

April 2017

# Autonomous Snowblower

Daniel Joseph Pongratz  
*Worcester Polytechnic Institute*

Harrison Louis Vaporciyan  
*Worcester Polytechnic Institute*

Nicholas Tyler Rowles  
*Worcester Polytechnic Institute*

Follow this and additional works at: <https://digitalcommons.wpi.edu/mqp-all>

---

## Repository Citation

Pongratz, D. J., Vaporciyan, H. L., & Rowles, N. T. (2017). *Autonomous Snowblower*. Retrieved from <https://digitalcommons.wpi.edu/mqp-all/2885>

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact [digitalwpi@wpi.edu](mailto:digitalwpi@wpi.edu).



# WPI

Worcester Polytechnic Institute

A Major Qualifying Project

Autonomous Snowblower - “Snoomba”

SUBMITTED BY:

**Daniel Pongratz**, Robotics Engineering

**Nicholas Rowles**, Electrical & Computer Engineering

**Harrison Vaporciyan**, Robotics Engineering and Computer Science

ADVISED BY:

**William Michalson**, Professor

Electrical & Computer Engineering, Computer Science,

Robotics Engineering

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.

## Abstract

The purpose of this project is to research and develop a system to autonomously clear a specified area of snow. The resultant system includes a modified snowblower, a base station, and a computer application to monitor and define the area for the snowblower to clear. A high precision GPS is equipped on both the snowblower and the base station to provide accurate location data. The snowblower is additionally equipped with a LIDAR sensor for obstacle detection in the snow, as well as a microcontroller to run embedded software and interface with the computer application. The team faced many design challenges and learned a substantial amount through the implementation of the research they conducted.

## Acknowledgements

We would like to thank Washburn Shops, Prof. Tobjorn Bergstrom, Paul Shiffler, Karin Suttor, Katherine Crighton, and AXP. We would also like to extend a very special thank you to Prof. William Michalson for all his help and patience throughout this project and beyond.

# Table of Contents

<b>Abstract</b>	<b>1</b>
<b>Acknowledgements</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>Table of Figures</b>	<b>5</b>
<b>1 Introduction</b>	<b>6</b>
1.1 Problem Statement	6
1.2 Project Statement	6
<b>2 Background</b>	<b>7</b>
2.1 Problem Overview	7
2.2 Related Work	7
<b>3 Design Requirements and Proposed System Design</b>	<b>8</b>
3.1 Use Cases	8
3.2 Power and Energy/Mechanical Design	11
3.3 Electrical and Embedded Design	18
3.4 Localization and Mapping	21
3.5 Software Design	26
<b>4 Methodology/Implementation</b>	<b>29</b>
4.1 Hardware Configuration	30
4.2 Software Implementation	31
4.2.1 User Interface	33
4.2.2 High Precision GPS Sensor	35
4.2.3 Lidar Sensor	37
4.2.4 g_mapping and rViz Interface	37
4.2.5 Pathfinding	38
4.2.6 Motor Controller	38
4.2.7 Packaging	38
4.3 Mechanical Implementation	39
<b>5 Results</b>	<b>39</b>
5.1 GPS	40
5.2 LIDAR	42
<b>6 Future Work</b>	<b>43</b>

6.1 Further Mechanical Work	43
6.1.1 Aiming the Chute	43
6.1.2 Controlling the Auger	44
6.1.3 Designs for the Future	44
6.1.4 Battery Mount	44
6.1.5 Alternator	44
6.2 User Interface Improvement	45
6.3 Sensor Improvements	47
6.3.1 GPS Improvements	47
6.3.2 Localization (LIDAR) Improvements	47
6.4 Navigation Improvements	48
<b>7 Conclusion</b>	<b>49</b>

## Table of Figures

1. Torque vs RPM graph of an electric motor vs a gasoline engine	12
2. Mechanical calculations	15
3. Undercarriage of Unmodified snowblower	16
4. CAD Model of the undercarriage	17
5. CAD model of the tread tensioning system	17
6. Implementation of the tread tensioning system	18
7. H-Bridge Concept	19
8. Motor Controller Circuit	19
9. Motor controller circuit to drive the snowblower	20
10. Map generated by Blanche robot using a 360-degree optical rangefinder	22
11. IRCF360 IR Distance Sensor	23
12. RPLIDAR 360° Laser Scanner	24
13. LIDAR Lite V3 (left) and Lynxmotion Pan and Tilt Kit (right)	24
14. Rendering of the LIDAR housing	25
15. User interface concept	26
16. uBlox setup	30
17. Example of u-center GUI during operation	31
18. Software Interface	32
19. Example of the GUI during operation	34
20. <i>NMEA</i> messages from the uBlox roaming GPS module as they are processed by the <i>gps</i> node	36
21. Measured GPS position vs. actual GPS position	40
22. u-center configuration showing the raw lat. & lon. (left) vs. the corrected lat. & lon (right)	41
23. Raspberry Pi I2C connections - Lidar Lite connected on channel 62	42
24. Datastream from the <i>DistanceSensor</i> topic	43
25. Glitch in the boundary definition apparatus	46

# 1 Introduction

## 1.1 Problem Statement

Across the country, homeowners in northern climates face the daunting and downright dangerous task of clearing their driveways of snow multiple times each winter. Every year, there are over 100 snow-clearing-related deaths reported in the US alone, mostly from heart attacks. Snowblowers mitigate this risk somewhat, but they are far from perfect. The main way for homeowners to avoid the risk of heart attacks is to pay someone else to clear their driveway for them. However, this approach merely transfers the danger to someone else and possibly ties up expensive capital (i.e., snowplows), making it rather costly and impractical. Clearly, some kind of alternative is needed.

## 1.2 Project Statement

The purpose of this Major Qualifying Project (MQP) was to develop a system for autonomously and safely clearing snow. After some initial design sessions, the team determined that this system would have two physical parts: an autonomous snow blower, and a “base station” of sorts that would allow the user to control the robot remotely, along with a computer application to give the user control over and feedback from the system.

The robot developed for this project, named Autonomous Snowblower, was designed to fulfill a number of requirements needed to create a system that could autonomously and safely clear snow from an arbitrary user-designated area, such as:

- A drive train capable of carrying the robot over (and sometimes through) snow, ice, concrete, dirt, and other assorted surfaces
- An auger-and-impeller assembly to break up ice and snow and throw them a considerable distance away
- A GPS system accurate enough to prevent the robot from accidentally going outside of its boundaries
- A sensor capable of detecting both static and dynamic obstacles at long ranges, even through heavy snowfall
- A UI that allows the user to easily define and change boundaries, start or stop the robot, and view its progress



## 2 Background

### 2.1 Problem Overview

Every year, over 100 people die in the U.S. while shoveling snow, mostly from heart attacks<sup>1</sup>. Snowblowers are safer, but still carry a nontrivial amount of risk<sup>2</sup>. What makes clearing snow so hazardous? First, and perhaps most obviously, lifting and throwing hundreds of pounds of snow in a given session - or even just pushing a heavy snowblower - is hard work. In one study, researchers found that the average snow-shoveling session raised subjects' heart rates more than running on a treadmill. Add to that the fact that many of the people shoveling their own driveways are sedentary, overweight, and/or old (all risk factors for heart attacks), and suddenly this seemingly innocuous chore becomes downright dangerous. Furthermore, the most popular time for clearing snow is usually between 6AM and 10AM, when circadian fluctuations already make people more vulnerable to heart attacks. Cold conditions further exacerbate this risk by constricting arteries, leading to a "perfect storm" that contributes to over 100 deaths and countless emergency room visits every winter. Clearly, this is somewhat of an issue, especially for those in the aforementioned at-risk groups (the elderly, overweight people, smokers, etc).

The solution to this issue has traditionally been for these people to hire workers to clear their respective yards and driveways for them. However, this can get expensive rather quickly, as the cost for one worker to clear snow can run up to \$75 an hour<sup>3</sup>, and these services may not operate in remote locations.

### 2.2 Related Work

In the course of researching this project, the team came across several relevant bodies of work that had already covered some of what they hoped to go over. One of these was the Kobi robot, which was announced and entered a closed beta halfway through this MQP. It bills itself as a "fully autonomous all-season garden robot", capable of autonomously mowing a lawn, gathering fallen leaves, and clearing snow. However, the Kobi's approach differs significantly

---

<sup>1</sup> <http://www.bbc.com/news/blogs-magazine-monitor-30119410>

<sup>2</sup> <http://www.sciencedirect.com/science/article/pii/S000291490101520X>

<sup>3</sup> <https://www.angieslist.com/articles/whats-annual-snow-removal-contract-cost.htm>

from ours: rather than clearing snow all at once like a traditional snowblower, it follows the weather forecast and clears the yard every time a few inches of snow fall. By doing this several times, it is able to minimize power consumption and avoids the need for a large and dangerous auger at the front. Although the Kobi's approach was interesting, the team decided that it was not particularly relevant to the project, as they had already committed to using the modified snowblower as a platform.

Another relevant body of work came from a seemingly unlikely source: tractors. Several companies, such as Trimble, Raven, and Ag Leader, specialize in assisted steering systems for tractors. These systems consist of a differential GPS system and an on-board computer that either assists a human driver or replaces them entirely. The latter situation was of specific interest to the team since it involved autonomously steering a large vehicle in a predetermined pattern, which seemed applicable to their project. What's more, these systems are incredibly advanced, capable of providing minor corrections to account for elevation changes and following a specified path with centimeter-level accuracy. However, upon further investigation, the team discovered that these systems were all designed to work exclusively with tractors, highly proprietary, and well outside of their budget.

### 3 Design Requirements and Proposed System Design

The first step of the design process was to develop a strong set of design requirements and use cases for the finished product, then design a system around those specifications.

#### 3.1 Use Cases

A snowblower can be a dangerous thing, even when piloted by an experienced human operator. This danger becomes magnified when turning the snowblower over to a completely automated system. Therefore, the team's goal with this project is to make it so this dangerous object can be used with minimal human interaction while still being safe. This requires the team go through every step and scenario that could be brought up while using the snowblower system, then identify any potential hazards that may arrive from those steps. To accomplish this, the team created several use cases ranging from setting up the machine to obstacle detection.

In addition to creating the actual use cases, the team also thought about and brainstormed some solutions to their associated problems, crossing out the ones they thought were not viable in their project. In some cases, their solutions were concepts or equipment that

some or all of the team had no previous experience with. They assigned these solutions to various group members, who then researched their viability. The use cases are as follows:

### Environment

- Snow covered driveway
- Snow covered sidewalk
- Snow covered yard/lawn

### User

- Residential
- NOT intended for commercial use

### Setup

- User needs to map out boundaries/determining where to throw snow
  - **GUI**
    - Need geospatial data
    - GPS differentiator
  - ⊖ ~~Beacons~~ (ruled out as impractical)
    - Additional setup from the user
    - Considerable cost addition (minimum of 4 necessary for the robot)
    - Development cost/time of implementing 4 beacons with the rest of the system
  - ~~Have user walk it around~~ (ruled out as even more impractical than the beacons)
- If the team have a GUI
  - Map of area
  - Map of where snow is put
  - Show obstacles/where it thinks there are obstacles
  - Snowcam
    - Outside the scope of the project
  - Manual control
- Differential GPS
  - Replace odometer for position tracking in mapping application (gmapping)
- Fuel, charging

- Alternator instead of charging battery
  - Electric start
- Put it in starting position
  - Know when to start/stop
    - Button on GUI? On machine? On both?
    - Link to weather service? ← stretch goal
    - Home? (shed/base station)

### Operation

- Needs to clear snow
  - 2-stage drive system - Auger & Impeller
- Movement
  - Treads
    - Small turn radius (can turn on a dime)
- Pathfinding
  - Space filling algorithm
- Sensing
  - Obstacle detection
  - Snow depth
  - Stationary vs. Mobile
  - Slope of driveway
  - Detect if obstacle is in path of snow being thrown
- Internal sensing
  - Amount of fuel left
  - Battery left
  - Jammed/Status of snowblower
  - Engine temperature
  - Stuck/not stuck
    - Computed internally
  - Stress on motors
- Obstacles
  - Immobile obstacle → go around
  - Mobile obstacle → stop if gets too close

- “Aurora” around each mobile obstacle
  - Path around, if it is in aurora, set off warning, cut engine
    - Restart once mobile object is at a safe distance

## 3.2 Power and Energy/Mechanical Design

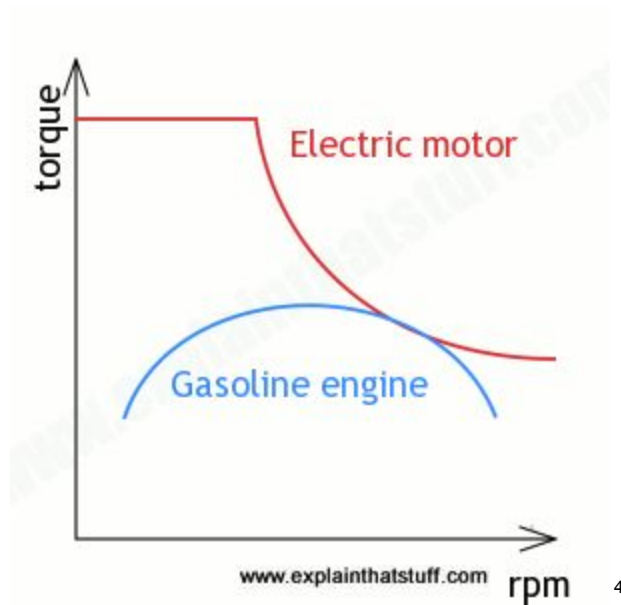
This project is not only a computer science problem, nor is it solely an electrical engineering problem - it is a robotics engineering problem, which implies the inclusion of a mechanical aspect in the system. The team acquired two of them over the summer for experimentation purposes. By examining these snowblowers, the team determined that these snowblowers (and by extension, any off-the-shelf snowblower) would require slight to severe modifications before they were suited to their purpose.

