided and discussed in Chapter 5 of this thesis.

4.3 Pixhawk Calibration

Calibration of the navigation system's sensors was a critical component to achieving reliable results. The internal sensors of the Pixhawk, specifically the magnetometer, required the most attention to calibration, as this component is key to the navigation system's calculations of vehicle heading. Optimal navigation performance is observed directly after completing both an accelerometer and magnetometer calibration. In addition, sources of interference are discovered throughout the calibration process. A discussion of these issues are provided in Chapter 5.

4.4 Steering Controller Tuning

One of the greatest challenges to achieving reliable autonomous navigation is the tuning of the steering controller, which is built into the Pixhawk system. A trial and error approach is employed to determine the optimal parameter values. Evaluation of the steering controller performance is determined by observing the offset from a straight line as the rover attempted to navigate for 10 meters. The results of these tests are discussed in the following chapter of this thesis.

4.5 Straight-Line Tests

The evaluation of the performance of the overall navigation system is conducted through field tests where the rover is instructed to follow a direct path between two points separated by 10 meters. The deviation from the straight path was measured as the rover traversed the field, and was used as the main quantifier in this section of the analysis. Conclusions and discussions of these experiments are included in Chapter 5 of this thesis.

CHAPTER 5
RESULTS, DISCUSSION

The results of the autonomous navigation testing are detailed in this section. Quantitative deviation results are provided for each test case, as described in the previous sections. A review of the straight line tests is performed, in addition to a discussion of the results. Limitations and issues related to the LIDAR, magnetometer, and GPS modules are also reviewed and discussed in this section.

5.1 Straight-Line Test Results

The performance of the navigation system utilized for this thesis is analyzed through the use of straight-line tests. The goal is to navigate to the GPS waypoints arranged in a straight line as accurately as possible. Deviation from the straight line connecting the waypoints is the main quantifier of performance. In these experiments, the rover is instructed to follow a straight path for 10 meters, make a bowing 180 degree turn, and return along an additional 10 meter straight path. Differences in performance is observed when the proximity of the paths to the RTK GPS base station is increased. Furthermore, creating separate departure and return paths improves the accuracy of navigation. All straight-line tests were conducted in an open grass field located at Meadowbrook Park in Urbana, IL. Sky conditions were fair, with sparse cloud cover. Through the process of tuning the PID steering controller, the offset from the straight-line path is minimized. The tests performed in this section are analyzed and the results are discussed.

Using a total of 18 waypoints, the straight line paths are manually constructed using the Mission Planner Software. Figure 6 depicts this path, as the GPS waypoints is shown as green markers, and the path taken by the rover is depicted with a purple line. Although straight-line test analysis could be performed with this software, the Piski Console was used for analysis due to its increased level of GPS information accuracy. The waypoint GPS information used in Test 1 is provided for reference in Table 2.

Table 2 – Test 1 waypoint GPS information. The first column contains the waypoint number, as determined when creating the path. The second and third columns contain the latitude and longitude coordinates, respectively, and the fourth column contains the elevation information in meters.

GPS Waypoints			
Number	Latitude	Longitude	Elevation
1	40.082058	-88.201256	221.06
2	40.082043	-88.201225	222
3	40.082024	-88.201225	222
4	40.082005	-88.201225	222
5	40.081985	-88.201225	222
6	40.08197	-88.201225	222
7	40.081951	-88.201225	222
8	40.081924	-88.201241	222
9	40.081913	-88.201256	222
10	40.081898	-88.201256	222
11	40.08189	-88.201241	222
12	40.08189	-88.201225	222
13	40.081898	-88.20121	222
14	40.081913	-88.20121	222
15	40.081924	-88.201218	222
16	40.081936	-88.201225	222
17	40.081951	-88.201225	222
18	40.082043	-88.201225	222

In order to evaluate the GPS navigation offset, waypoints 2 through 7, 16, 17, and 18 were of primary interest as these points all fell on the same longitudinal line. During the departure phase, the rover travels through a total of 6 waypoints. After completing the 180 degree turn, the rover travels between just three waypoints. By testing on both highly populated and sparsely populated navigational routes, a more realistic understanding of the real life performance was obtained since applications of this technology may use either approach. The offset of the rover's GPS coordinates from the waypoints is shown in Table 3. The code used to convert GPS coordinate pairs to distance is provided for reference in Appendix A.3.

Test 1 is conducted with a single departure and return path. The rover navigates the course with the use of GPS waypoints. The PID steering controller is tuned with a proportional gain of 0.9, and integral gain of 0.2 and a derivative gain of 0.03. Figure 10 depicts the results of this test in map form. The red marker indicates the location of the RTK GPS base station, the orange markers show the path of the rover, as determined by the Piksi Console, and the blue line indicates the desired straight-line path. The GPS coordinates of the waypoints, along with the coordinates of the rover is provided in Table 3. The distance between each pair of coordinates is measured in inches. An average offset error of 4.466-inches is observed during test 1.

Table 3 - Test 1 waypoint and rover GPS coordinates. The offset between each pair of coordinates is provided in inches in column six. The first column provides the waypoint number information. The second and third columns show the latitude and longitude coordinates of the waypoints, and the fourth and fifth columns show the latitude and longitude coordinates measured by the rover for each waypoint,

	Waypoint Coordinates		Rover Coordinates		
Number	Latitude	Longitude	Latitude	Longitude	Offset [in]
2	40.082043	-88.201225	40.082043	-88.201227	6.699045
3	40.082024	-88.201225	40.082024	-88.201226	3.349523
4	40.082005	-88.201225	40.082005	-88.201226	3.349524
5	40.081985	-88.201225	40.081985	-88.201226	3.349525
6	40.08197	-88.201225	40.08197	-88.201224	3.349526
7	40.081951	-88.201225	40.081951	-88.201224	3.349527
16	40.081936	-88.201225	40.081936	-88.201226	3.349528
17	40.081951	-88.201225	40.081951	-88.201227	6.699054
18	40.082043	-88.201225	40.082043	-88.201227	6.699045

