Ice Inspection Robot

A Major Qualifying Project Report:
Submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science
By:

Derrick Brown (RBE)

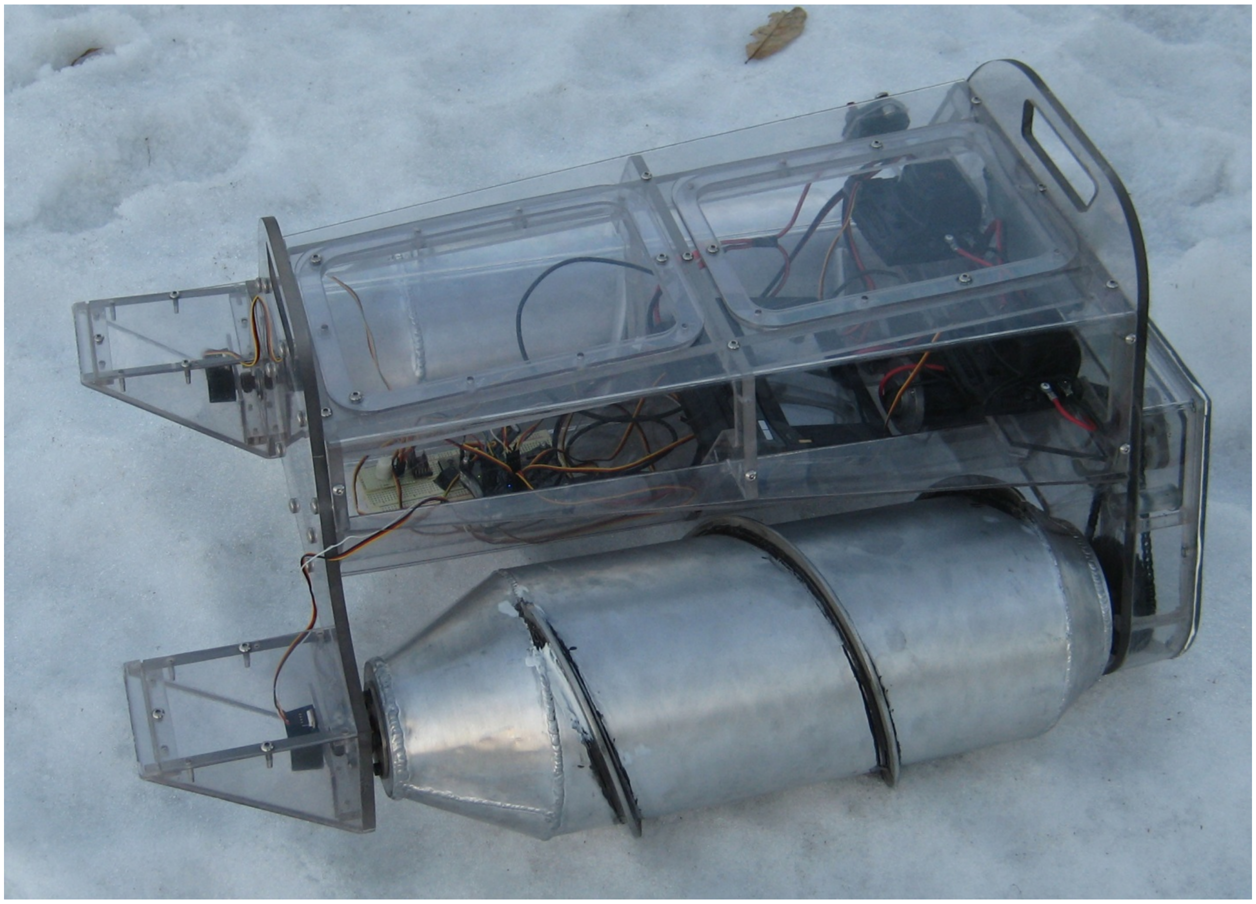
Minkyu Lee (RBE/ME)

Mark Mordarski (RBE)

Matthew Richards (RBE)

Approved:

Prof. Ken Stafford, Major Advisor
Prof. Fred Looft, Co-Advisor



Abstract

The purpose of this project was to design, create, and test a robotic mobile platform capable of housing and interfacing with an ice thickness measuring sensor. This robot was designed to: drive across natural snow, ice, and water surfaces; follow a user-defined path; report live position and heading information to a user. The unique auger-drive system of this robot was designed to provide efficient movement across ice, as well as buoyancy and aquatic propulsion, in the case of broken ice. A user interface was also designed and implemented as part of this project. This interface was designed to: display the live information sent by the robot; allow the user to send instructions to the robot; prompt the user for input; inform the user of the program's progress and state.

Acknowledgements

We would like to thank the following parties for their support throughout this project:

- Professor Kenneth Stafford, for providing his vast experience and support.
- Professor Fred J. Looft, for providing us with lab space and funding.
- Geophysical Survey Systems, Inc. for generously donating their time and consultation.
- Jake Cutler for assistance with aluminum welding.
- Neal Whitehouse for his advice.
- Mike Fagan for his manufacturing expertise.
- Jeff Gorges for teaching us how to weld steel.
- Catherine Emmerton for being a sweetheart.

Executive Summary

In cold weather climates, frozen ponds and lakes are a source of entertainment for many people. To ensure safety, there needs to be a reliable way to determine that the ice used is thick enough to hold the weight of those on its surface. The purpose of our project was to design a robotic solution to the problem of unreliable ice thickness reading. Such a robotic system would need to be able to drive on ice and snow, survive in water, keep track of its position, detect the thickness of ice, and send data to and receive data from a user. After review of the workload such a project would require, the scope of the project was narrowed down to exclude the actual sensing of ice. This left the goal of the project at designing and building a mobile, autonomous robotic system capable of housing and interfacing with an ice sensor.

The first step in the design phase was to research the capability of driving on ice. Extensive research was done on current drive systems, including track, propeller, wheel, and auger drives. After this research, it was apparent that the best drive method for this purpose would be the auger drive. This drive system provides good traction and mobility on ice, as well as the ability to float and drive in water. This was an important factor because, in the case of thin ice, the robot may end up breaking through into the water beneath.

After the auger drive type was decided upon, the specifics of the auger needed to be decided. Factors that needed to be considered included weight, pontoon shape and size, blade pitch and height, and strength. The total auger weight, along with the pontoon shape and size, were important factors because they determined the buoyancy that each auger would provide for the robot. The shape also determined the auger's ability to pass over obstacles in its path. The pitch of the blades determines the ratio of conversion from angular velocity of the augers to linear velocity of the robot. The height of the blades determines the propulsion in water. The strength of the auger as a whole is an important factor that determines survivability and maintainability of the system.

Concurrently with the auger design, the chassis was also being designed. The chassis needed to be able house the drive train, motors, motor controllers, the robot controller, and any sensors that the robot would use. In addition to housing these components, the chassis needed to keep these components protected from the elements (excessive cold, water, etc.). The chassis also needed to provide physical support for the augers to keep the system stable while driving. It

was decided that polycarbonate would be the ideal material to construct the chassis from, because of its relative strength, heat-insulating properties, and ability to be welded to form water-tight seals. Later developments in the chassis design let to a sloped front section for ease of surpassing obstacles, handles for ease of transportation, panels for access to interior components, and a heat sync to compensate for the polycarbonate's insulation of the heat from the motors.

Next, a decision was made on how the robot would be controlled. A few systems were considered, but rather quickly it was decided that the Neuron Robotics DyIO, in accompaniment with a Fujitsu Lifebook, would be the ideal system. The DyIO provided a system in which all of the low-level coding used for interfacing with sensors had already been completed. In addition, performing the higher-level code on an internal Lifebook would minimize the potential of over taxing the control system. These two components, used in conjunction, provided a relatively easy base upon which the software system could be developed.