The next step was to determine exactly what these modifications were. A conventional snowblower usually runs on a single gas powered engine. This engine powers the auger, which grinds up the snow and pushes it to the impeller. The impeller, which is also driven by the engine, takes the snow and throws it through the chute. The engine additionally drives the wheels, which are connected by a single axle. Since this snowblower is designed to be steered by a human, the team’s next task was to figure out a way to retrofit an automated steering system onto it.

The first step in the design was to decide what sort of “wheels” the snowblower should have. Since most of the members of their team have driven through the snow, they knew from experience that plain rubber tires would not have enough traction. They came up with two potential solutions: chained tires, and treads. Chained tires are a hassle to put on, are very noisy, and prone to breaking. Treads, on the other hand, provide more traction due to their increased surface area and are better suited to varying types of terrain. For these reasons, they decided to go with treads.

The next issue to consider was the question of how to propel the snow itself. The Team had two options: a gas powered engine, or an electric motor. They began by researching what type of motors electric snowblowers use, on the assumption that those same motors would work for their application. This research would also tell the team how much power these motors typically draw, a nontrivial consideration on a system with multiple powered parts. Upon starting their research, however, the team quickly realized that most if not all of the electric snowblowers

they could find were corded. This was unfeasible for their needs for a variety of reasons: it would vastly complicate pathfinding, limit the distance the robot could travel from its base, require additional sensors to ensure it didn't drive over its own cable, and create a tripping hazard. A further consideration was the fact that the relationship between torque and RPM differs for gas-powered and electric engines, as shown below:



**Figure 1:** Torque vs RPM graph of an electric motor vs a gasoline engine

As the above graph (roughly) shows, the curve for a gas engine is roughly an inverted parabola, with a “sweet spot” that outputs the most power that steadily decreases as RPM increases or decreases. Conversely, electric motors have relatively constant torque at low RPMs but drop off abruptly past a certain point before settling at a lower asymptote. The team knew that driving the impeller and auger would require a good amount of torque and a high enough RPM to get the snow a decent distance away. These factors seemed to point to a gas-powered engine.

Once the team have figured out that they wanted a gas powered engine to drive the impeller and the author to actually throw the snow, they wanted a way to have the snowblower drive on its own. Again, they were faced with choosing either gas-powered motors or electric ones. They had already decided that the snowblower should not achieve speeds greater than 2 mph, for both effective snow clearing and general safety concerns. This meant that electric motors, which have high torque but low RPM, were perfectly suited to their needs. Furthermore,

<sup>4</sup> <http://www.explainthatstuff.com/electriccars.html>

electric motors are much more easily actuated by a microprocessor than gas-powered ones, which was a large consideration as the steering is entirely processor-controlled.

The first motors the team had mind were the CIM motors commonly used in FIRST robotics. These motors are familiar to the team and heavy duty. Most importantly, however, they're cheap: the team couldn't find motors with comparable torque outputs for less than double the price. The team also thought that the batteries used in FIRST robotics were also well-suited to their purpose, with a 17-amp-hour capacity and a good retention rate.

Using electric motors on the drivetrain raised several concerns, however. The largest of these was power consumption, as clearing a driveway and yard could potentially take several hours. This led the team to the idea of using an alternator. The alternator would have same function on their snowblower that an alternator in a car would have; namely, continuously charging the battery using some of the mechanical output from the gas engine. Because of this, the team needed to find an engine that could not only drive the impeller and auger but also the alternator. This, in turn, meant that the team needed to see how much power the battery would require. They began by trying to find an alternator that would meet their requirements for their battery. They wanted something that would produce around 17 amps; following that, they wanted to make sure that the alternator the team would use actually existed, and to see its data sheet. Unfortunately, they could not find an alternator that provided 17 amps. After bit of digging they did find an alternator that provided 35 amps at 12 volts. After taking a look at where they would buy the FRC batteries. When they realized that the battery came in a package of two they took advantage designing a system to wire the batteries in parallel so that the charge could be distributed at a longer period of time and use the alternator to charge the batteries. After some calculations, they realize that this alternator would only use a single-digit percentage of the power that the gas powered engines provided.

The engines they looked at were 11.5 and 14.5 horsepower:

$$11.5HP * 745.7 = 8.5755KW$$

$$14.5HP * 745.7 = 10.812KW$$

For the alternator, the team wanted to find out the required power:

$$17 Amps * 12 Volts = 204 Watts$$

They wanted to assume that the alternator was only 50% efficient (worst-case scenario), so the team doubled the number to get 408 Watts.

$$408W / 8.575 KW = .04$$

$$408W / 10.812KW = .03$$

Once the team had decided on all of the parts the team wanted to use, the team realized that they hadn't factored in the fact that if the drivetrain motors stalled, it would cause a voltage drop across the other on-board electronics, such as the sensors and the microcontroller. In practical terms, this meant that every time the robot drove into a solid obstacle, it would immediately and completely shut down. The team decided that they would work around this using either separate batteries for the sensitive electronics or by wiring them up with a voltage regulator, deciding on one or the other as they reached that point. They then did some calculations to make sure that all of the chosen parts were in fact completely viable for their purposes. The problem with this was that they did not have all of the variables needed to complete these equations. They found out that heavy wet snow is about 20% of the density of water<sup>5</sup> at its heaviest. The team assumed the snow to be 25% of the density of water for more than the worst-case scenario, ensuring steady operation in even heavy snowfall. In addition, they also wanted to know the weight of the snowblower. They visited HomeDepot.com and averaged six of the top rated snowblowers' weights, which came out to about 200lb. However, they knew this would be lower than their final product's weight, since the aftermarket modifications would add a significant amount of weight. To account for this, they doubled the estimated weight, bringing it up to approximately 400lb. They also needed to know the force that was required to drive the snowblower into snow. They decided that this force would be related to the weight of the actual snow. They then created a spreadsheet that allowed them to calculate the current needed by the CIM motors based off of a number of different factors, using functions to allow for instant feedback after editing any cell.

---

<sup>5</sup> <http://www.fsavalanche.org/density-snow-1/>



Variable	Value	Units	
Force needed to push snow	31.2	Lbs	Assuming
Speed	1.5	MPH	CIM motors
Wheel Diameter	8	Inches	2 Cubic feet of snow
Power to move Forward no snow	20	Watts	Force of snow is equal to the weight
voltage	12	Volts	
Effeciency	0.55		
Stall Current	133	Amps	
Free Current	2.7	Amps	
Stall Torque	343.4	Oz In	
<b>Calculated values</b>			
Force needed to push snow	138.784464	N	
Speed	0.67056	m/s	
Power to move snow	113.0633102	Watts	
Current	17.13080457	Amps	
Wheel Speed	63.02535746	RPM	
Torque	10.4	Ft Lbs	
Torque	0.124802688	HP	
Stall Torque	2.4249369	NM	
Torque (no gear ratio)	0.2685632425	Ft Lbs	
Gear ratio	38.72458458	I/O	

*Figure 2: Mechanical calculations<sup>6</sup>*

When it came to the actual decision of which gas powered engine to use for this snowblower, the team was at a loss. The budget for this project was \$750, the problem being that most gas-powered engines costs anywhere from \$200 to far beyond the constraints of their budget. Thanks to two generous donations, the team acquired the two aforementioned snowblowers.

During the second term of the MQP process, the mechanical section hit many snags. Although the team would have love to work on their snowblower that the team received in A term, they could not since they did not have a room for virtually all of the term.

One of the first things the team did when they returned from their A term break was examine the snowblower. There remained several questions from the last term about whether the equipment that the team had decided on would easily interact with the pre-existing snowblower parts. To do this, the team simply drained the snowblower of all of its fluids (the oil

<sup>6</sup>

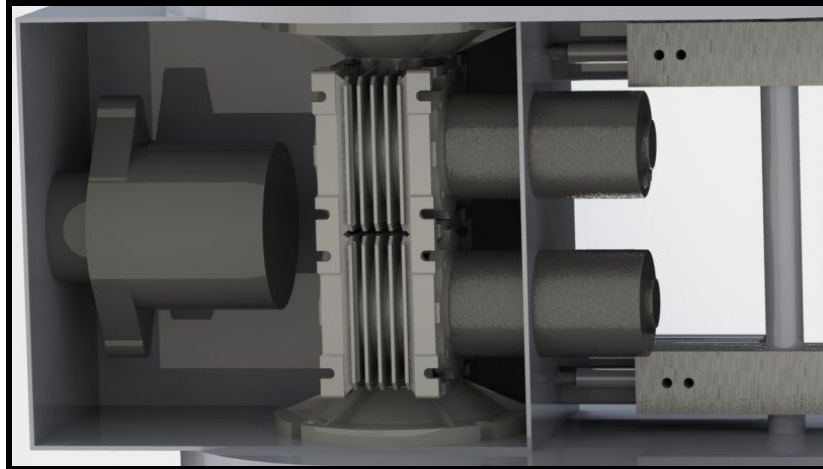
[https://docs.google.com/spreadsheets/d/1pR\\_RTsbTWFoel4QymraSxgSQsC4iqSCiiHwcnUxvTs/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1pR_RTsbTWFoel4QymraSxgSQsC4iqSCiiHwcnUxvTs/edit?usp=sharing)

and gasoline) to determine the quality of the engine. This was because, had the engine been relatively low-quality, tipping it over to check the underside could potentially flood it. Once they had drained everything they flipped the snowblower over to find that the snowblower was ideal for their situation. As the figure below shows, the snowblower appeared to have enough room to store the various parts researched the term before.



*Figure 3: Undercarriage of the unmodified snowblower*

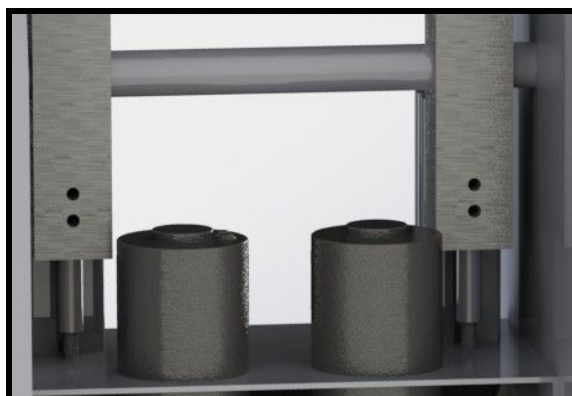
One thing that was immediately obvious was that the team's fears of the difficulty of getting the alternator to interface with the engine were unfounded. As shown above, the snowblower's original drive train was powered by a belt running down from the engine. Since this drive train was going to be scrapped and replaced with electronic motors and treads, this belt would then be free to power an alternator mounted in the same spot as the old transmission. Additionally, after doing some calculations using the table in Figure 2, it was determined that the CIM motors would need a 40:1 gear ratio, so the team purchased appropriate gearboxes. Unfortunately, due to the size of the alternator and its position in the undercarriage, the CIM motors were forced to stick out through holes in the chassis. The preliminary CAD model of the undercarriage based on this design is shown in Figures 4 and 5.



*Figure 4: CAD model of the undercarriage*

After inspecting the snowblower further, the team realized that the alternator could actually be mounted above the engine, outside the chassis. This would free up room in the undercarriage, allowing both CIM motors to rest safely inside.