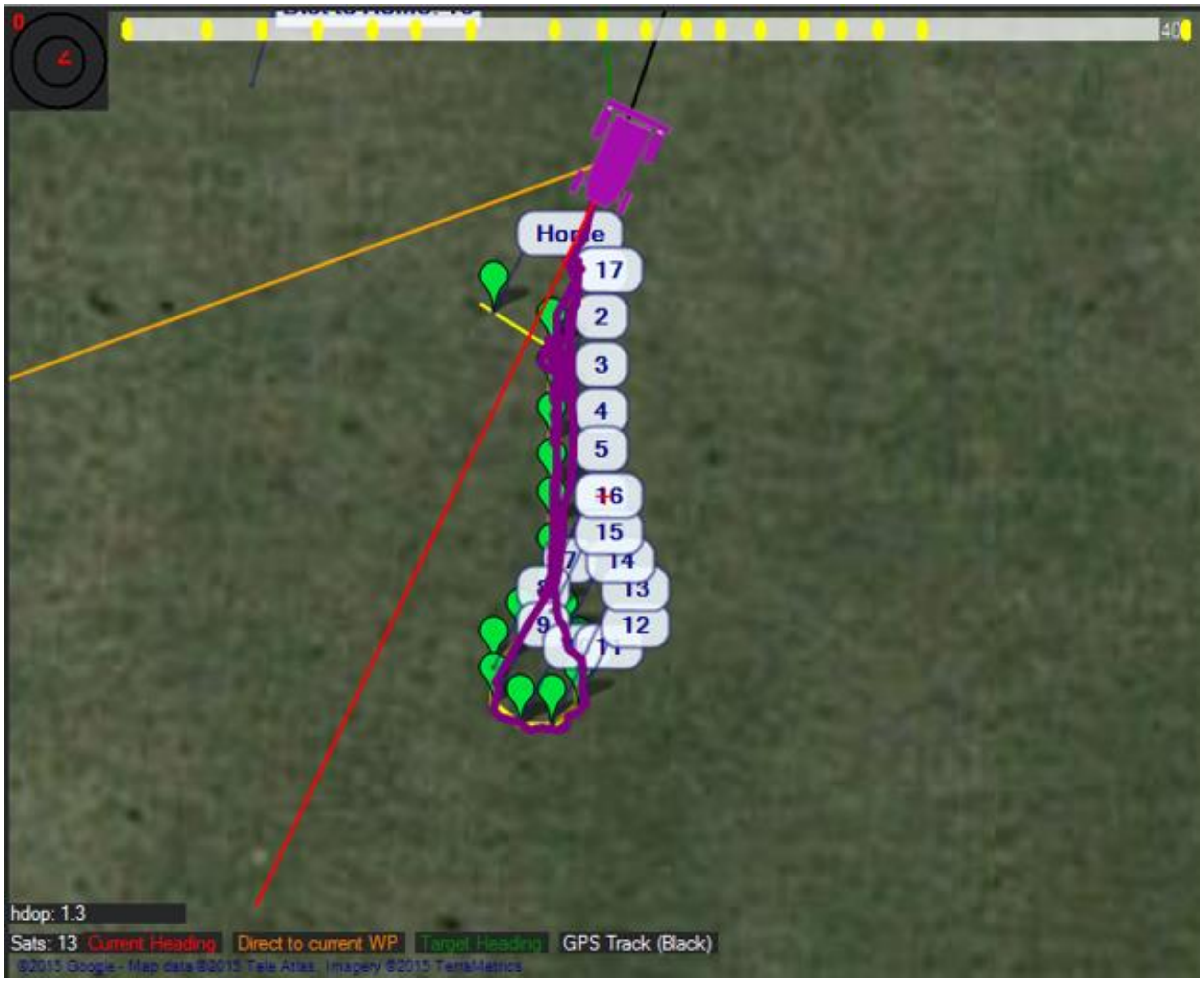


Figure 9 – Test 1 path as seen from the Mission Planner software. The rover is instructed to follow the path constructed by the seventeen numbered waypoints. After driving through waypoints two through seven, the rover makes a sweeping turn and drives back through the same path, as outlines by waypoints sixteen through eighteen.

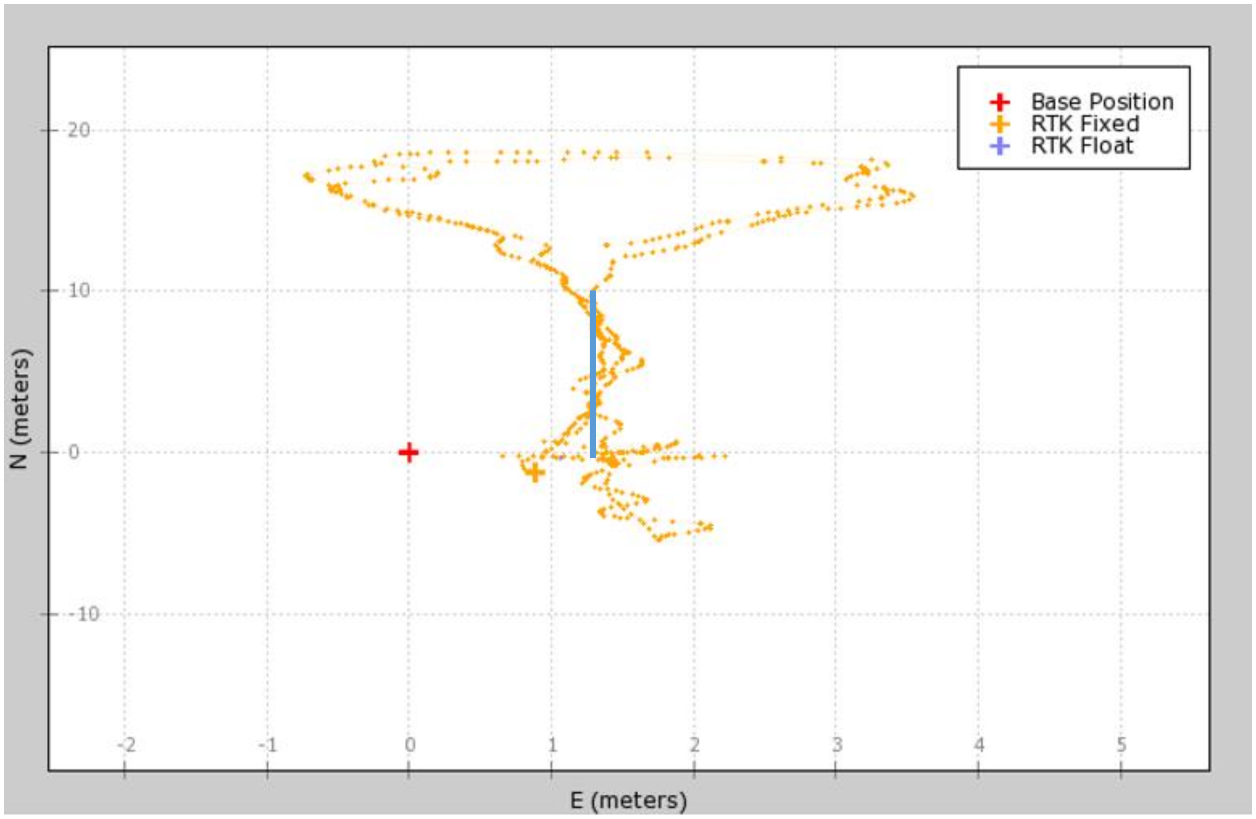


Figure 10 – Straight line test one plot. This has a single departure and return path, and was created using the Pixhawk Console software. The orange markers show the GPS latitude and longitude coordinates that were measured by the rover. The straight blue line depicts the straight line path, as the rover was instructed to navigate.