The next major system that was developed for the project was the sensor suite that would be used for localization. The primary sensors that were considered were gyros, accelerometers, compasses, encoders, and GPS. GPS was eliminated from the potential sensors rather early in this design phase. The reason for this is that the range of error of a GPS is extremely significant when compared to the scale on which this robot was intended to drive. Later in the design process, accelerometers were eliminated as well. This was due to the complexity of designing a complete IMU in the time that this project was to take place. These eliminations left encoders and a gyro for driving straight, and the gyro and a compass for turning.

After choosing the sensors, the software architecture needed to be developed. The design for the software was broken into three main components. The first was the GUI. This part of the software needed to have the capability of interpreting the intentions of a user sending commands to the robot, and displaying information that was sent from the robot to the user. In both directions, the majority of information was sent through a map on the GUI. The user selects target points for the robot to travel to by clicking on the map. The GUI displays the robot's live position and heading, as well as data points representing the thickness of the ice at certain locations.

The next section of the software system was the communication between the Lifebook inside the robot and an external computer for the user. After researching various techniques, this task was accomplished by a private desktop connection over an ad hoc, computer-to-computer, wireless connection. This provided the display from the Lifebook to be shown on the user's computer, and allowed input from the user's computer to reach the internal computer.

Finally, the major portion of the software was the robot's actual code. This included interpreting the commands sent from the GUI, keeping track of positions from data acquired through sensors, communicating with those sensors, driving control algorithms, and the sending of positional data.

After each system of the robot was designed and fabricated in its own manner, these systems were brought together and tested. Testing phases ranged from initial, stationary testing of the driving, to indoor remote controlled driving, to outdoor remote controlled driving on ice, snow, and water, and finally semi-autonomous driving of the robot through the GUI's map interface. There was also extensive testing of the sensors and GUI interface before they were combined with the drive system and the rest of the robot. This testing led to optimized designs of many of the robot's systems, but it also revealed flaws in the project that could not be fixed before the conclusion of the project.

Overall, most of the project goals were accomplished, and a few were surpassed. On the other hand, a few goals were not met. The largest example of such a shortcoming was the robot's inability to keep track of its position, mainly due to its failure to drive straight. This failure was likely caused by a combination of inconsistencies in the two auger blades that came from manufacturing errors, and a false assumption that the auger blades would not slip or free spin on ice. Attempts were made to change the way the robot determined if it was driving straight, but a solution was not found before the end of the project.

Before a robotic project can be completed, certain social issues must be considered. For example, any safety issues associated with a robot must be identified. For this robot, there are two major safety concerns. The first is that the robot is relatively heavy and bulky for one person to safely carry, and injuries may occur if it is handled improperly. To account for this issue, two carrying handles have been designed on the top of the chassis. This makes it easier for one or two people to carry the robot, and reduces the chances of the robot being dropped. The other safety concern is the blades on the augers. During motion, these blades spin fairly quickly, and could cause injury if they are touched. There is not too much that can be done about this issue, but in order to reduce the potential for injury, the blades were not sharpened after they were delivered. Another social issue that needs to be considered is any potential job displacement issues. This robot is designed to inspect ice autonomously. While there may be a few people whose job would be eliminated if this robot were to become a large-scale solution, the number of people would not be significant. Additionally, the robot still needs an operator, and such an operator would not need to be highly trained, due to the simplicity of the GUI interface. Rather than being displaced by this robot, a person could easily just be reassigned from an inspector to an operator.

This project leaves room for future project groups to pick up the system and make improvements. For example, an ice thickness sensor could be acquired or developed, and interfaced with the platform. Similarly a GPS or other advance localization system could be added to improve the robot's ability to track its motion.

At the conclusion of this project, a semi-autonomous, mobile, robotic platform capable of housing an ice sensor had been created. This robot met the majority of the specifications that were set forth from the beginning of the project. The shortcomings in the project proved to be a learning experience for the students involved. Important information about the engineering design process had been acquired, and this knowledge will be applied to future projects in their engineering careers.