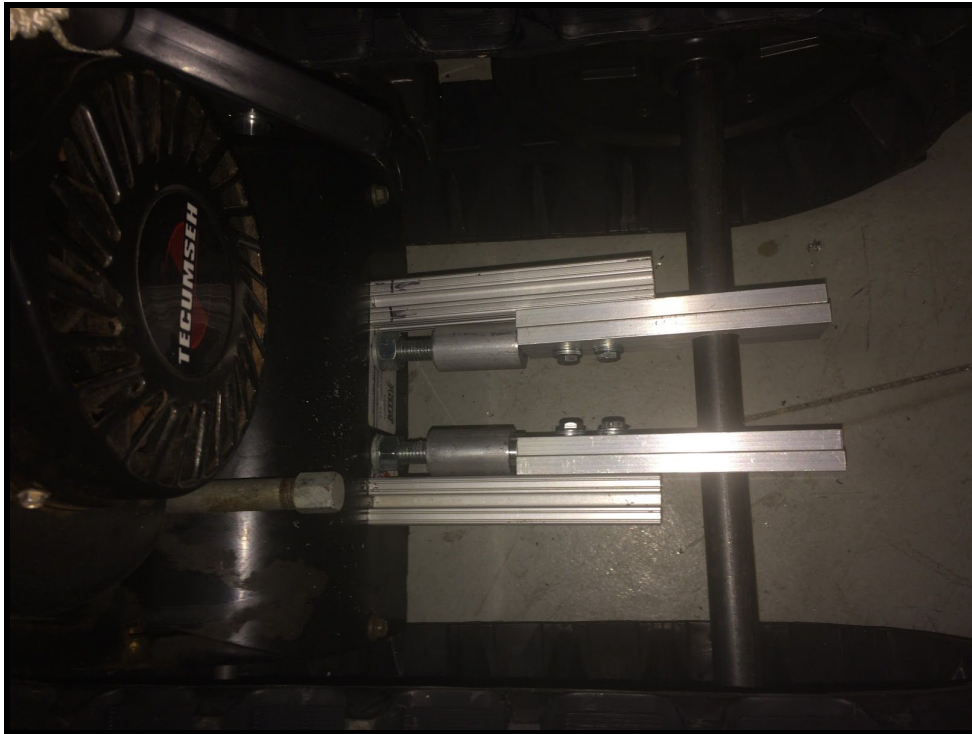
Once the team had the above fully described in a CAD file, they focused on the treads. Choosing the specific treads was difficult, as none of the listed measurements seemed to make any sense or correspond to any physical dimensions of the treads themselves. After some consideration, the team decided to use treads that their advisor had recommended them, since they worked on his snowblowers. Since the snowblower was designed to function at a specific height, the treads were chosen to be at the same height as the original wheels. This had the somewhat unfortunate side effect of causing the treads to extend past the back of the existing undercarriage chassis. The other challenge was to make sure that the treads were taut. This problem was solved with a tread tensioning system, shown in Figures 5 and 6 below.



*Figure 5: CAD model of the tread tensioning system*

As shown above, the treads in the back are connected by a single axle. The treads were seated with bearings, which allowed them to spin independently of one another.

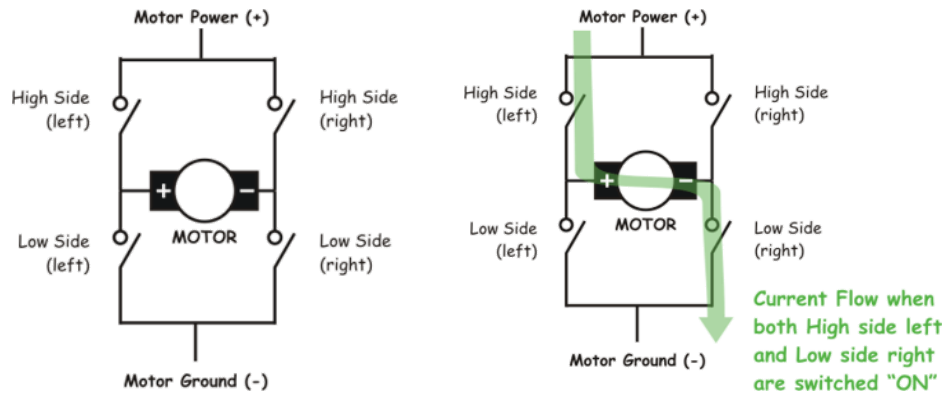
As shown in the picture below, a threaded rod with a tapped shaft is used to push up a block that is connected to the base using 80/20 with a built-in slider. The block is connected to the axle. Turning the shafts will cause the blocks to slide forwards or backwards, which will in turn cause the axle to move, which will in turn tighten the treads.



*Figure 6: Implementation of the tread tensioning system*

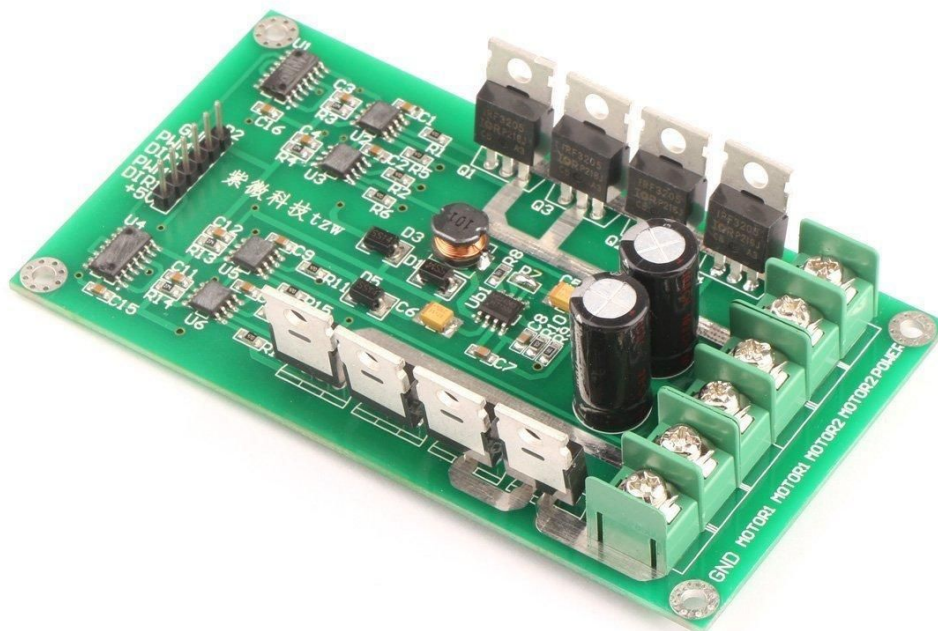
### 3.3 Electrical and Embedded Design

The system's main microcontroller is responsible for controlling the two motors responsible for moving the vehicle around. Most microcontrollers operate in the range of 3.3V-5V. However, the CIM motors used in this project operate at 12V, and draw 15A of current each. As a result, a motor controller must be used to supply this power to the motors, while being controlled by signals from a 3.3V-5V source. To do so, an H-Bridge circuit will be used. Shown below, an H-Bridge circuit can be used to control the direction of a motor by selecting the path of current through the motor. The motor controller used in the system is more sophisticated than this however, as it is able to change the speed at which the motor operates by controlling the voltage available to the motor.



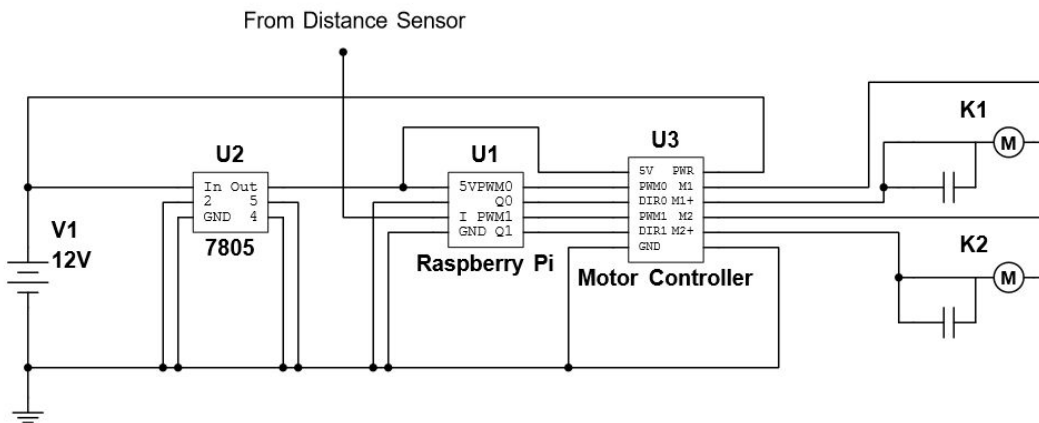
**Figure 7: H-bridge concept**

This motor controller is shown below. It can control two motors independently, and has 6 pins responsible for driving the motors; 5V supply, ground, two digital inputs (DIR1, DIR2), and two PWM (Pulse Width Modulation) inputs (PWM1, PWM2). DIR1 and DIR2 control the direction that the two motors spin, while the PWM inputs control the speed at which the motors spin; a PWM signal with a high duty cycle (majority of time spent in “high” or “1” state) will cause a motor to spin faster, while a lower duty cycle (more time spent in “low” or “0” state) will cause that same motor to spin slower. At the other side of the board are the connections for the motors. Each motor attaches (at both the positive and negative terminal) to the board. Additionally, the power supply for the motors (12V, in this case) and ground are hooked up here.



**Figure 8: Motor controller circuit**

The circuit below shown in figure 9 is the proposed design for a full motor controller. This circuit is able to control (via software running on the Raspberry Pi microcontroller) the motors *K1* and *K2*, which are responsible for snowblower movement. The 12V supply *V1* is used to model the power available from the alternator (or batteries). The alternator (and batteries) supply up to 35A of power, which is enough for this application. The motors draw up to 15A of current each, while the Raspberry Pi, 7805 linear voltage regulator, and Motor Controller collectively draw less than 5A.



**Figure 9:** Motor controller circuit to drive the snowblower

The Raspberry Pi is a relatively powerful general purpose microcontroller. The current model, the Raspberry Pi 3 model B, has enough features on board to meet the requirements of the project. Two PWM signals are required to drive the inputs to the motor controller. Note that these must be made in hardware and not in software, as the motor controller is sensitive to minor changes in the signal. Software PWMs cannot provide the accuracy needed for this application. The Raspberry Pi 3B meets this specification, as it can generate up to two hardware PWM signals. If additional PWM signals are needed, the Raspberry Pi would need to communicate with an external circuit, similar to the Adafruit PWM module, which can provide up to 16 additional hardware PWM signals.

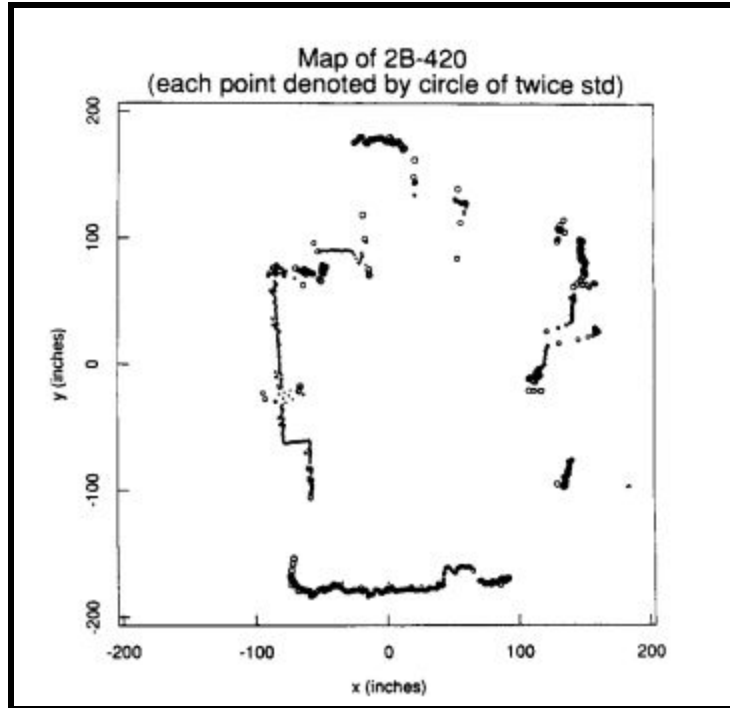
The Raspberry Pi is also capable of running the Linux operating system. This is necessary to the project, as the backbone of the software is ROS (Robot Operating System). A ROS system will contain various *nodes*, or processes that perform computations. Each node will typically serve only one function, but can communicate and exchange information with other nodes in the system using *messages*. Messages are sent over different channels called *topics*. Nodes can post to a topic, and/or subscribe to a topic to receive all messages posted by other

nodes. Additionally, ROS provides a platform for *services*. Similar to a web-server, a ROS service is comprised of pairs of a request and the subsequent response.

In order to run ROS on the Raspberry Pi, the microcontroller must be reconfigured to run a special version of Debian (a distribution of Linux). Once this is configured, ROS can be installed on the device. One advantage of using ROS in this setting is that information can be made available to view from a laptop or desktop. This makes testing the system significantly easier, as the Raspberry Pi will not need to be hooked up to a monitor.

### 3.4 Localization and Mapping

In order for the vehicle to properly clear an area of snow, it must know both its location, and the location of obstacles relative to it. To achieve this, a similar approach was taken to the *Blanche* robot. Blanche is an autonomous vehicle capable of mapping its environment and understanding its relative location. It uses only two sensors to achieve this; a 360-degree optical rangefinder, and an odometer. The odometer is used for “dead-reckoning”, that is, given an initial position and direction, the robot is able to determine its current location based on the movements that it has made from that initial position (recorded by the odometer). The rangefinder is used to determine the distance from the robot to an obstacle (wall, car, etc) in a particular direction. In order to get a full understanding of its surroundings, the rangefinder will rotate 360° and take 180 measurements per rotation, shown below.