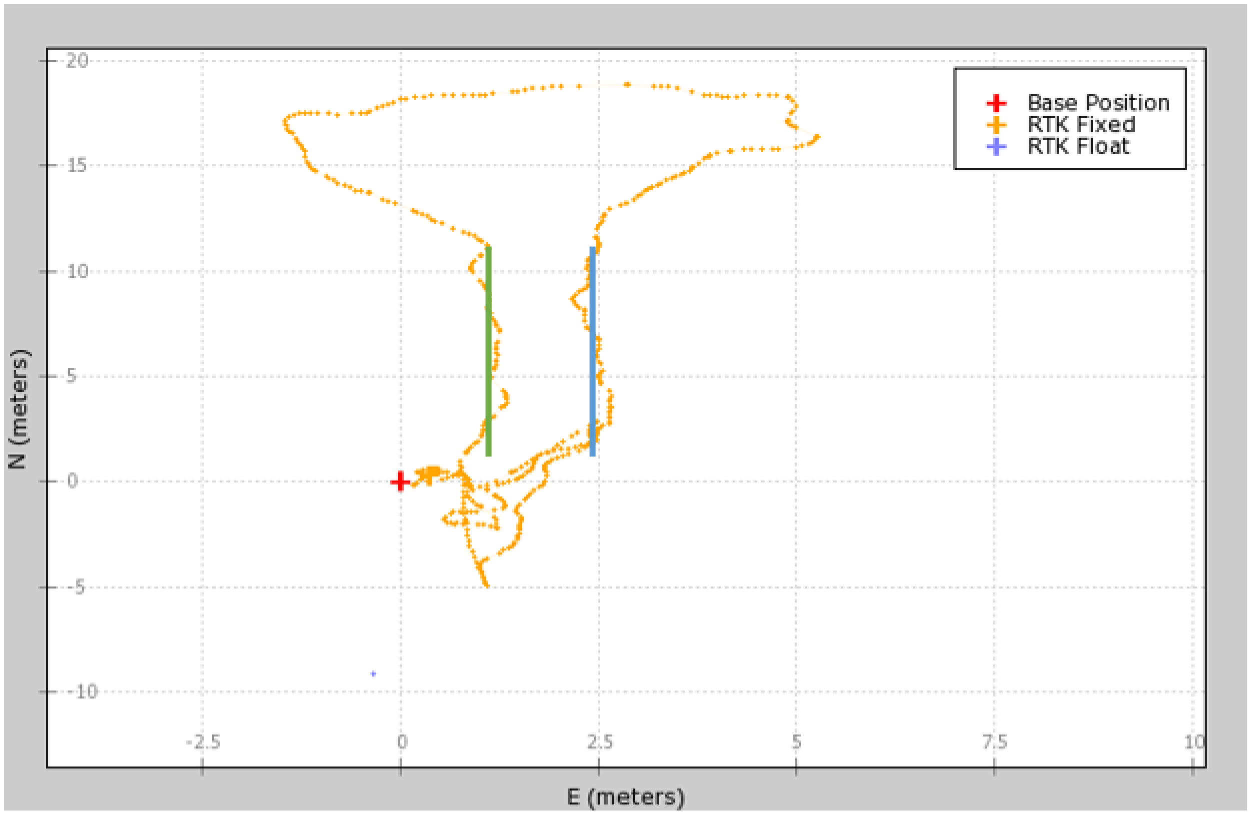


Figure 11 – Straight line test two plot. The ground rover is instructed to depart from the initial waypoint along the path shown in blue. After completing the turn shown at the top of the figure, the rover returns along the path shown in green. The orange points are GPS coordinates as measured by the Piksi RTK GPS system.

Figure 11 shows the navigation of the rover in test 2, which has a slightly altered path. Instead of one single departure and return path, the rover was instructed to follow two separate paths. The steering PID controller was tuned with a proportional gain of 1, an integral gain of 0.2, and a derivative gain of 0.05. The departure path is shown in Figure 11 as a blue line and the return path is a green line. The orange markers indicate the path taken by the rover. An average overall offset error of 3.56-inches was observed during test 2. Figures 10 and 11 were created using the Piksi Console software, provided with the Piksi RTK GPS developer kit.

5.2 RPLIDAR Module

As previously discussed in Chapter 4 of this thesis, testing of the RPLIDAR module was conducted in an indoors setting. A sample of the data collected is shown below in Figure 12. The points on the figure correspond to the x and y locations of the sensed objects in millimeters. Two rows of plungers were arranged, separated by 24-inches. The plungers of each row were separated by approximately 12-inches. The various limitations that were discovered throughout the testing process are discussed greater detail in the following sections.

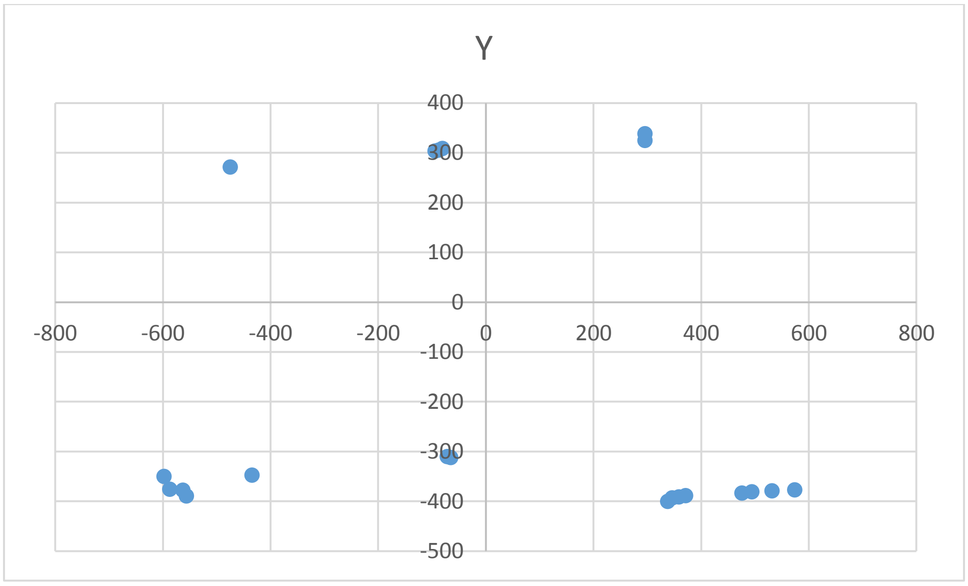


Figure 12 - LIDAR test scan data using plungers to simulate the stalks of a row crop environment. The blue points represent sensed objects, which are the plungers in this case. The LDIAR module is located at point (0,0). The units of the axis are millimeters.