Contents

Table of Figures and Tables

Introduction

Problem Statement

Across the nation, cold weather enthusiasts engage in winter activities occurring on ice. These activities include ice fishing, ice skating, and snowmobiling, which all depend on the ice being safe. Each year there is an average of 235 ice related deaths reported in the US alone. There are several ways that cities attempt to prevent and reduce these incidents; however the most common current method of ice inspection is not effective enough in terms of time, cost, and labor. The most common method is creating a hole in the ice and taking a measurement with a ruler; this takes a long time and only gauges the thickness of that one spot. Using satellites is another method, but is not cost effective. It is apparent that a more efficient and effective method of measuring ice thickness could make ice-borne activities more accessible and safe.

Project Statement

The purpose of this Major Qualifying Project (MQP) was to develop a robot for autonomously measuring ice thickness while moving over the surface of bodies of ice. After some background research on methods of sensing ice thickness, it was found that Ground Penetrating Radar (GPR) is the most effective method for measuring ice thickness. However, GPR is extremely expensive, difficult to use, and far beyond the budget of this project. After talking with the advisor of this project and contacting Geophysical Survey Systems, Inc. (GSSI), the world’s largest manufacturer of GPR systems, it was decided to narrow the focus of the project to developing a mobile platform. The intent is that a future project team could develop a sensor for, or integrate an existing sensor into, this platform.

The robot developed for this project, named Auger Driven Ice Surveyor (ADIS), was designed to fulfill a number of requirements needed to effectively create a system with which an ice sensor could be interfaced. These requirements included:

- Traveling on a natural ice and snow surfaces
- Sensing its location
- Receiving commands from the user

- Preserving itself if it falls through the ice
- Transmitting or storing data

The user interface needs to be capable of:

- Receiving data from the robot
- Creating a gradient map of thicknesses throughout the surveyed area based on thickness and location data
- Displaying this map to a user

Background

Dangers of Thin Ice

Each year, there are an estimated 235 deaths in the United States that occur as a result of people falling through the ice on a frozen lake or pond. According to the Minnesota Department of Natural Resources' (DNR) [2] ice-related fatality statistics there is an average of 6.4 deaths per winter season in the state of Minnesota (based on the past 30 years of data). Over the past five winters (2004/2005 through 2008/2009) 35% of the fatalities were from people traveling on the ice by foot (including ice skates and skis), 41% of the deaths resulted from people falling through the ice on snowmobiles or ATVs, and the remaining 24% are from people in passenger vehicles (cars and trucks) [3].

235 deaths due to ice related accidents may not seem like very many, but there are thousands of undocumented cases where people fell through the ice and did not die. There are specially trained rescue teams who specialize in rescuing victims who have fallen through the ice. Their methods are very effective and as long as someone is present to call for help, they can often save the victims in time. However, as stated in the statistics above, there are times where this is not the case and deaths do occur.

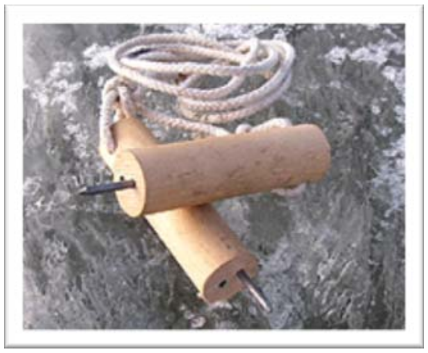
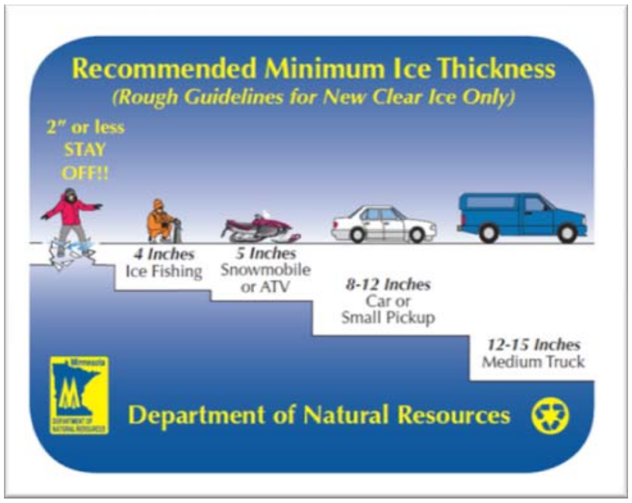