**Figure 10:** Map generated by Blanche robot using a 360-degree optical rangefinder.

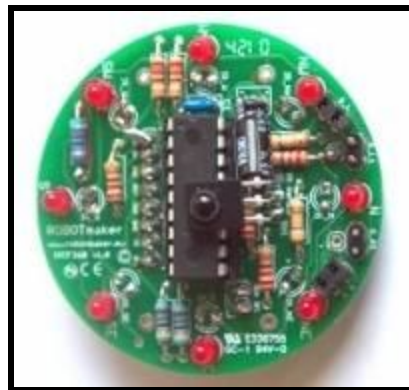
For the autonomous snowblower, a similar 360-degree rangefinder is used. However, the odometer used in Blanche is replaced by a high precision GPS unit. Dead-reckoning is not feasible in this situation, as the snowblower's treads will likely slip during operation, which would result in the odometer increasing (thinking the vehicle is moving forward) while in reality it remains still.

In order to map information from the rangefinder, the snowblower takes a similar approach to Blanche, in that it will use a 2D occupancy grid to represent the area that the snowblower will navigate over. An occupancy grid is effectively an array, where each position in the grid represents the likelihood that the real area represented by that square is occupied or not. The grid values are continuously updated as the rangefinder makes more readings, either from the same or a different position.

In order to choose the proper rangefinder to use, multiple sensing technologies were evaluated. Radar was the first technology researched. Although techniques exist for dealing with snowfall, RADAR is ultimately not suitable for this application, as there aren't any suitable 360-degree rangefinders available. RADAR is also typically used to detect motion, so if an object remains the same distance from the snowblower, a RADAR would not be able to sense that the object is there. When investigating ultrasonic, similar issues arose. Ultrasonic sensors



have a relatively limited range of (typically) only a few meters. This would not be suitable for their application, as the vehicle should be able to sense obstacles far enough in advance such that it has time to react to the information and path around it. Similarly, there does not exist (based upon current research) a 360-degree ultrasonic distance sensor. The next technology evaluated was Infrared. The closest item to a 360-degree IR sensor is the *IRCF360*, shown in figure 11 below. However, this sensor module is not able to provide the accuracy necessary for the system. This module can make at most 8 measurements in the horizontal plane. For reference, Blanche's rangefinder makes 180 measurements. Additionally, IR sensors are susceptible to interference from sunlight. This is a major red flag, as the system will be operating outdoors.



**Figure 11:** *IRCF360 IR Distance Sensor*

Given the research, the technology the system will use for a 360-degree rangefinder is LIDAR. Most sensors of this type are not only very accurate, but also have a range upwards of 25m, which should be more than enough for their application. The only major drawback to this approach is that readings may be distorted in snow, due to the laser reflecting off of falling snow. Testing is needed to determine if this will be an issue for their system. If it is, a backup sensor will be necessary in order to ensure that the vehicle continues to operate normally, even if the LIDAR rangefinder's readings are distorted due to snowfall.

As LIDAR is still an emerging technology, it is relatively expensive, more so for a 360-degree LIDAR sensor. The cheapest sensor of this type available is the *RPLIDAR 360° Laser Scanner* (shown below) coming in at a cost of ~\$400.



*Figure 12: RPLIDAR 360° Laser Scanner*

An alternative to this module is to create a similar mechanism out of a static LIDAR module and a servo motor. With this in mind, LIDAR Lite V3 was chosen as the LIDAR sensor, and Lynxmotion Pan and Tilt Kit as the s`ervo (2 servo motors, in this case, both for sweeping horizontally) for testing purposes. By connecting the LIDAR sensor to the Lynxmotion Kit, it is possible to configure the Kit to rotate at a constant speed, and to have the LIDAR sensor sample at a regular interval. Knowing the speed of the servo and the time between LIDAR samples, it is possible to construct a map as described above from the LIDAR readings.



*Figure 13: LIDAR Lite V3 (left) and Lynxmotion Pan and Tilt Kit (right)*

While the Lynxmotion motors were adequate for testing purposes, they had far too much shaking for use in the final product. Additionally, since they were servos, they were incapable of rotating continuously, meaning that they had to rotate 360°, stop, and then rotate in the opposite direction. This is undesirable behavior for a sensor that serves as the primary safety measure on the entire system.

With this in mind, the team decided to purchase the ROB-09328 stepper motor for use in the final product. This motor is capable of rotating continuously and also keeps track of how far it has rotated, making it perfect for the application in question (since keeping track of the angle of the sensor is extremely important). The naive approach would be to mount the LIDAR sensor directly on top of the motor; however, since wires need to be run out of both the sensor and the motor, this would result in either the wires becoming hopelessly tangled after one rotation (if the motor is under the sensor) or needing to build supports above the sensors that would block its range of vision (if the sensor is under the motor).

For these reasons, it was decided that the LIDAR system would utilize a gear system, with one gear being mounted on the motor and the other one serving as a platform for the sensor itself. This permits the wires from the sensor to drop through the gear and through a slip ring that allows for continuous rotation. All of this was housed inside of a custom-made 3D-printed casing that provided all the components of the system (the sensor, the slip ring, and the motor) protection from the elements. A rendering of this housing can be seen below in Figure 14.

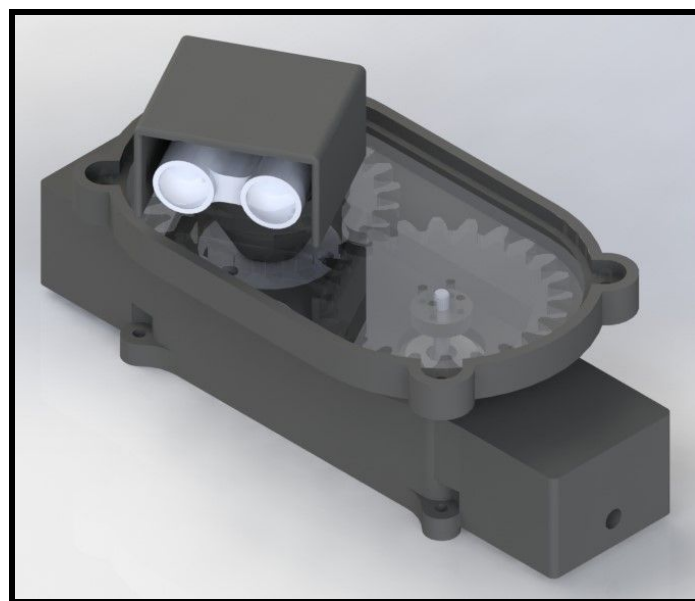


Figure 14: Rendering of the LIDAR housing

### 3.5 Software Design

The team was still to establish several key design parameters for the code the robot runs. In addition, the team also purchased and began to experiment with the implementation of high-end GPS chips.

The team began by brainstorming the general outline of what the code will have to look like once loaded into the completed robot and eventually settled on several key points. Firstly, the robot would have to stay within a set of user-defined boundaries. These boundaries could exist purely within software or take the form of physical beacons placed along the edge of the area to be cleared, but the important part is that these boundaries should be easily visible to the user and highly customizable to account for uneven property lines or irregularly shaped obstacles. Initially, the plan was to use radio beacons mounted on stakes that could be driven into the ground at corners in the border, much like those used in “invisible fence” systems for dogs. However, the team eventually decided that creating the beacons and integrating them with the rest of the robot would be more effort than it was worth, so they settled on geofencing. Geofencing relies on the robot’s onboard GPS and is entirely software-based, so it would require no additional setup from the user beyond defining the fence using some sort of UI. A mockup of this UI is shown below.

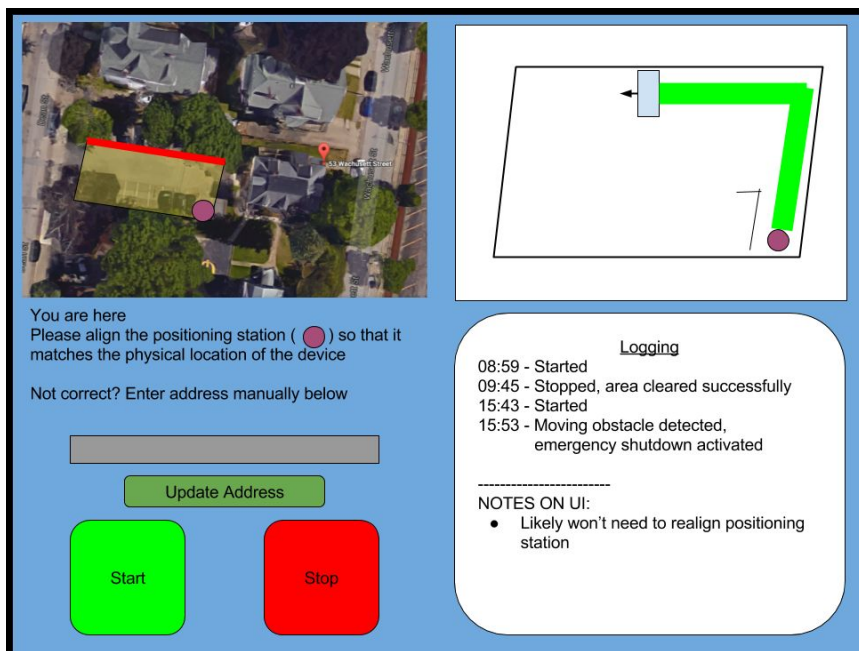


Figure 15: User interface concept

The second major design specification was that the robot should be able to detect and avoid obstacles, both stationary and mobile. This was arguably the single most important part of the entire robot: if it fails, the robot could damage itself and the user's property, and possibly kill or maim someone. Thus, obstacle detection is of the highest priority. This, in turn, required high-quality sensors capable of detecting obstacles at significant range in heavy snowfall, a problem covered in more detail in other sections of this paper. Processing the data from these sensors is at least as important as making sure that the sensors are calibrated correctly. Although some noises can be eliminated using op-amps and band pass filters, there was still a significant amount of erroneous data due to the amount of snow.

Once an obstacle is detected, it should obviously be avoided. How this avoidance is actuated depends on the obstacle itself. Obstacles can be broadly split into two categories: stationary, which includes things like trees, parked cars, and houses; and mobile, which includes things like animals, moving cars, and people. Stationary obstacles should ideally be pathed around as closely as possible (within reason) to clear the maximum amount of snow possible. Mobile obstacles, on the other hand, should cause the robot to stop entirely as soon as they enter a specified radius around it, then start up again when they leave. This model has several problems, however. Firstly, categorizing obstacles as stationary or mobile is much easier said than done. For example, how can the robot differentiate between someone standing still and, say, a tree? Obviously anything that's currently moving should be treated as a mobile obstacle and cause the robot to stop, but how can the robot be 100% sure that the "stationary" obstacles it's pathing around aren't just temporarily stopped mobile obstacles?

The elegant solution to this problem would be to keep track of every obstacle as a separate entity within the virtual map the robot has stored, with every obstacle having a flag indicating whether it's mobile or stationary. Obstacles would be initially assumed to be stationary, then permanently flagged as mobile as soon as the robot detects movement from it. This solution may run into issues with the aforementioned sensor noise, as the robot could interpret this noise as movement and permanently stop due to a nonexistent "mobile" obstacle. Finding a balance between not running people over and not stopping every two feet due to sensor interference was one of the major challenges of this project. Erring on the side of caution is preferable (a stopped robot is better than an injured person), but having the robot stop too often is also bad.

The next design specification was that the robot's code should also allow the snow to be thrown to a specified obstacle-free location. This location could be either defined by the user or

automatically selected by the robot based on some set of criteria. This leads to two conclusions: one, that the impeller should be able to rotate and have that rotation controlled by the microprocessor on board (not directly driven from the engine); and two, that obstacle detection must extend beyond the user-defined limits of the area to be cleared. The obstacle detection discussed previously technically has no reason to care about obstacles a good distance away from the edge of the area, since the robot will never get close enough to them to hit them. However, when throwing snow, it is imperative that the robot first ensures that its chosen dumping area is completely free of obstacles. While not quite as lethal as being run down by the snowblower itself, being hit in the face by a massive jet of snow is obviously something to avoid if at all possible.

There were two main challenges that could have stopped the team from achieving this goal. First, since the snow could potentially be rather light, wind could have a significant effect on its parabolic arc, causing the snow to land far away from its intended location or even be blown right back at the robot. If this project was to eventually be released as a consumer product, this could probably be fixed by adding some form of wind speed sensor to either the robot or the base station and including that data in the calculations. The second factor to consider is that the snow being thrown will necessarily block the sensors from detecting obstacles at the location being thrown to. This seems like it could be solved by ensuring the area is clear when throwing begins and then continually checking either side of the stream for mobile obstacles headed towards it; however, this does not deal with the corner case of someone (or something) approaching the stream of snow head-on. While this is admittedly unlikely, it's not entirely possible to rule out. Again, the team will most likely be ignoring this corner case for the project, but if it were to become a consumer product there would have to be measures to prevent this.