5.2.1 Electromagnetic Interference

One major drawback that is associated with using the RPLIDAR module is the high level of electromagnetic interference (EMI) that is produced. Various operational characteristics of the RPLIDAR module cause the emission of electromagnetic noise which negatively impacts the reliability and quality of the IMU readings [15]. This interference deteriorates the ability for the navigation system to accurately operate. Miscalculations in the vehicle attitude solution causes the Pixhawk autopilot system to control the movement of the vehicle poorly. Increased levels of noise deteriorate the ability for the navigational system to achieve highly accurate movements, like those that are required for precise navigation of a row crop environment.

5.2.2 Dynamic Environment

The dynamic nature of the agricultural environment provides an additional limitation to a vision-based navigation system. As previously discussed in this thesis, various changes in the environment can create a significant amount of noise. As opposed to the LIDAR test scan data shown in the figure above, unpredictable environmental processes, such as wind, can cause displacement in the vegetation. Since the navigation system is unable to predict how the wind will change the environment at any given point in time, these changes are not accounted for by the autopilot. As a result, an increase in sensor noise is observed, which decreases the accuracy of the LIDAR readings.

5.2.3 Solar Interference

An additional limitation to the LIDAR module used in this thesis is the interference caused by solar radiation. Indoor testing conditions provide a stable environment for LIDAR data acquisition. Outdoor testing, however, is complicated by the continuous solar radiation. In certain cases, direct solar radiation into the visual sensing component of the LIDAR module provided false readings. This, in turn, was interpreted as a normal reading, and resulted in an increase in sensor noise. The most notable interference was observed when the sun’s position in the sky was close to the horizon, as more direct radiation was able to be exposed to the visual sensing component.

5.3 Calibration Issues

One major problem that is encountered when using a Pixhawk autopilot system with a large metal frame, as used in this project, is the interference of the on-board magnetometer. This is a significant issue because the magnetometer is a primary component of the navigation system, as all heading readings came primarily from this sensor. Due to the large heavy metal design of the chassis, calibration of the magnetometer proved to be difficult.

As shown in Figure 13, the calibration process is improved as the autopilot system, and all of its sensors, are moved farther away from the chassis. The images in Figure 13 show the Mission Planner's GUI calibration process screen. A proper magnetometer calibration would result with all 14 white dots, as seen in Figure 13.a, completely covered with sensor samples, as seen in Figure 13.e and 13.f. The top two images, a and b, show the final result of the calibration when the navigational sensors were mounted directly on top of the chassis. Images c and d in Figure 13 show the calibration result when the sensors were separated from the chassis by 2-inches. The last two images, e and f, show a normal calibration result, as experienced when the navigational sensors were raised to a height of 4-inches above the metal chassis. Therefore, it is shown that it is necessary to isolate the navigational sensors from any large metal components, such as the frame of the autonomous vehicle.

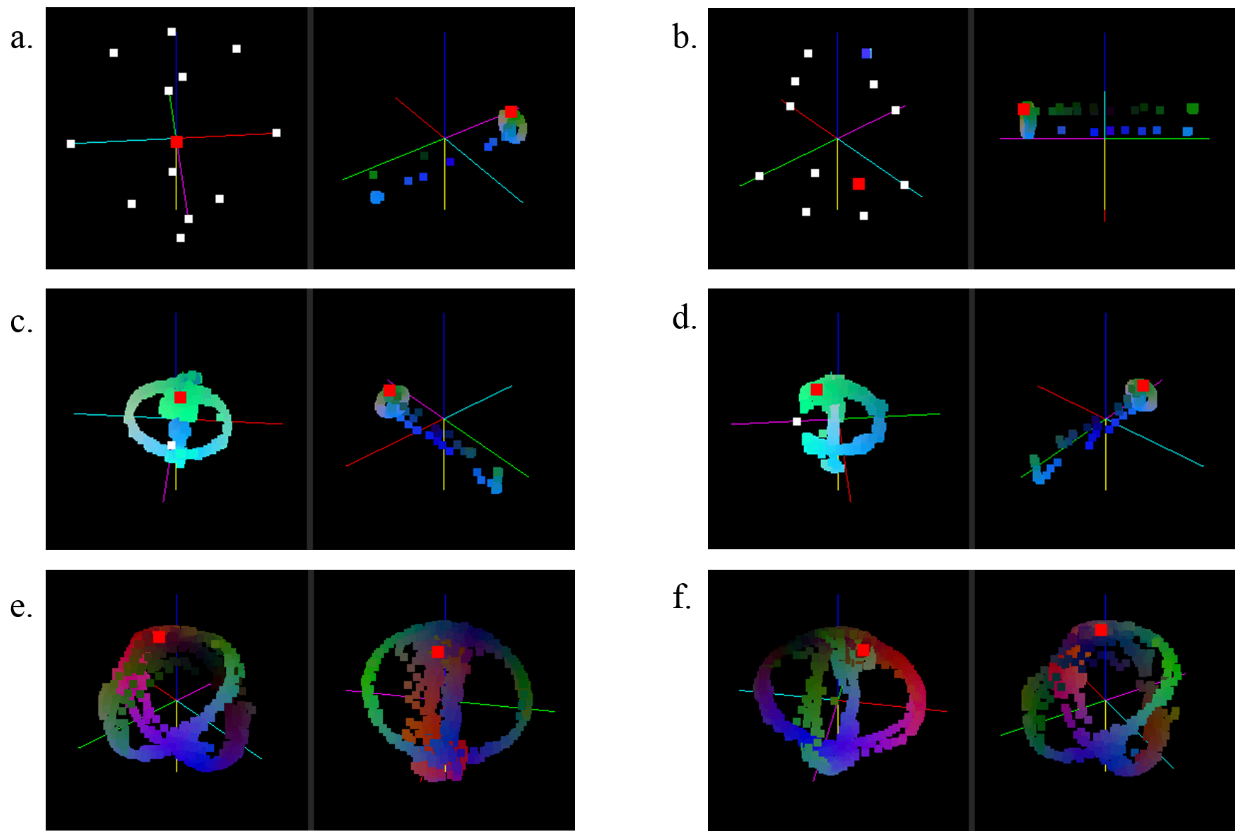


Figure 13 – Final compass calibration results from three IMU mounting techniques, involving no metallic components. The calibration result is shown for IMU placement (a, b) on the chassis, (c, d) 2-inches above the chassis, and (e, f) 4-inches above the chassis.

5.4 GPS Limitations

It is observed that heading readings are inconsistent at cruise speeds below 0.5 m/s. Investigation into the Pixhawk autopilot system reveals that, with default settings, ground rovers running ArduRover v2.50 automatically use GPS readings to determine the heading of the rover. By manually customizing certain parameters, it was possible to override this setting. As a result, the GPS readings from the Piksi RTK GPS were used instead. Despite marginal improvements at low speeds, certain scenarios suffering from inaccurate heading readings were persistent. Examples of these situations include navigation through environments that possesses visual obscurations of the sky, such as locations with significant tree populations. It is hypothesized that the trees created the GPS limitations by restricting the number of unobstructed satellites picked up by the GPS modules.