Figure 1: Ice Claws

http://www.dnr.state.mn.us/safety/ice/claws.html

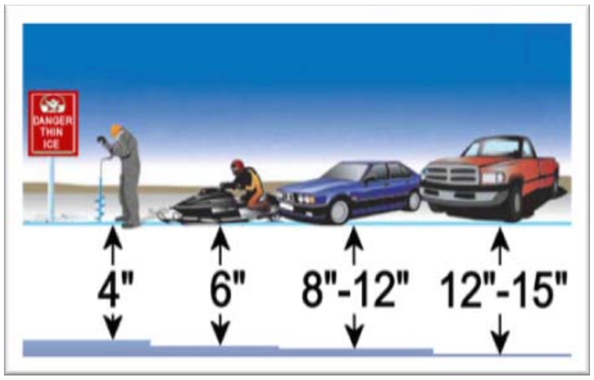
The best way to prevent death due to falling through the ice is to take preventative measures. The most extreme preventative measure would be never going out on the ice to begin with, but this may be too harsh for enthusiasts. Many public bodies of water are checked regularly to ensure that the ice is safe for various activities. Minnesota's DNR, like many other state departments, recommends that anyone participating in any activities on ice first checks the thickness to verify it is strong enough for the desired activities (i.e. ice fishing, ice skating, snowmobiling, etc.). Most cases where someone fell through the ice on a pond or lake occurred in a spot where, for whatever reason, the ice happened to be thinner than the surrounding areas. This is why they also recommend carrying what they call "Ice Claws" which are wooden handles with sharpened nails sticking out that could be used to grip the ice and pull themselves out if they were to fall through. See **Error! Reference source not found.** for an image of what they look like.

The Minnesota DNR provides the rough guidelines shown below in Figure 2 which indicate how thick the ice must be for various activities. North Dakota Game and Fish Department [4] provides a similar graphic on their website shown in Figure 3. The general consensus is that 4 inch thick ice is safe for activities on foot. However, this is for clear ice. Ice that is opaque is considered to be weaker, usually by a factor of 2, which means 8 inches of opaque ice is the minimum that is safe to walk on.


Recommended Minimum Ice Thickness
(Rough Guidelines for New Clear Ice Only)
2" or less STAY OFF!!
4 Inches Ice Fishing
5 Inches Snowmobile or ATV
8-12 Inches Car or Small Pickup
12-15 Inches Medium Truck
Department of Natural Resources

http://www.dnr.state.mn.us/safety/ice/thickness.html

Figure 2: Minnesota Ice Thickness Guidelines



http://gf.nd.gov/education/ice-brochure.html

Figure 3: North Dakota Ice Thickness Guidelines

Methods of Measuring Ice

There are a few ways to effectively measure the thickness of ice. Of all these methods, they can be broken down into two subcategories: Invasive and Noninvasive. Invasive methods of measuring the ice thickness require cutting through the surface. Noninvasive methods of measuring the ice thickness will use a sensor that does not damage the ice in the process. Some examples of noninvasive methods include Ground Penetrating RADAR (GPR), ultrasonic sensors, and acoustic sensors.

Invasive Methods

There are several invasive methods of measuring ice thickness; however they are all similar in theory: Break through the ice and measure the thickness. The most common methods involve creating a hole in the ice and measuring the thickness with a ruler. One traditional way of checking ice thickness is using an ice chisel or ice pick. In this method the user simply stabs the ice with the chisel or pick until they break through to the water and then measure the thickness with a ruler. A more modern way to break through the ice in order to measure the thickness is using an ice auger like the one shown in Figure 4, or simply a cordless drill with a sufficiently long drill bit if only a small hole is needed.

http://www.aworldofrentals.ca/images/equipment/lawn-

Figure 4: Gas Powered Ice Auger

The city of Worcester, MA employs the Department of Public Works to check the ice thickness of all public bodies of water regularly. Their current method of checking the ice thickness is to drill a hole and measure with a ruler that has a hook on the end. While the general consensus is that 4 inches of ice is safe enough for activities like ice fishing and ice skating, the city of Worcester takes extra precautions and does not allow anyone on the ice until it is 6 inches thick. On the ponds that are maintained for ice skating, there needs to be even more ice, at last 8 inches, because the machine used to clear off the snow is approximately 1500lbs.