The final design constraint is that all of these elements should be visible to the end user via some form of user interface, much like the mockup shown previously. One of the main decisions to make regarding this UI was where exactly it should be displayed. The team eventually settled on a standalone base station, preferably with some form of touchscreen. The main reason behind this decision was related to the GPS system, which requires a stationary base in order to calibrate itself. However, this did not rule out the possibility of having the associated software be a standalone app, capable of running on phones or computers.

The project implemented a differential GPS system (DGPS), which is a method that allows consumer-grade GPS chips to get centimeter-level accuracy. This amount of accuracy is

necessary to define the robot's boundaries using geofencing – a few centimeters could be the difference between cleaning someone's driveway and taking out their side-view mirror. Traditional GPS chips use signals from four satellites at known distances and angles to calculate their own position on the Earth. This can result in varying levels of inaccuracy due to factors such as chip quality, obstacles between the receiver and a satellite, or one or more of the satellites being too close to the horizon. To solve these issues, DGPS operates on the assumption that two GPS receivers located close to each other will experience the same atmospheric interference and therefore have the same inaccuracies. DGPS uses two GPS chips: one on the thing to be tracked (in this case, the robot), and an immobile one used to calculate corrections (in this case, on the base station). The immobile GPS knows that it can't possibly move and can thus ignore or average out variations in the data to obtain a "true" set of coordinates for its own position. Once it obtains these coordinates, it compares every new GPS signal to them. The difference between the base station's known location and its calculated location is then used as the offset for the roamer's position at that moment, resulting in accurate location calculation.

There are many different methods that can be used to attain this desired behavior. For example, the corrections can be applied in post-processing to correct logged locations from the roamer after the fact. The one most suited to this particular application, however, is most likely real-time DGPS. This method involves the base station transmitting corrections to the robot in real time via a radio link. From the end user's perspective, this would involve setting up a base station somewhere with a relatively clear line of sight to the sky and calibrating its position.

## 4 Methodology/Implementation

This chapter details the implementation of the electrical, software, and mechanical sub-systems of the autonomous snowblower. These sub-systems are designed to meet the requirements put forth in Chapter 3.

The final system includes the autonomous snowblower (snowblower), a base station for high-precision GPS measurements (base-station), and a standalone computer application (app).

## 4.1 Hardware Configuration

As discussed in section 3.3, the snowblower includes a Raspberry Pi (model 3b) to run the embedded software for the snowblower. This is powered by the alternator and connected to the motor controller as detailed in Figure 9.

The Raspberry Pi is connected to one of two identical uBlox high precision GPS modules via a Serial interface (USB). These GPS modules are used for differential GPS measurements as described in section 3.5; the GPS on the snowblower acts as the roaming GPS. The other uBlox GPS module acts as the stationary GPS, sending correction via radio link (see Figure 16 below).



**Figure 16:** *uBlox setup*

Included with the uBlox GPS modules is a computer application *u-center*. By default, both GPS modules are configured to be in roaming mode. In order to receive differentially corrected GPS, one must be configured to act as the base GPS. The base GPS is configured as shown below in *u-center*.



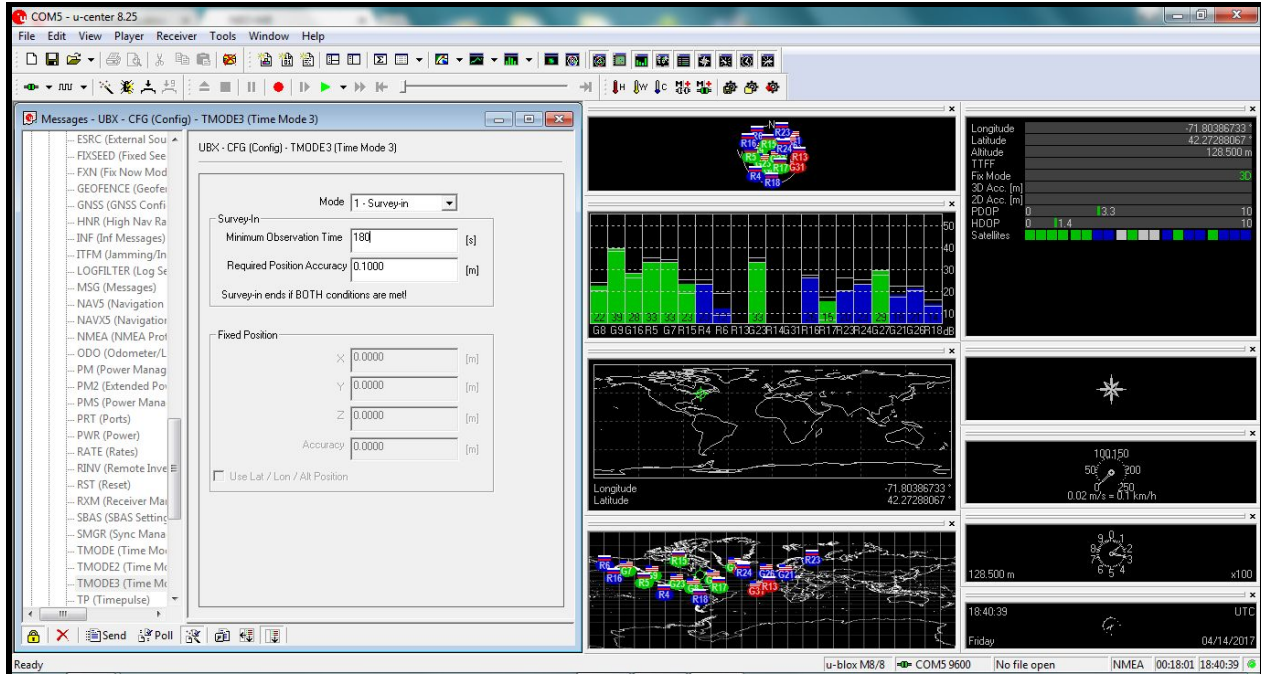


Figure 17: Example of u-center GUI during operation

Here, the precision of the base station is set to be 0.1 meters. Once configured properly, the base GPS will start to collect location readings. The module then calculates its position by averaging the position of the samples. Once the base station has determined its location, it will begin to transmit correction data over the radio link for the roaming GPS to pick up.

The roaming GPS is able to use the correction data, along with its own readings, to generate differentially corrected readings, which gives the roaming GPS a high level accuracy, compared to the roaming GPS acting on its own.

A LIDAR module is used by the system to detect obstacles in the snowblower's environment. It is connected to the Raspberry Pi via the I2C interface, while the servo motors are controlled by an Arduino uno (acting as a PWM driver), which is connected to the Raspberry Pi via the SPI interface. This is due to the limitations of the Raspberry Pi; only two hardware PWM modules are included on the Raspberry Pi, which are already used by the motor controller, thus the Adafruit PWM driver is needed to provide the additional hardware PWM signal.

## 4.2 Software Implementation

The backbone of the software in the project is ROS (Robot Operating System). ROS is a powerful platform that provides tools to aide in developing robots, including a powerful mapping

engine, *gmapping*. It uses a system of nodes and topics. Nodes are “processes that perform computation” (wiki.ros.org/Nodes). In the case of the project, they are used to interface with various hardware components (sensors and actuators), as part of the user interface, and to interface with the *gmapping* system. Topics are “channels” used to allow different processes (running on either the same or a different computer) to communicate with each other. A node can subscribe or publish to a topic. Publishing to a topic sends messages on that channel. If a node subscribes, it “listens” to the channel; when any message is published, the node will process the incoming message.

For the project, the majority of the coding is done in Python. ROS has a client library for Python. Additionally, there are multiple open source libraries for interfacing with the raspberry pi’s GPIO written in Python. In terms of UI development, Python contains a powerful GUI module *tkinter*.