Additional issues are observed during testing when cloud cover is significantly thick. It is hypothesized that this is caused by a decrease in the satellite visibility, which is required for accurate GPS readings. As shown in Figure 14, GPS readings were intermittent during the testing period. The red line represents the HDOP value, which is an indicator of GPS fix quality. Smaller HDOP values correspond to more accurate GPS readings. This issue should be taken into account during all future navigational projects that require highly accurate GPS systems.

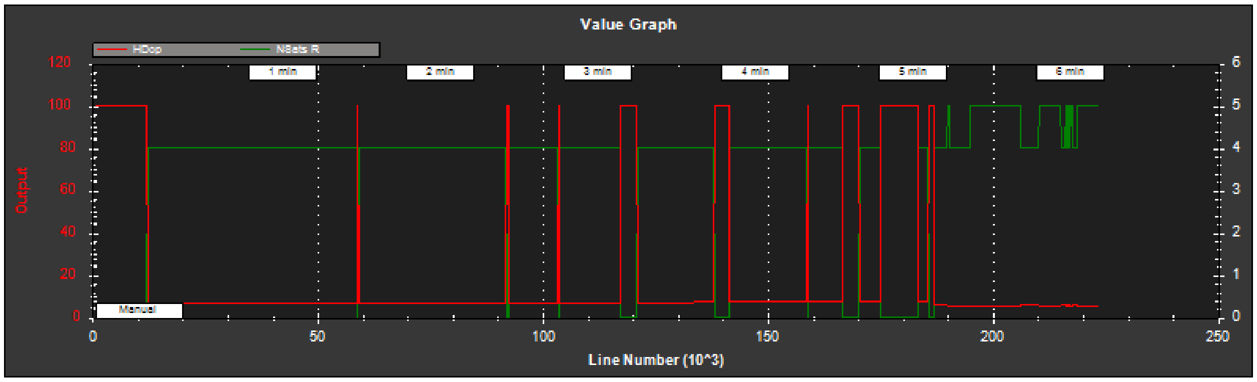


Figure 14 - GPS HDOP value graphed versus time. The red line represents the HDOP value. Higher HDOP values correspond to less accurate GPS measurements. This figure depicts the intermittent behavior observed during cloudy conditions.

CHAPTER 6
CONCLUSIONS, FUTURE WORK

6.1 Summary

Ground based navigation of agricultural environments has proven to be a significant challenge for the process of measuring the parameters that inevitably impact the growth stages of row crops through the use of on-board sensors. Large arrays of sensors that are required to obtain the same resolution of data gathered by an autonomous ground-based system are much too expensive and lack the reliability required for realistic applications.

This study focused on the navigational system of a ground-based robotic system that is designed to be used for growth stage research in agricultural settings using on-board sensors. This approach was based on proven autonomous navigation techniques and was inspired by the current lack of low-cost autonomous navigation systems for use with ground-based vehicles. Using high accuracy RTK GPS modules, the rover discussed in this research is capable of high accuracy localization. This was utilized to successfully enable autonomous navigation of a row crop environment.

It is found in this study that an average offset from the straight-line path was observed as 3.56-inches when using the RTK GPS system. By performing 10 straight-line tests, all with slightly different PID gain settings for the steering controller, navigational accuracy was measured using the offset from the GPS waypoints. The PID controller settings that produced an average offset of 3.56-inches consisted of a proportional gain of 1, an integral gain of 0.2, and a derivative gain of 0.05. This is important to all researchers using autonomous ground-based navigation systems because it produces a method for highly accurate navigation.

In addition, it is found that the most reliable results were obtained when the 2-D LIDAR system was removed from the navigation system. Although numerous drawbacks were observed with the use of a LIDAR module, further research into this component of the system should be conducted as beneficial results from the high accuracy of these systems has shown to be promising. A proven record of LIDAR-based navigation systems has been established, specifically in the application of indoor obstacle avoidance. By expanding this knowledge into an agricultural row crop environment, improved navigation capabilities of ground bases systems are likely to improve in the future.

It has been shown that a ground based navigation method for agricultural sensing of crop growth parameters is possible at a low cost. This was accomplished through the use of inexpensive sensing devices, such as the Piksi RTK GPS system, and the reduction in necessary expensive localization equipment that is most commonly seen in similar applications currently.

6.2 Sensing Alternatives

Autonomous navigation of ground-based vehicles often rely upon expensive sensing and localization equipment. This approach has been shown to be a viable solution to a wide array of applications, such as indoor navigation and robotic cluster algorithms [16]. As technological advances in sensing techniques and equipment continue to accelerate, it should be noted that new and affordable technologies may become available in the near future. Furthermore, these new technological advances may provide means for an autonomous navigation system that is capable of achieving the requirements put forth by this thesis in an affordable manner. One key component that should not be overlooked is the various LIDAR modules that are becoming available in the market. Reductions in prices of components will likely lead to a decrease in the overall cost for such modules.

6.3 Future Work

Further investigation is required to adequately analyze the effectiveness of a PID control method without the use of GPS coordinates. Utilizing a primary LIDAR module for robotic mapping, effective solutions for ground based navigation systems of autonomous robots may be available. Research into the various complications of using this type of system should be taken into consideration, as many limitations were discovered in this project. A high precision RTK GPS module has shown to be a very promising addition to robotic navigation systems. Further research into this topic should be conducted in order to adequately identify the most appropriate means of applying this technology to an agricultural based navigation system.

REFERENCES

[1] P. B. Sujit, Saripalli, S., Sousa, J., "Unmanned Aerial Vehicle Path Following," *IEEE Control Systems Magazine*, February 2014.

[2] S. Park, Deyst, J., How, J.P., "A New Nonlinear Guidance Logic for Trajectory Tracking," *American Institute of Aeronautics and Astronautics*, 2003.

[3] R. Curry, Lizarraga, M., Mairs, B., Elkraim, G. H., "L2+, an improved Line of Sight Guidance Law for UAVs," *American Control Conference*, Washington DC, 17-19 June 2013.

[4] J. R. Azinheira, Paiva, E., Ramos, J. G., Bueno, S. S., "Mission Path Following for an Autonomous Unmanned Airship," *IEEE International Conference on Robotics and Automation*, San Francisco, April 24-28 2000.

[5] J. Borenstein, B. Everett, and L. Feng. “Navigating Mobile Robots: Systems and Techniques.” A. K. Peters, Ltd., Wellesley, MA, 1996

[6] Thrun, Sebastian. "Robotic mapping: A survey. Exploring artificial intelligence in the new millennium” 1-35. 2002

[7] Izadi, Shahram, et al. "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera." *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011.

[8] D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors. “AI-based Mobile Robots: Case studies of successful robot systems”, Cambridge, MA, 1998.

[9] Driscoll, T. “Complete coverage path planning in an agricultural environment.” Iowa State University. 2011

[10] Yang, S. X. and C. Luo. “A neural network approach to complete coverage path planning.” *IEEE Transactions on Systems, Man, and Cybernetics* – Part B: Cybernetics. 34(1): 718-725, 2004.