Invasive methods are the current standard; however there are negative consequences of using them. They require breaking the ice which can lessen the structural integrity of the surrounding ice, making invasive methods more dangerous when done repeatedly in a small area. Additionally, invasive methods only measure at one specific location. Research shows that often people fall through the ice in an area where the ice is thinner than the surrounding areas. By drilling a few holes and measuring thickness, areas where the ice is thinner may not be identified and accidents can still occur. This factor makes invasive methods less dependable.

Noninvasive Methods

Noninvasive methods use sensors to measure the ice thickness without physically altering the ice. Some sensors that could work to accomplish this are: ground penetrating radar (GPR), ultrasonic sensors and acoustic sensors. The most reliable technique appears to be GPR. These systems have been proven to work for ice and snow applications [5]. They can reliably, and accurately, measure the thickness of ice from thicknesses as thin as mere centimeters up to tens of meters. Another feature is that radar can measure the thickness at varying distances from the surface of the ice. CryoSat 2 [6], for example, is a satellite that was launched in April 2010 that is capable of measuring ice thickness of the Polar Regions from orbit with 1 centimeter of accuracy. One negative aspect is that GPR systems can be quite expensive in comparison to other technologies.

Ground Penetrating Radar equipped snowmobiles [7] are widely used in northern Sweden. Several companies that work in car and tire industry do winter tests in Sweden where test tracks are constructed on lakes during the winter. GPR equipped snowmobiles are lighter than the cars and can be used to test the tracks to ensure sufficient thickness to keep cars on the test track, and out of the water.

Ultrasound technology, similar to that used for sonograms, could, in theory, be used for ice measurement, though no off-the-shelf sensors have been found through research. In theory if a frequency could be found that penetrates ice but reflects off water, the travel time of the sound waves could be measured to calculate the thickness.

The final noninvasive method that has been identified as a possibility is to tap the ice with some predetermined force and listen to the sound it makes with an acoustic sensor. The theory is that different thicknesses of ice would resonate at a different pitch when struck, though this may be altered by the quality of the ice as well.

Methodology

Propulsion/Locomotion Design

Selecting the method of locomotion for our robot was a core part of our design process. Auger Driven Ice Surveyor (ADIS) needed to be able to effectively drive over natural ice and snow surfaces. Because of this, the drive method was decided on after extensive background research and comparison of various propulsion techniques.

After the background research was completed, five different types of driving methods were considered. These methods were: track belt drive, propeller driven, wheel drive, and auger drive.