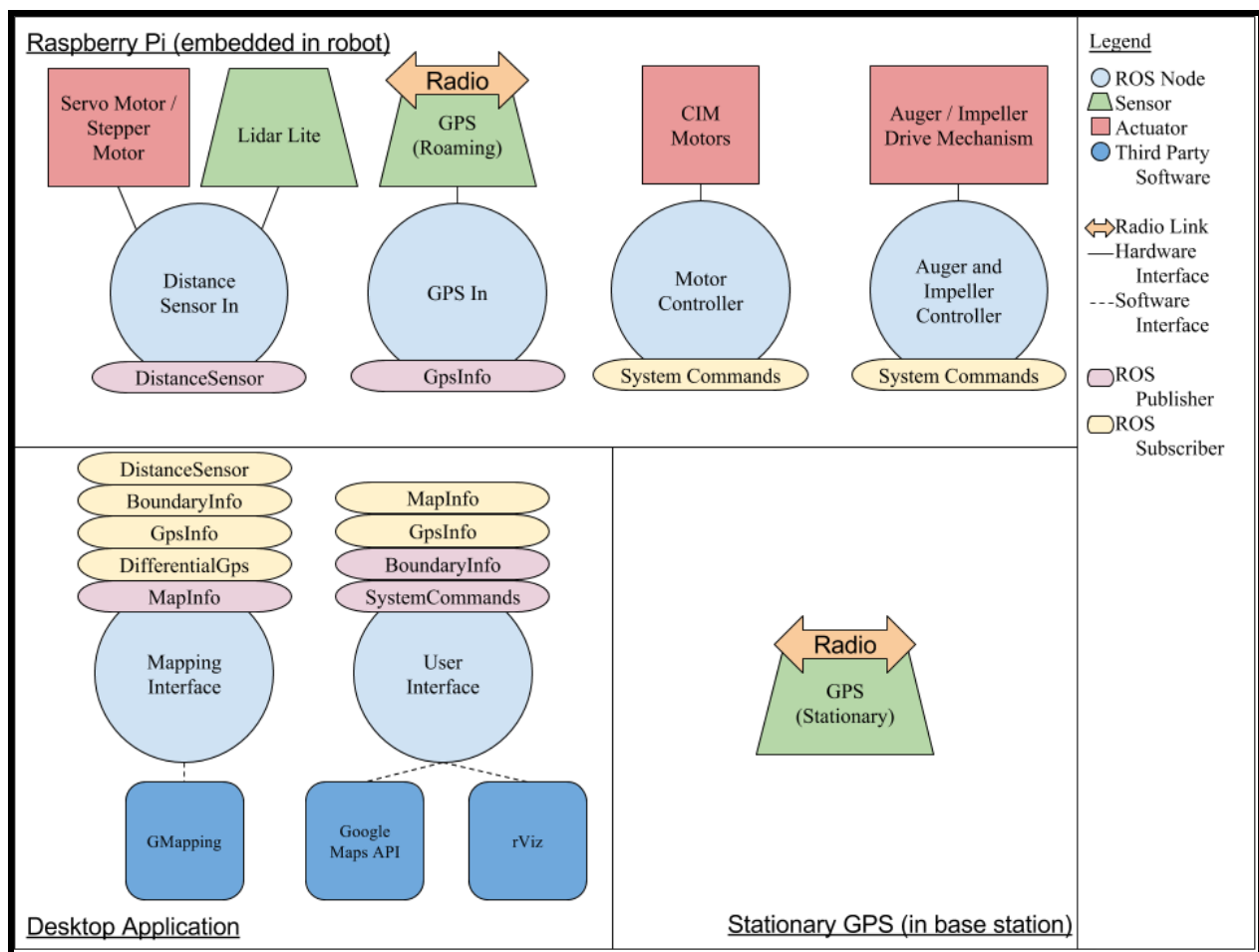


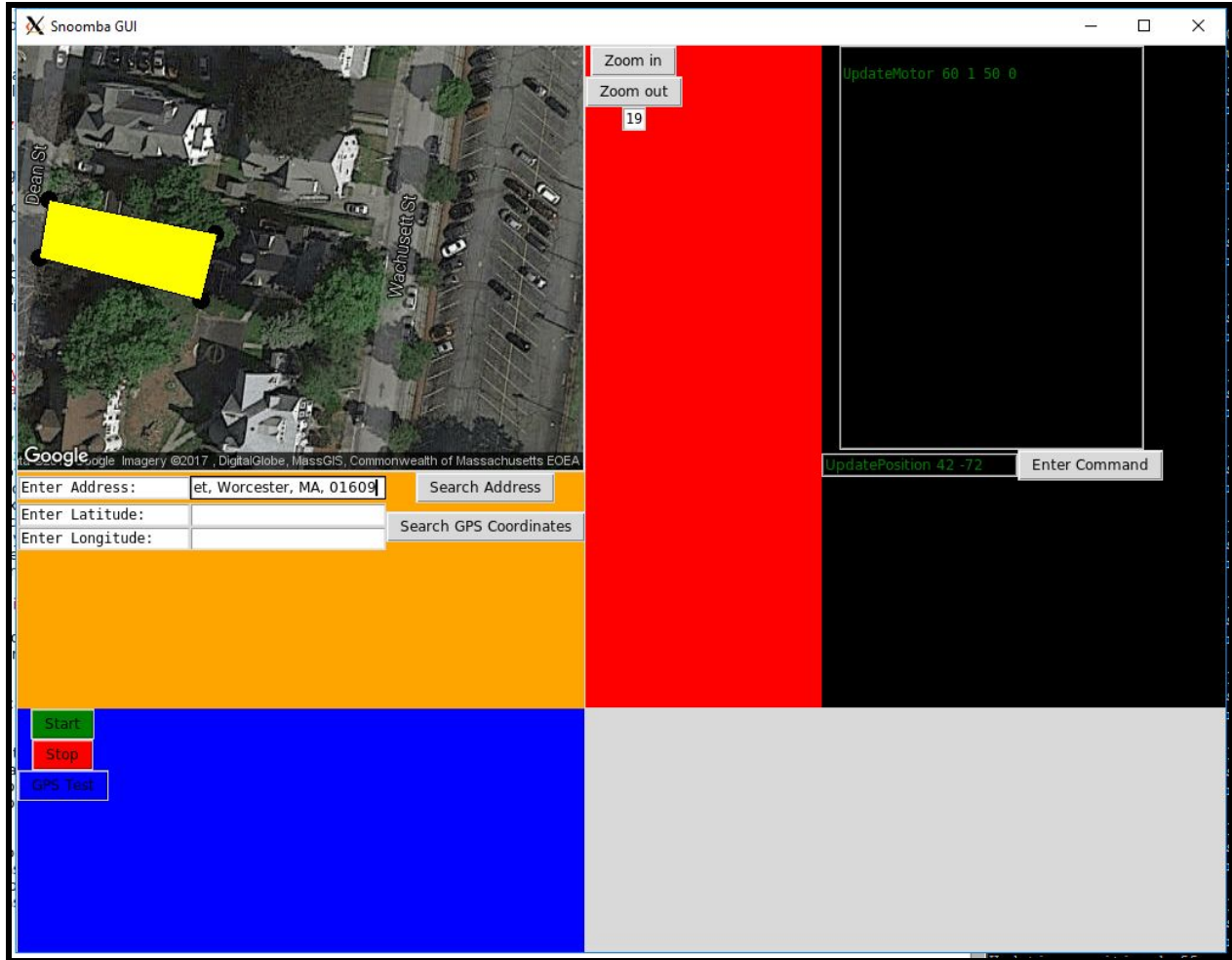
Figure 18: Software interface

The figure above documents how the software is divided into two separate applications, the responsibilities of the various nodes, and the various channels of communications that allow the embedded software to communicate. The desktop application, in addition to running the *User Interface* and *Mapping Interface* nodes, runs *roscore*, which spawns a ROS server (accessible at <http://nicholas-raspi.dyn.wpi.edu:11311> by default). As long as the Raspberry Pi is on the same WiFi network, it is able to communicate with the desktop application, and the nodes can communicate across the various topics regardless of which system the nodes are running on.

Note that for the final system, all nodes run directly off of the Raspberry Pi, while the *User Interface* and *rViz* can be displayed on a remote machine over SSH.

#### 4.2.1 User Interface

The *User Interface* node is responsible for providing the end user control over the autonomous snowblower, as well as relay system and mapping information back to the user. Shown below is the current state of the user interface.



**Figure 19:** Example of the GUI during operation

In the top left corner is the mapping interface. The map is provided from Google, using the Google Static Maps API. In order to retrieve the map, a call is made to the following URL - <https://maps.googleapis.com/maps/api/staticmap> - using the wrapper function `googleApiRetrieveStaticImage`, while providing the appropriate latitude, longitude, and zoom level. The node keeps track of these parameters, and allows the user to alter them using the controls surrounding the map. When the user changes one of these parameters, a call is made to this function to replace the image. Of particular note is the address bar, which allows the user to bring up an image centered around the given address. In order to do this, another function `googleApiRetrieveCoords`, is used to make a call to the Google Address API at the URL <https://maps.google.com/maps/api/geocode/json>. This call is supplied with address given by the user, and spits back the latitude and longitude of the best matching address returned from the

API call. This latitude and longitude are used to call the aforementioned update image function, thus displaying the specified address.

On the map is a movable polygon. The user can drag the vertices, and thus move the object. The polygon is used to specify the boundaries for the autonomous snowblower. Whenever the user moves the boundaries, the position change is relayed through a message over the *BoundaryInfo* topic. The message follows the format of:

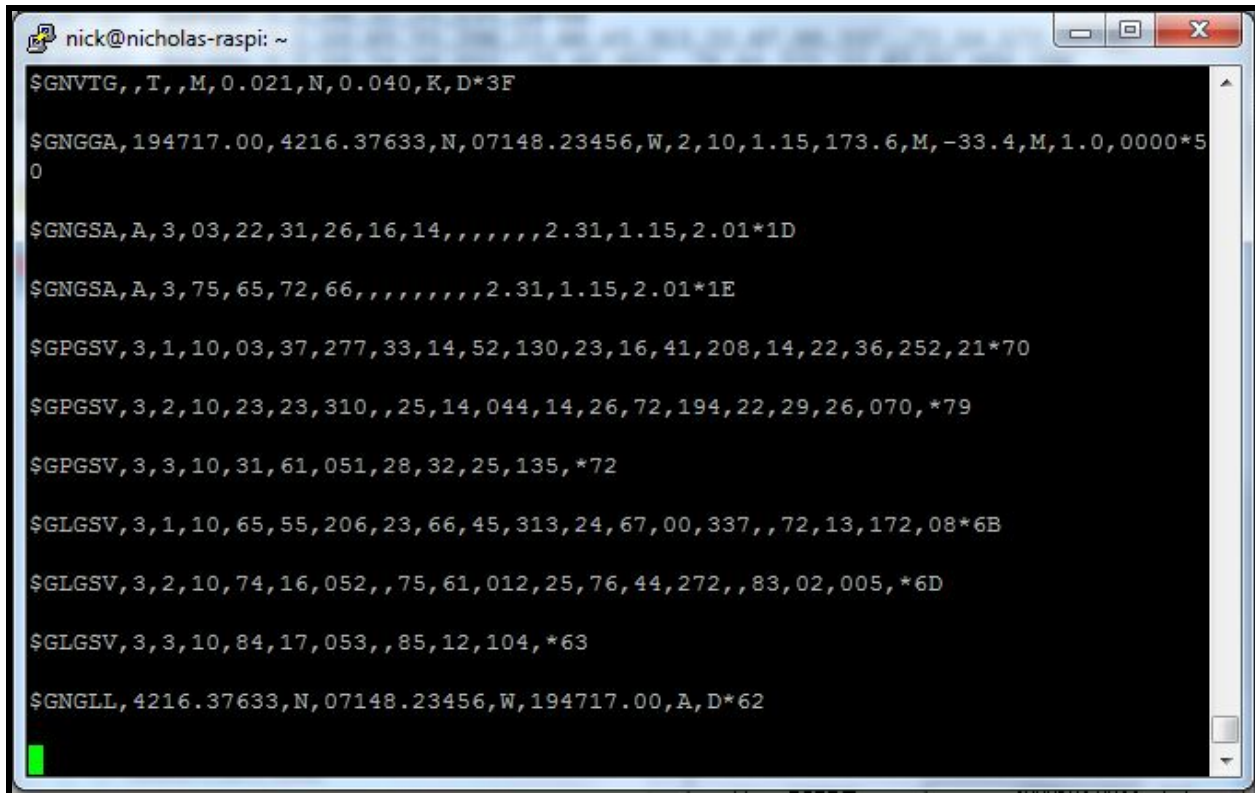
```
UpdateCorner <corner number> <latitude> <longitude>
```

*Corner number* is a unique identifier for each vertex equal to  $\{0,1,2,\dots,n-1\}$ , where  $n$  is the number of sides of the polygon. *Latitude* and *longitude* correspond to the new GPS coordinates. To calculate the updated GPS coordinates, a set of open source python functions (found at <http://stackoverflow.com/questions/7490491/capture-embedded-google-map-image-with-python-without-using-a-browser>) are given the zoom level and GPS coordinates of the center of the map. These serve as the base for calculating the GPS coordinates of other pixels. The code also receives the difference (in pixels, horizontal and vertical) between the point where the vertex is placed and the center of the map. Given this information, the code calculates the GPS coordinates of the given pixel, which in this case corresponds to the vertex's new boundary. The messages sent over *BoundaryInfo* are received by the *Mapper* node to provide boundaries in the backend mapping system (`g_mapping`).

#### 4.2.2 High Precision GPS Sensor

The onboard uBlox GPS module acts as the roaming GPS, as discussed in section 4.1. As such it receives differentially corrected messages to provide highly accurate position readings of the robot. However, this only works if the base station is properly transmitting position correction data, and the roaming GPS is able to receive the data. The *gps* node (running on the Raspberry Pi) is responsible for communication with the GPS module over this line. To achieve this, the node utilizes Python's built in *Serial* library to set a connection to receive and send messages from/to the GPS module. The Baud rate is set at 9600, as this is the rate that the GPS module operates at. The uBlox GPS module communicates via the *NMEA* and *UBX* protocols. *NMEA* messages are used to relay position updates, as well as meta-data about the state of the GPS module, such as connection status to various satellites. It is a very comprehensive and widely used protocol. *UBX*, on the other hand, is a protocol made specifically for uBlox products, and is used to relay device information. The node is configured to read in messages over the established Serial line, parse each message sequentially, and

then issue the appropriate action based on the message type and content of the message. The figure below shows various messages sent to the node for processing.

A terminal window titled 'nick@nicholas-raspi: ~' with a black background and white text. It displays several NMEA messages from a uBlox GPS module. The messages are: \$GNVTG, ,T, ,M, 0.021,N, 0.040,K,D\*3F; \$GNGGA, 194717.00, 4216.37633,N, 07148.23456,W, 2, 10, 1.15, 173.6,M, -33.4,M, 1.0, 0000\*50; \$GNGSA, A, 3, 03, 22, 31, 26, 16, 14, , , , , , , , , 2.31, 1.15, 2.01\*1D; \$GNGSA, A, 3, 75, 65, 72, 66, , , , , , , , , 2.31, 1.15, 2.01\*1E; \$GPGSV, 3, 1, 10, 03, 37, 277, 33, 14, 52, 130, 23, 16, 41, 208, 14, 22, 36, 252, 21\*70; \$GPGSV, 3, 2, 10, 23, 23, 310, , 25, 14, 044, 14, 26, 72, 194, 22, 29, 26, 070, \*79; \$GPGSV, 3, 3, 10, 31, 61, 051, 28, 32, 25, 135, \*72; \$GLGSV, 3, 1, 10, 65, 55, 206, 23, 66, 45, 313, 24, 67, 00, 337, , 72, 13, 172, 08\*6B; \$GLGSV, 3, 2, 10, 74, 16, 052, , 75, 61, 012, 25, 76, 44, 272, , 83, 02, 005, \*6D; \$GLGSV, 3, 3, 10, 84, 17, 053, , 85, 12, 104, \*63; \$GNGLL, 4216.37633,N, 07148.23456,W, 194717.00, A, D\*62. A green cursor is visible at the bottom left of the terminal window.

```
nick@nicholas-raspi: ~
$GNVTG, ,T, ,M, 0.021,N, 0.040,K,D*3F
$GNGGA, 194717.00, 4216.37633,N, 07148.23456,W, 2, 10, 1.15, 173.6,M, -33.4,M, 1.0, 0000*50
$GNGSA, A, 3, 03, 22, 31, 26, 16, 14, , , , , , , , , 2.31, 1.15, 2.01*1D
$GNGSA, A, 3, 75, 65, 72, 66, , , , , , , , , 2.31, 1.15, 2.01*1E
$GPGSV, 3, 1, 10, 03, 37, 277, 33, 14, 52, 130, 23, 16, 41, 208, 14, 22, 36, 252, 21*70
$GPGSV, 3, 2, 10, 23, 23, 310, , 25, 14, 044, 14, 26, 72, 194, 22, 29, 26, 070, *79
$GPGSV, 3, 3, 10, 31, 61, 051, 28, 32, 25, 135, *72
$GLGSV, 3, 1, 10, 65, 55, 206, 23, 66, 45, 313, 24, 67, 00, 337, , 72, 13, 172, 08*6B
$GLGSV, 3, 2, 10, 74, 16, 052, , 75, 61, 012, 25, 76, 44, 272, , 83, 02, 005, *6D
$GLGSV, 3, 3, 10, 84, 17, 053, , 85, 12, 104, *63
$GNGLL, 4216.37633,N, 07148.23456,W, 194717.00, A, D*62
```

Figure 20: NMEA messages from the uBlox roaming GPS module as they are processed by the gps node.

In this system, only the *GNGLL* messages trigger any actions, all others are discarded. *GNGLL* messages are part of the *NMEA* library, and are used to relay position measurement data. The first four data points (separated by commas) provide the location of the module. The final data point before the checksum (separated by ‘\*’) will either be ‘A’ or ‘D’, for Autonomous, or Differentially corrected. In the case of the last line in Figure 20. above, the measurement is differentially corrected. When processing these messages, the *gps* node sends a message over the *GpsInfo* topic. The message is in the format of

```
GLL <latitude> <longitude>
```

The latitude and longitude are parsed from the message using a Python NMEA parsing library found at <https://github.com/Knio/pynmea2>. The processing for this message type can be extended to include a check for the nature of the measurements; whether they are differentially corrected or not, and provide the user feedback that they are not getting the best possible

measurements. The messages sent over the *GpsInfo* are processed by the *Mapper* node/*g\_mapping*.