[11] Sörensen, M. J. “Artificial Potential Field Approach to Path Tracking for a Non-Holonomic Mobile Robot.” *11th Mediterranean Conference on Control and Automation*, June 2003

[12] Sørensen, C. G; Bak, Thomas; Jørgensen, RN. “Mission planner for agricultural robotics” *Proc. AgEng*. pp894-895. 2004.

[13] Takasu, Tomoji, and Akio Yasuda. "Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB." *International symposium on GPS/GNSS*. Jeju, Korea: International Convention Centre, 2009.

[14] Reid, John F., et al. "Agricultural automatic guidance research in North America." *Computers and electronics in agriculture* 25.1: 155-167. 2000.

[15] Tristancho, Joshua, Enric Pastor, and Marcos Quilez. “An Electromagnetic Interference Reduction Check List for Unmanned Aircraft System.” *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*.

[16] Hiremath, Santosh. “Laser range finder model for autonomous navigation of a robot in a maize field using a particle filter.” *Computers and Electronics in Agriculture* 100:41–50. 2014.

[17] Salih, Atheer L., et al. "Flight PID controller design for a UAV quadrotor." *Scientific Research and Essays* 5.23: 3660-3667. 2010.

[18] “RPLIDAR Devkit Manual.” http://www.RoboPeak.com/. Accessed: 2015-10-08.

[19] “Swiftnav Piksi Datasheet v2.3.1”. http://docs.swiftnav.com/pdfs/piksi_datasheet_v2.3.1.pdf. Accessed: 2015-10-14.

APPENDIX A CODE

A.1 Background

The purpose of this document is to provide detailed information on the programming elements utilized in this thesis. The programming language used for the PID controller provided in this document is Python.