Multi-Track Belt

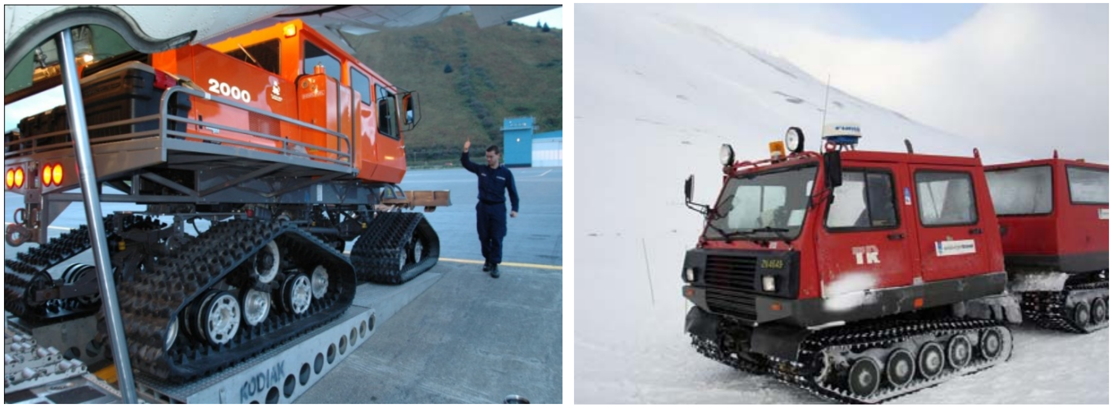


Figure 5: 4-track belt drive Snowcat, 2-track belt drive Snow Truck

Multi-track belt drives are a commonly used drive method for all-terrain vehicles. This drive system is capable of moving on top of snow, ice, pavement, mud, and many others. A majority of large construction vehicles, tanks, and snow vehicles use this system because of its high traction in loose mediums (soft soil, mud, snow, etc.). The reason for this high traction is that a tread has more points of contact with the driving surface, and the treads can be designed to have a high coefficient of friction.

Although treads would offer a high maneuverability on ice and snow, there are a few disadvantages to this drive system. Treaded vehicles generally have a slower maximum speed than other types of drive offer. Additionally, the cost of these types of systems is usually more than that of other systems.

Single-Track Belt

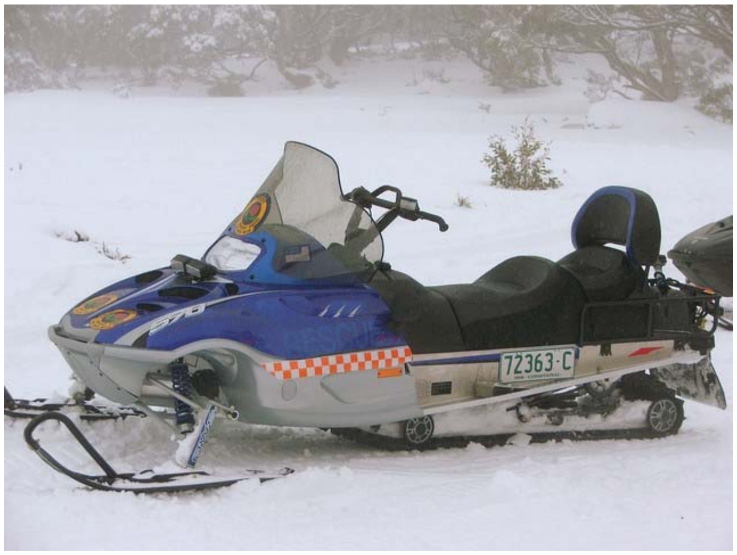


Figure 6: Single-track Snowmobile

Single-track vehicles, such as snowmobiles, use one tread in the rear for propulsion and two skies in the front for steering. While this method of drive generally has a higher speed, it is less agile than multi-track drives when turning. One of the major advantages to this type of drive system is the capability to dynamically distribute weight, causing downhill travel to be more efficient when weight is shifted properly. Additionally, a single-track vehicle can slow down or accelerate more quickly by shifting weight over the tread. While this ability is advantageous for human drivers, developing a software system capable of capitalizing on this advantage would be extremely challenging.