### 4.2.3 Lidar Sensor

The LIDAR sensor interacts with the Raspberry Pi via an I2C connection. This connection is managed by the *LidarNode*, which is responsible for triggering and reading distance measurements from the Lidar Lite sensor. It continuously runs a loop to trigger and read measurements, while at the same time rotating in increments the servo motors which controls the rotation of the Lidar Lite sensor. In doing so, the node is able to capture 360 degree distance measurements. As the node receives measurement data from the Lidar Lite, it sends the information on the *DistanceSensor* topic in the following message format:

```
UpdatePosition <angular position> <distance (cm)>
```

Note that each message corresponds to a single measurement. The *angular position* is provided by an internally stored variable, which is updated whenever the stepper motor changes positions. The Lidar module includes a little switch that is triggered once per rotation. This is used to calibrate the system, as there is no way to determine the starting position of the stepper motor, thus it would be impossible to tell which direction is forward. The *Mapper* node subscribes to the *DistanceSensor* topic, and uses the distance readings to aid in mapping (via *g\_mapping*).

### 4.2.4 *g\_mapping* and rViz Interface

Discussed in the previous sub-sections (4.2.1, 4.2.2, 4.2.3), the *Mapper* node receives input across the *BoundaryInfo*, *GpsInfo*, and *DistanceSensor* topics from the *Gui*, *Gps*, and *Lidar* nodes, respectively. There are callback functions in the *Mapper* node corresponding to each topic. Additionally, the *Mapper* node is responsible for interacting with *g\_mapping*. *G\_mapping* is a powerful, ROS based mapping tool that builds a map of the system's environment from the incoming distance scans (provided by LIDAR) and location information (GPS).

Whenever a message is sent across *BoundaryInfo* (by the *Gui* node), the *Mapper* receives and handles the message using the *biCallback* function. This function first parses out

the function arguments (corner index, gps coordinates), and uses them to update the system map boundaries in *g\_mapping*. This function is not currently fully implemented

Messages sent over *GpsInfo* (by the *Gps* node) are similarly handled by *Mapper* using the *giCallback* function. The position of the robot is updated via communicating with the *tf* node (spawned under *g\_mapping*). Note that normally, *g\_mapping* is used with a system that uses an odometer and not a GPS. As such, most examples using *g\_mapping* send odometry data, using the data between readings to update the position and rotation. However, it is possible to alter these directly without odometry data by sending a *tf/tfMessage* (message type in ROS) to the *tf* node, which is updated to include new position/rotation data based on GPS readings received. This function is not fully implemented, as it does not currently interact with *g\_mapping*.

Distance readings sent over *DistanceSensor* are processed similarly. The callback function for this topic, *dsCallback*, constructs *sensor\_msgs/LaserScan* messages to send to the *scan* node (spawned under *g\_mapping*). These messages are constructed by grouping together one full rotation of scans. The ROS based program *rViz* is used to visualize the map as it is constructed.

#### 4.2.5 Pathfinding

Pathfinding, at this point, has not been implemented yet. See discussion of future work (section 6).

#### 4.2.6 Motor Controller

The motor controller node includes a single listener, which listens on the *systemCommands* topic. The node listens for an *UpdateMotor* command sent on the channel, which is sent in the form of:

```
UpdateMotor <PWM 0> <GPIO 0> <PWM 1> <GPIO 1>
```

The pwm values are sent to the PWM driver, while the output from the PWM driver and the GPIO values are sent to the motor controller. As a reminder from section 3.5, the PWM is responsible for altering the speed of the connected motor, while the GPIO (binary; 0 or 1) is responsible for setting the direction of the motor.

This module is disabled by default, as the PWM driver discussed in section 3.4 is not implemented. As such, priority is given to the *LidarNode* to use the two PWM modules on the Raspberry Pi.



### 4.2.7 Packaging

A ROS launch file is used to start and stop the program in its entirety. The file first launches a bash script with the following two lines:

```
source /opt/ros/kinetic/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/catkin_ws/
```

The first line is responsible for configuring the environment to use ROS. The second line updates one of the ROS environment variables - *ROS\_PACKAGE\_PATH* - to include the user's *catkin\_ws* directory, which includes the raw source code for the snoomba.

With the environment properly configured, the launch program can start up both third-party nodes, as well as the nodes made specifically for this project. Nodes are specified using the *<node>* tag in the launch file, while the *<machine>* tag is used to specify which machines the nodes will run on (embedded on the Raspberry Pi, or on the user's own machine).

## 4.3 Mechanical Implementation

To start, the team had to gut the snowblower. The undercarriage, which was to house most of the new components, needed to have the old drive train removed in order to make room. Once this was accomplished, the sides of the chassis were cut down to remove a lip running around the edge that was blocking the gearboxes. A plasma cutter was used to widen the holes that the gearboxes were to mount onto.

The next step was to obtain parts for the tread mounting and tensioning system shown in Figures 4 and 5. After some searching, the team was able to find a local scrapyard named Sullivan Metals graciously willing to provide the parts for free. Since there was no area for the tapped shaft collar to push up on and the team only had  $\frac{1}{4}$  in scrap, they doubled them up to make for a thicker area to push on. In order for the tensioning system to be effective, the threaded shafts need something to push against that will move the axle back and forth. The original idea was to use two pieces of aluminum scrap; however, due to the thinness of the scrap provided ( $\frac{1}{4}$ "), the team doubled it up on each side of the axle, using four pieces in total. They then tapped the 80/20 so that it could be easily connected to the base of the snowblower. After that, they drilled some holes to allow for the axle to pass through the block. They then drilled and tapped the shaft collar for the threaded rods. Finally, the rod was cut down to the appropriate size.

## 5 Results

This section details the capabilities and limitations of the robot's design and implementation to date in a variety of areas.

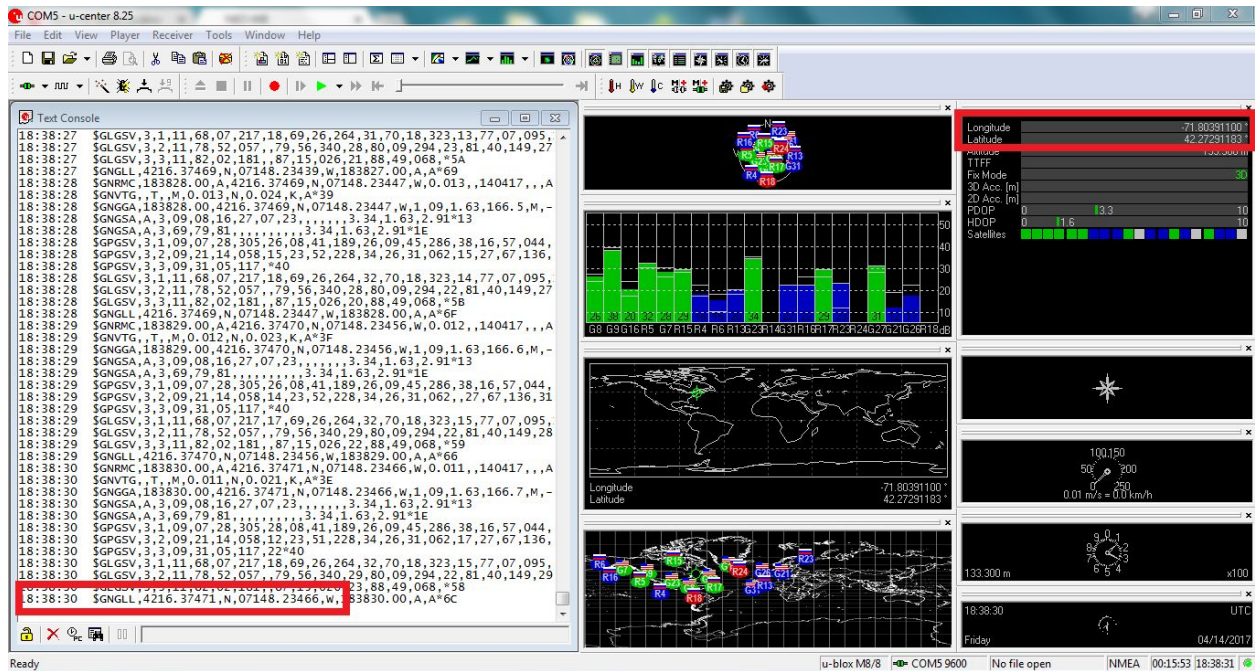
### 5.1 GPS

The uBlox GPS module turned out to be precise, but not necessarily accurate. As shown in the figure below, the differentially corrected GPS consistently reported the same location, even over 50 samples (~2-3 minutes of sampling). However, the actual position of the measurements is significantly off, as shown by the difference between the GPS's actual and measured positions.



**Figure 21: Measured GPS position vs. actual GPS position**

The offset in this instance can be attributed to the message processing techniques used by the Raspberry Pi. The uBlox module attached to the snowblower (roaming GPS) sends messages to the raspberry pi in the NMEA message format, as discussed in section 4.2.2. The Raspberry Pi then uses the raw latitude and longitude from the *GNLL* messages; however, this is not the correct latitude and longitude displayed by the *u-center* program (uBlox's proprietary software for interfacing with and setting up the uBlox GPS modules). The program uses corrections from other messages, most likely the *GNGGV* and *GNRMC* messages. These messages provide fix information regarding the position of the GPS.



**Figure 22: u-center configuration showing the raw lat. & lon. (left) vs. the corrected lat. & lon (right)**

Rather than applying the corrections provided by the other NMEA messages, the *GpsNode* sends the raw latitude and longitude to the *MapperNode*, where an offset is applied. This offset is meant to correct the raw lat. & lon. such that it is more accurate, and closer to the real lat. & lon. of the module.

To calculate this offset, the team first gathered a set of 50 raw data points from the GPS (via the *GNLL* messages), and then calculated the average latitude and longitude of these points. Next, the team used *u-center* to determine the actual position of the GPS module. Finally, the team calculated the difference in latitude and longitude between the actual position and the measured position. This difference was as follows:

$$\text{Lat\_offset} = \text{actual\_lat} - \text{measured\_lat} = 42.21292400 - 42.16373864 = 0.10918536$$

$Lon\_offset = actual\_lon - measured\_lon = -71.80409283 - (-71.482420478) = -0.321672352$

As shown in figure 21. above, this correction is not quite accurate enough for use in the snowblower. The measured position, with the offset applied, is still inaccurate by about 3 meters. Given that boundaries are defined by GPS coordinates, this offset would cause the robot to operate in an area that is shifted 3 meters from the user defined boundaries. The robot may believe it to be inside the user defined boundaries, but due to the GPS error, it could be up to 3 meters outside.

## 5.2 LIDAR

The LIDAR sensor works as expected. The Raspberry Pi is able to receive the distance measurements from the Lidar Lite sensor over I2C. In order for the Raspberry Pi to use the I2C module, it needed to be configured to do so, as this functionality is not enabled by default. To enable this behavior, the user must run the command `sudo raspi-config -> advanced options (8) -> I2C (A7)`. Once enabled, the Raspberry Pi can communicate using its I2C bus. The Lidar Lite sensor is set up as an I2C slave (on channel 62, default I2C address of the Lidar Lite module), as shown below in figure 23.

```
nick@nicholas-raspi:~$ i2cdetect -y 1
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  62  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
nick@nicholas-raspi:~$
```

Figure 23: Raspberry Pi I2C connections - Lidar Lite connected on channel 62

Using a python I2C library, the Raspberry Pi is able to parse incoming messages from the Lidar Lite. Given the position of the servo motors, along with the distance information from the LidarLite, the *LidarNode* is able to send the polar coordinates of the distance readings over the *LidarInfo* topic, with both the angle and distance included as shown below in figure 24 (see section 4.2.3).

```
nick@nicholas-raspi: ~  
----  
data: PC 28 114  
----  
data: PC 24 59  
----  
data: PC 20 61  
----  
data: PC 16 120  
----  
data: PC 12 142  
----  
data: PC 8 201  
----  
data: PC 4 103  
----  
data: PC 0 5  
----  
data: PC 4 5  
----  
data: PC 8 5  
----  
data: PC 12 219  
----  
data: PC 16 113  
----  
data: PC 20 96  
----  
data: PC 24 110  
----  
----
```

*Figure 24: Datastream from the DisatnceSensor topic*

## 6 Future Work

Due to several unforeseen hurdles and issues that arose over the course of this project, it was unfortunately unable to be completed on time. This leaves several sections with large amounts of room for improvement.

### 6.1 Further Mechanical Work

#### 6.1.1 Aiming the Chute

Initially conceived as a stretch goal, this particular goal is one of the more involved ones left to implement. It requires several components, including robust LIDAR sensor readings and path planning capable of taking into account snow piles it creates itself. Additionally, there would need to be some way to aim the auger using a microcontroller, most likely with a belt drive and a stepper motor.