A.2 PID Controller

```
import time
import controller
from droneapi.lib import VehicleMode, Location
from pymavlink import mavutil
from time import sleep
import math
import serial
import numpy as np
import sys, traceback
from LidarPrint import Lidar
from wrapPi import wrap_PI

# This code is based off of the Red Balloon Finder Project used in Sparkfun’s 2014 AVC
competition. The original code can be found at: https://github.com/rmackay9/ardupilot-balloon-
finder

# Make sure mavproxy is running (mavproxy.py --master={USBPATH} --baudrate=57800
# Next, load the api module (module laod api)
# Last, start the api (api start {DIRECTORY}/navigate.py


class Navigate(object):
    def __init__(self):


        self.home_lock = False
        self.arming_check = False
        self.home_lock = False
        self.home_lock_last = time.time()
        self.lock = False
        self.api = local_connect()
        self.rover = self.api.get_vehicles()[0]
        self.angle_sum = 0
        self.leftTest = 195
        self.rightTest = 200
        self.testCount = 0
        self.initialRun = True
        self.initial_yaw = None
        self.target_yaw = None
        self.search_heading_change = None
        self.turn_left = False
        self.turn_right = False
        self.yaw_rotation_rate = 5
        self.target_velocity = None
        self.commands = None
        self.drive_forward = True
        self.left_distance = 301
        self.right_distance = 300
        self.rover_position = None
        self.PID = controller.Controller(1, 0, 0, math.radians(10.0))



    def mode_check(self):
        if self.rover.mode.name == "GUIDED":
            if not self.lock:
                self.lock = True
            return

        if self.rover.mode.name == "MANUAL":
            self.lock = False
            return

        self.lock = False

    # Hopefully, this will release control back to the receiver
    def RC_reset(self):
            # Cancel override by setting channels to 0
            self.rover.channel_override = { "1" : 0, "2" : 0, "3" : 0, "4" : 0, "5" : 0, "6" : 0, "7" : 0, "8"
: 0 }
            self.rover.flush()

    def navigate_row(self):
```

```
if not self.lock:
    print "navigate row error - no lock"
    return

port = '/dev/ttyUSB0'
ser = serial.Serial(port, 115200, timeout = 5)
#ser.setDTR(False)
#print ser.name
lidar = Lidar(ser)
now = time.time()

#look for new points flag
# then load in new distances?
#ex. self.difference, heading, etc = lidar.get_points()
# Maybe updating the controller too often?

current_yaw = self.rover.attitude.yaw
new_yaw = current_yaw

# Determine LIDAR offset & target yaw
# increment target yaw if left > right
# decrement target yaw if right < left
offset = lidar.getPoints(port,True)
#scale offset to yaw correction
if offset < 0:
    #veer right
    if offset < -1000:
        #add 5 degrees
        new_yaw = wrap_PI(current_yaw + 5)
    else:
        new_yaw = wrap_PI(current_yaw + 1)
if offset > 0:
    if offset > 1000:
        new_yaw = wrap_PI(current_yaw - 5)
    else:
        new_yaw = wrap_PI(current_yaw - 1)

if offset == 0:
    print "no rows detected - driving forward"
    new_yaw = current_yaw


error_yaw = wrap_PI(current_yaw - new_yaw)
dt = self.PID.get_dt(2.0)

yaw_correction = self.PID.get_pid(error_yaw, dt)
```

```
    yaw_final = wrap_PI(current_yaw + yaw_correction)
    # send yaw heading
    print "yaw: %s" % yaw_final
    self.condition_yaw(math.degrees(yaw_final))

# just testing here
def steer(self):
    if not self.lock:
        return



    self.initial_yaw = self.rover.attitude.yaw
    self.target_yaw = self.initial_yaw
    self.angle_sum = 0

    if self.testCount < 10:
        self.left_distance = self.leftTest + 1
        self.testCount = self.testCount + 1

    if self.testCount > 10 and self.testCount < 20:
        self.right_distance = self.rightTest + 1
        self.testCount = self.testCount +1

    if self.testCount > 19 and self.testCount < 25:
        self.right_distance = 200
        self.left_distance = 200
        self.testCount = self.testCount + 1

    if self.testCount > 24:
        self.right_distance = 195
        self.left_distance = 195


    if self.left_distance < 200:
            self.turn_right = True
            self.turn_left = False
    if self.right_distance < 200:
            self.turn_left = True
            self.turn_right = False
    if ((self.right_distance > 200) and (self.left_distance>200)):
            self.turn_left = False
            self.turn_right = False
    if ((self.left_distance < 200) and (self.right_distance<200)):
            self.turn_right = False
            self.turn_left = False
            self.drive_forward = False
            self.RC_reset()



    def armRover(self):
            if self.arming_check:
                    return True
            self.rover.armed = True
            self.rover.flush()
            while not self.rover.armed and not self.api.exit:
                time.sleep(1)
                print "please arm"
            self.arming_check = True
            return True


    def setGuided(self):
            # Set vehicle mode and armed attributes (the only settable attributes)
            print "Set Vehicle.mode=GUIDED (currently: %s)" % self.rover.mode.name
            self.rover.mode = VehicleMode("GUIDED")
            self.rover.flush()  # Flush to guarantee that previous writes to the vehicle have taken
place
            while not self.rover.mode.name=='GUIDED' and not api.exit:  #Wait until mode has
changed
                print " Waiting for mode change ..."
                time.sleep(1)



    def check_home(self):

        if self.initialRun:
            self.initial_yaw = self.rover.attitude.yaw
            self.target_yaw = self.initial_yaw
            self.angle_sum = 0
            port = '/dev/ttyUSB0'
            ser = serial.Serial(port, 115200, timeout = 5)
            ser.setDTR(False)
            print ser.name
            lidar = Lidar(ser)
            self.home_lock_last = time.time()
            self. initialRun = False
        if (time.time() - self.home_lock_last > 2):

            self.home_lock_last = time.time()

            if self.rover is None:
                self.rover = self.api.get_vehicles()[0]
                return

            # ensure the vehicle's position is known
            if self.rover.location is None:
                return False
            if self.rover.location.lat is None or self.rover.location.lon is None or
self.rover.location.alt is None:
                return False

            # #  hard coded test
            # home_lat = self.vehicle.location.lat
            # home_lon = self.vehicle.location.lon
            # self.home_initialised = True
            # return self.home_initialised

            # download the vehicle waypoints if we don't have them already
            if self.commands is None:
                self.fetch_mission()
                return False

            # get the home lat and lon
            home_lat = self.commands[0].x
            home_lon = self.commands[0].y

            # sanity check the home position
            if home_lat is None or home_lon is None:
                return False


        return self.home_lock


    def fetch_mission(self):
            print "\nGet home location"
            self.commands = self.rover.commands
            self.commands.download()
            self.commands.wait_valid()
            print " Home WP: %s" % self.commands[0]

    def RC_forward(self):
            print "\nOverriding RC channel 1 & 3 - should go forward"
            self.rover.channel_override = {"1" : 1500, "2" : 1499, "3" : 1624, "4" : 1510, "5" : 1900,
"6" : 1098, "7" : 898, "8" : 0  }
            self.rover.flush()
            print " Current overrides are:", self.rover.channel_override
            print " Channel default values:", self.rover.channel_readback  # All channel values before
override

    def RC_left(self):
            print "\nOverriding RC channel 1 - should go left"
            self.rover.channel_override = {"1" : 1373}
            self.rover.flush()

    def RC_right(self):
            print "\nOverriding RC channel 1 - should go right"
            self.rover.channel_override = {"1" : 1584}
            self.rover.flush()



    def resetVariables(self):
            ## Reset variables to sensible values.
            print "\nReset vehicle atributes/parameters and exit"
            self.rover.mode = VehicleMode("MANUAL")
            self.rover.armed = False
            self.arming_check = False
            #self.vehicle.parameters['THR_MIN']=130
            self.rover.flush()
            self.RC_reset()



    def starting_position(self):

        self.initial_yaw = self.rover.attitude.yaw
        self.target_yaw = self.initial_yaw
        #return self.rover_position
        self.initial_yaw = self.rover.attitude.yaw
        self.target_yaw = self.initial_yaw
        self.angle_sum = 0
        port = '/dev/ttyUSB0'
        ser = serial.Serial(port, 115200, timeout = 5)
        ser.setDTR(False)
        print ser.name
        lidar = Lidar(ser)
        self.home_lock_last = time.time()
        self. initialRun = False



    # condition_yaw - send condition_yaw mavlink command to vehicle so it points at specified
heading (in degrees)
    def condition_yaw(self, heading):
        # create the CONDITION_YAW command
        msg = self.rover.message_factory.mission_item_encode(0, 0,  # target system, target
component
                                0,     # sequence
                                mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
# frame
                                mavutil.mavlink.MAV_CMD_CONDITION_YAW,         #
command
                                2, # current - set to 2 to make it a guided command
                                0, # auto continue
                                heading, 0, 0, 0, 0, 0, 0) # param 1 ~ 7
        # send command to vehicle
        self.rover.send_mavlink(msg)
        self.rover.flush()

    # send_nav_velocity - send nav_velocity command to vehicle to request it fly in specified
direction
    def send_nav_velocity(self, velocity_x, velocity_y, velocity_z):
        # create the SET_POSITION_TARGET_LOCAL_NED command
        msg = self.rover.message_factory.set_position_target_local_ned_encode(
                                0,       # time_boot_ms (not used)
                                0, 0,    # target system, target component
                                mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
                                0,       # type_mask (not used)
                                0, 0, 0, # x, y, z positions (not used)
                                velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
                                0, 0, 0, # x, y, z acceleration (not used)
                                0, 0)    # yaw, yaw_rate (not used)
        # send command to vehicle
        self.rover.send_mavlink(msg)
        self.rover.flush()

    # advance_current_cmd - ask vehicle to advance to next command (i.e. abort current
command)
```

```
    def spin(self):
        if not self.lock:
            print "spin error - not in control of vehicle"
            return
        tempyaw = math.fabs(wrap_PI(self.rover.attitude.yaw - self.target_yaw))
        temptarget = math.radians(self.yaw_rotation_rate * 2.0)
        print "%s < %s" % (tempyaw, temptarget)

        if math.fabs(wrap_PI(self.rover.attitude.yaw - self.target_yaw)) <
math.radians(self.yaw_rotation_rate * 2.0):

            self.target_yaw = self.target_yaw - math.radians(self.yaw_rotation_rate)
            self.angle_sum = self.angle_sum + math.radians(self.yaw_rotation_rate)
            # send yaw heading
            print "yaw: %s" % self.target_yaw
            self.condition_yaw(math.degrees(self.target_yaw))


    def run(self):
            while not self.api.exit:
                    if self.armRover():
                        if self.check_home():
                            #print "made it to check home"
                            if self.initialRun:
                                self.starting_position()

                            self.mode_check()
                            #self.spin()
                            self.navigate_row()
                            time.sleep(0.2)


    def strategy(self):
        if not self.lock:
            return
        #self.steer()
        if self.drive_forward:
                self.RC_forward()
        if self.turn_left:
                self.RC_left()
        if self.turn_right:
                self.RC_right()

        #time.sleep(1)
```

```
start = Navigate()
start.run()

import serial
import math
from time import sleep
import threading
import numpy as np

#  Based off of code written by John McCormack (2015)
#  www.jdmccormack.com

Start_Scan = "\xA5\x20" #Begins scanning
Force_Scan = "\xA5\x21" #Overrides anything preventing a scan
Health = "\xA5\x52" #Returns the state of the Lidar
Stop_Scan = "\xA5\x25" #Stops the scan
RESET = "\xA5\x40" #Resets the device


class Lidar():
    def __init__(self,port):
        #set the port as an instance variable

        self.port = port
        print "port %s" %self.port
        #lock checks if the connection is made
        lock = False
        self.totalPoints = 0
        debug = True

        #Begin by starting the scan
        lock = self.startScan(self.port)
        print lock
        #Once scan is started, beging printing data
        if lock == True and debug == True:
            print "getPoints"
            difference = self.getPoints(self.port)
            print "shouldn't print"
        else:
            print "Exiting"


    def startScan(self, port):
        print "Connecting"
        line = ""
        #Lock is true once connected
        lock = False
        #Continue looping until connected
        while lock == False:
            print "..."
            # First reset the port
            port.write(RESET)
            # Wait
            sleep(2)
            #Start reading
            #Look for the correct start
            #frame of A55A
            port.write(Start_Scan)
            try:
                #If after looping nothing found,
                #Reset and try again
                for a in range(0, 250):
                    character = port.read()
                    line += character
                    #print "%s" %line
                    if (line[0:2] == "\xa5\x5a"):
                        if(len(line) == 7):
                            lock = True
                            break

                    elif (line[0:2] != "\xa5\x5a" and len(line) == 2):
                        line = ""
            except KeyboardInterrupt:
                break
        return lock


    def getPoints(self,port,polar=True):
        line = ""
        arrayIndex = 0
        pointsAngle = 0
        pointsDistance = 0
        self.totalPoints = 0


        while self.totalPoints < 20:
            try:
                #print "reading points"
                character = port.read()
                line += character

                # my soul cries for a different approach than this
                rightMin = 5000
                leftMin = 5000

                #Data comes in 5 byte blocks
                if (len(line) == 5):

                    distance = self.polar_distance(line)
                    #angle = self.point_Polar.angle
                    #angle = self.polar_angle(line)
                    if distance is not 0:
                        if distance < 1700:
                        #pointsAngle[arrayIndex] = angle
                        #pointsDistancearrayIndex) = distance
                        #arrayIndex = arrayIndex + 1
                        #print "%s, %s" %distance %angle
                            angle = self.polar_angle(line)
                            distanceEst=distance*math.sin(angle)
                            # negative means left side
                            if distanceEst < 0:
                                if distanceEst < leftMin:
                                    leftMin = distanceEst
                            if distanceEst > 0:
                                if distanceEst < rightMin:
                                    rightMin = distanceEst
                            print distanceEst

                            #print angle
                            self.totalPoints = self.totalPoints + 1


                    #print point
                    line = ""
                    #print self.totalPoints

            except KeyboardInterrupt:
                break
        difference = math.fabs(leftMin) - rightMin
        print "stop scan"
        port.write(Stop_Scan)
        print " left, right, diff"
        print leftMin
        print rightMin
        print difference
        return difference

    def stop_scan(self, port):
        print "stop scan"
        port.write(Stop_Scan)


    def leftshiftbits(self,line):
        line = int(line, 16)
        line = bin(line)
        line = line[:2] + "0" + line[2:-1]
        line = int(line, 2) #convert to integer
        return line

    def polar_distance(self,serial_frame):
        distance = serial_frame[4].encode("hex") + serial_frame[3].encode("hex")
        distance = int(distance, 16)
        distance = distance / 4 #instructions from data sheet
        return distance

    def polar_angle(self,serial_frame):
        angle = serial_frame[2].encode("hex") + serial_frame[1].encode("hex")
        angle = self.leftshiftbits(angle) #remove check bit, convert to integer
        angle = angle/64 #instruction from data sheet
        return angle

    def wrap_PI(angle):
        if (angle > math.pi):
            return (angle - (math.pi * 2.0))
        if (angle < -math.pi):
            return (angle + (math.pi * 2.0))
        return angle


if __name__ == "__main__":
    port = '/dev/ttyUSB0'
    ser = serial.Serial(port, 115200, timeout = 5)
    ser.setDTR(False)
    print ser.name
    lidar = Lidar(ser)

import math
import time

class Controller(object):

    def __init__(self, initial_p=0, initial_i=0, initial_d=0, initial_imax=0):
        # default config file
        self.p_gain = initial_p
        self.i_gain = initial_i
        self.d_gain = initial_d
        self.imax = abs(initial_imax)
        self.integrator = 0
        self.last_error = None
        self.last_update = time.time()

    # __str__ - print position vector as string
    def __str__(self):
        return "P:%s,I:%s,D:%s,IMAX:%s,Integrator:%s" % (self.p_gain, self.i_gain, self.d_gain,
self.imax, self.integrator)

    # get_dt - returns time difference since last get_dt call
    def get_dt(self, max_dt):
        now = time.time()
        time_diff = now - self.last_update
        self.last_update = now
        if time_diff > max_dt:
            return 0.0
        else:
            return time_diff

    # get_p - return p term
    def get_p(self, error):
        return self.p_gain * error

    # get_i - return i term
    def get_i(self, error, dt):
        self.integrator = self.integrator + error * self.i_gain * dt
        self.integrator = min(self.integrator, self.imax)
        self.integrator = max(self.integrator, -self.imax)
        return self.integrator

    # get_d - return d term
    def get_d(self, error, dt):
        if self.last_error is None:
            self.last_error = error
        ret = (error - self.last_error) * self.d_gain * dt
        self.last_error = error
        return ret

    # get pi - return p and i terms
    def get_pi(self, error, dt):
        return self.get_p(error) + self.get_i(error,dt)

    # get pid - return p, i and d terms
    def get_pid(self, error, dt):
        return self.get_p(error) + self.get_i(error,dt) + self.get_d(error, dt)

    # get_integrator - return built up i term
    def get_integrator(self):
        return self.integrator

    # reset_I - clears I term
    def reset_I(self):
        self.integrator = 0

    # main - used to test the class
    def main(self):

        # print object
        print "Test PID: %s" % test_pid

        # run it through a test
        for i in range (0, 100):
            result_p = test_pid.get_p(i)
            result_i = test_pid.get_i(i, 0.1)
            result_d = test_pid.get_d(i, 0.1)
            result = result_p + result_i + result_d
            print "Err %s, Result: %f (P:%f, I:%f, D:%f, Int:%f)" % (i, result, result_p, result_i,
result_d, self.get_integrator())

# run the main routine if this is file is called from the command line
if __name__ == "__main__":
    # create pid object P, I, D, IMAX
    test_pid = Controller(1.0, 0.5, 0.01, 50)
    test_pid.main()
```

A.2 GPS Conversion Code

```
Public Function getDistance(latitude1, longitude1, latitude2, longitude2)
earth_radius = 6371
Pi = 3.14159265
deg2rad = Pi / 180

dLat = deg2rad * (latitude2 - latitude1)
dLon = deg2rad * (longitude2 - longitude1)

a = Sin(dLat / 2) * Sin(dLat / 2) + Cos(deg2rad * latitude1) * Cos(deg2rad * latitude2) *
Sin(dLon / 2) * Sin(dLon / 2)
```

```
c = 2 * WorksheetFunction.Asin(Sqr(a))

d = earth_radius * c

getDistance = d

End Function
```