### 6.1.2 Controlling the Auger

A more essential goal for future work is effectively engaging and disengaging the auger. This is accomplished via a Bowden cable, much like the brakes on a bicycle. When the cable is taut, the auger is engaged; when it is slack, the auger is disconnected from the engine. Being able to tighten or loosen the cable at will would allow the robot to turn on without the auger on, as a safety measure, and would allow the user to disengage it remotely for safety reasons. The team's planned solution was to use a motor with a spool on it to tighten and loosen the cable at will; however, pneumatic systems may be more suited to this task.

### 6.1.3 Designs for the Future

A design the team could not implement in time was the motor mounts. Due to the fact that the gearboxes arrived so late and a team member having to deal with personal issues, the team was unable to create a mount for the motors that would attach them to the gearboxes. One challenge involved in designing the mounting mechanism for the motors was the physical limitations involved in the undercarriage, which made using allen wrenches awkward or downright impossible. Additionally, they never got a chance to design the coupling between the CIM motors and the gearboxes, which is needed due to the fact that the drive shaft of the CIM motor is significantly smaller than the gearbox's hole. This coupler could easily be machined on a lathe.

### 6.1.4 Battery Mount

Mounting the batteries was another challenge the team could not surpass. The problem they mainly ran into, other than time, was running out of space on the snowblower. Putting the batteries on the back of the snowblower was considered; however, the tread mount got in the way. In any case, the final battery housing case must be watertight and have wire access to both the microcontroller and the alternator, wherever on the snowblower it ends up being installed.

### 6.1.5 Alternator

The other place where the team ran out of time was with the alternator. They decided to drive the alternator with the engine that was originally used for driving the wheels. They found that they could take some of the casing off of the snowblower and mount the alternator to the

top of the snowblower. However, the belt that connects the alternator and the engine needs to be taut and the alternator needs to be connected in two different places on either side of the alternator. The other challenge presented was the fact that it needed to be taut and because of the fact that the mounting spots are on opposite sides of the alternator. They originally planned to have an alternator on the snowblower so that the batteries would not die, if not have a prolonged life.

## 6.2 User Interface Improvement

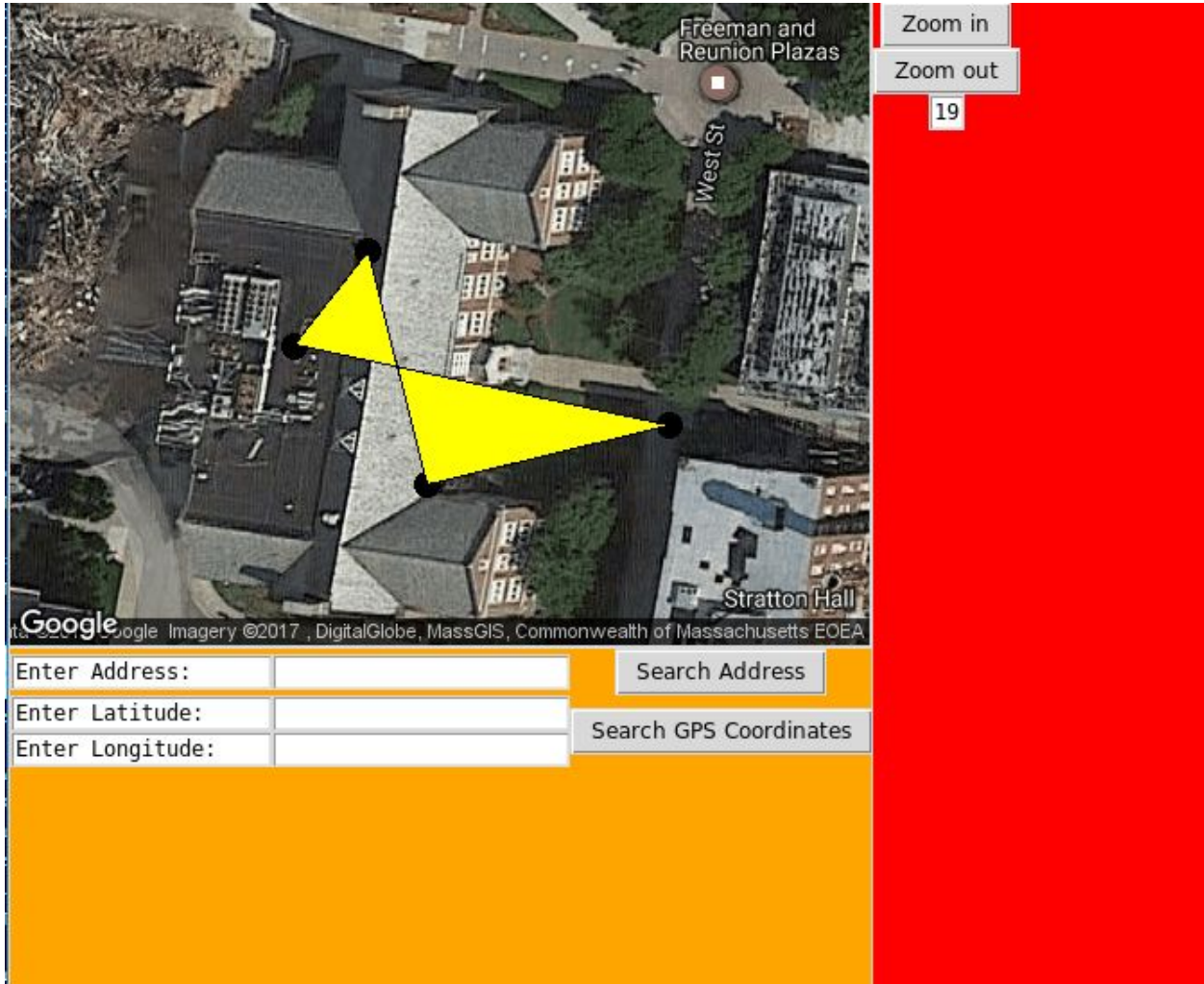
The user interface at the moment is relatively bare bones. It is able to send the necessary boundary information to the rest of the system, but there remain four areas where the user interface can be improved upon: additional console interaction, system feedback, boundary selection, and presentation.

The console interaction at the moment is very minimal. It has a limited number of functions that it can call, which are mostly used for direct interaction with the peripherals. However, the console could be greatly extended. First, it should be able to publish to all topics, not just the ones that the UI already publishes to. Next, it would be worthwhile to implement some ROS callbacks, especially *rostopic*, which would allow the user to view the messages sent over various channels.

Currently, the system does not provide much feedback to the user interface. It therefore may be worthwhile to additionally subscribe to all (or most) topics. Although the user interface currently receives messages from the mapper node (in the form of corrected GPS measurements), it does not use them, except to display the 50 most recent points to the user via the “GPS Test” button. It should, in addition to this, display the snowblower’s position, and update it every time a new gps reading is available.

The boundary selection window only allows the user to define a boundary within four corners, and gives the user no option to specify where the snow is sent to. As discussed in section (insert section here), one of their design requirements for the user interface is to allow the user to define where the snow is sent. As there is currently no chute control mechanism, there is no immediate need for this feature. However, should such mechanism be implemented, this feature would be needed immediately, as it would provide a target area for the chute to aim for. In terms of defining the boundary for operation, it should be extended upon to allow for a

variable number of corners, not just four (for areas more complex than a rectangular driveway). Additionally, there exists a bug in the current implementation of the boundary selection mechanism. As shown in figure 25. below, if a corner crosses over a figure, the area within the rectangle will not be filled, but will instead create two triangles due to the center lines crossing over each other.



**Figure 25:** Glitch in the boundary definition apparatus

Given the current implementation, there are two solutions to this problem. The first option is to dynamically reassign the corner indices as needed. This would require complex calculations to re.-assign the indices whenever one of the corners crosses over a non-connected line, making it difficult to implement, as well as being a strain on the processor. Otherwise, this reassignment would need to occur when gmapping accepts boundary info. This may not be possible with gmapping, and the issue still occurs that the boundary will not appear properly filled to the user.



## 6.3 Sensor Improvements

The sensors currently provide the snowblower with the information it needs to navigate a specified area autonomously. However, the GPS module is not perfectly accurate, and the lidar sensor requires better housing.

### 6.3.1 GPS Improvements

As discussed in section 5.1, the GPS module still has an offset of about 3 meters, due to the method that the latitude and longitude are extracted from the GPS messages. Currently, the team adds an offset which mostly corrects the raw latitude and longitude; however, it is still unsuitable for use by the snowblower.

The best way to improve this error is to use the correction information found in other message types sent from the GPS module. Further discussed in section 5.1, the *GpsNode* currently only listens for *GNGLL* messages.

### 6.3.2 Localization (LIDAR) Improvements

Although the final LIDAR system is fairly robust, it is not without its faults. The most obvious of these is the fact that the acrylic gears need to be recut, since they were accidentally cut at an angle and therefore have trouble meshing together. Additionally, the 3D-printed adapter that attaches the second gear to the LIDAR sensor above it and the slip ring below it needs to be reprinted to properly fit where it should. There are also potential changes or improvements that could be made to the main body of the assembly; for example, changing the attachment mounts from 1.5" 80/20 aluminum to some other size or shape might allow for better integration with the rest of the system.

The other major change that could be made is to the system that determines what angle the sensor is currently pointing in. Although the final build uses a stepper motor capable of tracking its own position, this is not entirely foolproof, as it is still susceptible to sudden impacts that might cause it to miss steps and therefore have inaccurate readings. The team's finalized solution to this problem was a bump switch mounted under the second gear in such a way that it is triggered once per revolution. This allows the software to calculate what angle the sensor is pointing in by multiplying the time since the last bumper switch interrupt by the gear's rotational velocity. This particular method was chosen since it was easy to implement and relatively

resilient to environmental interference. However, there are many other ways to achieve this same outcome. The most straightforward of these would be to attach an encoder to the shaft under the sensor, but there is potential in using a magnet and a Hall effect sensor in the same way the bumper switch is currently being used.

The final issue with the LIDAR system is due more to inherent limitations in the technology itself rather than any design flaws in the housing. LIDAR systems in general do not handle falling snow or rain very well, due to the snowflakes/raindrops causing interference and false “obstacle” readings. Since the final system was envisioned as being able to operate during heavy snowfall, this is a major issue. Unfortunately, as of the publishing of this paper, vision systems capable of reliably piercing through snow are not commercially available. There is some promise in a new process being developed by Ford that combines multiple LIDAR sensors with a post-processing algorithm designed to detect and correct for raindrops; however, this technology is still experimental at this time and also highly proprietary.

## 6.4 Navigation Improvements

Although the project never reached the stage where navigation was possible, the team still drew up several design specifications for the future pathfinding code of the robot during the initial design phase. More specifically, the robot’s code should take the aforementioned geofenced area and run a space-filling algorithm on it, generating a path that the robot can follow to most efficiently clear the specified area of snow. Space-filling algorithms are somewhat self-explanatory: given an enclosed area and a starting position, they will break the area down into a grid of “cells” and calculate a path that visits all of them. The space-filling algorithm used for this particular implementation will have to be slightly modified, as the robot clears snow several cells on either side of its calculated path (which describes the motion of the robot’s center of rotation). To account for this, the algorithm needs to be written in such a way that it considers cells off to the side of the path to be “cleared”, avoiding situations where the robot makes multiple passes over the same area. At the same time, however, there should be some overlap between these cleared but unvisited cells to avoid thin lines of uncleared snow that would occur due to the robot pushing snow slightly off to the side as it moves forward.

These aforementioned physical limitations of the robot also extend to obstacle avoidance. Since the robot moves on treads, it is capable of turning in place about its center of rotation. However, this might cause the side of the robot to move into uncleared snow. For

example, if the robot started clearing a square area on the rightmost side, got to the top border, and attempted to rotate 90° to the left, it would be pushing its left face – which obviously lacks any snow-clearing mechanisms – into potentially deep snowbanks. To avoid this, the turns generated by the space-filling algorithm should cause the robot to back up while turning at a slight angle, then position itself at the correct pose without colliding with snow.

## 7 Conclusion

Although this project's final product ended up not being entirely functional, it is not without its accomplishments. First, the team learned several valuable lessons about determining the scope of a project. Although this project was originally slated to take three terms, it ended up taking four; this was partially due to the sheer amount of work that needed to be done, but was exacerbated by unforeseen time-consuming hurdles such as renting out a workspace and ordering parts. Additionally, the team initially had difficulties determining the scope of the project and what goals would be realistic to achieve within what was originally assumed to be a three-term timeframe. This could have been partially remedied by setting hard, specific goals at the end of each term, which would have given us a sense of how far behind we were.

Secondly, the project has provided a robust jumping-off point for a future major qualifying project (MQP). Since the logistics and most of the base platform were completed, a future team would be able to focus all their efforts on troubleshooting and software development. Even if this project isn't directly continued, much of the research is applicable to a wide range of projects in robotics and beyond. For example, the LIDAR scanner and associated mapping packages in ROS can be repurposed to be used in virtually any robotic system.

As a whole, this project served as a valuable learning experience for all parties involved and laid down a solid foundation for future work.