Propeller

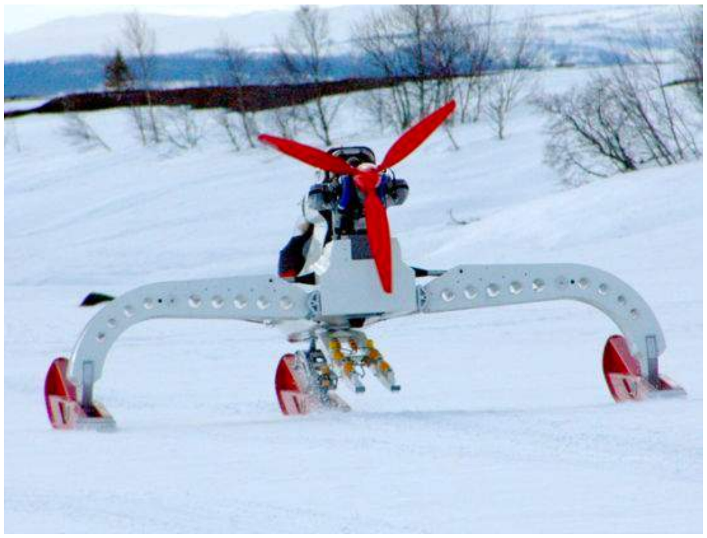


Figure 7: Lotus concept Ice vehicle

A propeller drive system pushes air with rotating blades for thrust and a separate mechanism for steering. One example of a propeller driven vehicle is an airboat. These boats use a large fan mounted above the water, rather than an underwater propeller.

Another example of a propeller driven vehicle is the Lotus Ice Vehicle. It was designed by Lotus CIV to prove the concept of propeller drive in place of more conventional systems on ice for the 2005 Ice Challenger expedition. The Lotus vehicle used skis were used for steering, but also provided a low friction surface to slide across the ice.

The main advantage of this system is that it does not require friction between a surface and the vehicle, making it ideal for snow and ice. Because the system relies on low friction with the surface to maintain movement it can only work efficiently in specific environments. In addition, a large enough fan to propel a robot designed for the purpose of this project could easily become dangerous in less than ideal situations.

Wheels

Figure 8: Snow Tread Wheel

Wheel drive is one of the most commonly used drive systems. A wheel driven system has many advantages in complexity and price. There does not need to be many moving parts, and those parts are very simple when compared to track systems. Furthermore, because of the simplicity and availability of parts, the price becomes much less substantial. The major disadvantage to wheel driven system is that they are less versatile in the medium that can drive on. On ice, wheels have much less traction than other systems, and therefore driving becomes much more difficult.

Screw- Propelled Vehicle (Auger Driven Vehicle)

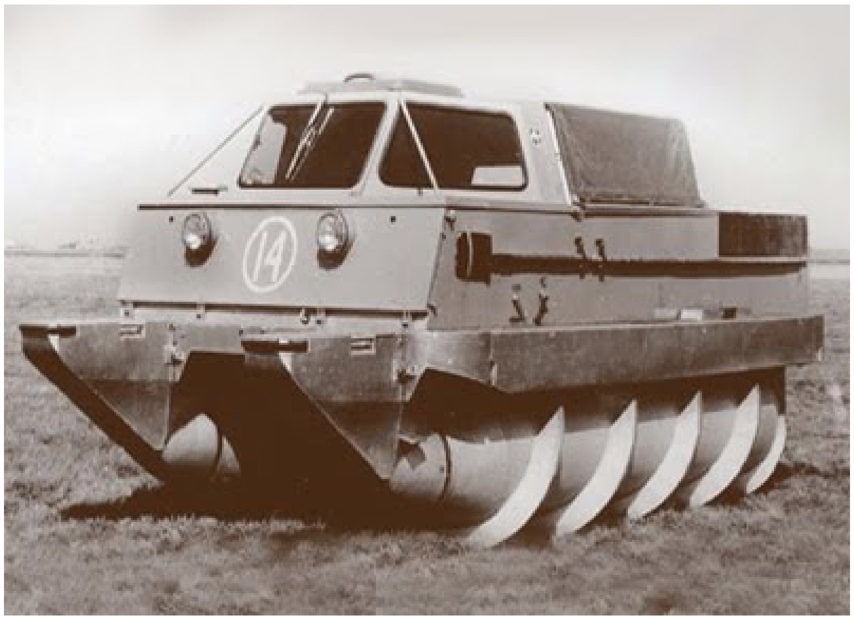


Figure 9: 1970 ZiL-29061 Screw Propelled Vehicle (Soviet Union)

The screw-propelled vehicle in Figure 9 is amphibious; it floats and propels itself on water, ice, snow, mud, and even dry land. The augers underneath the vehicle transform rotational energy into translational energy, propelling it forward. Screw-propelled vehicles move forward by rotating one auger clockwise and the other one counter-clockwise. One auger is threaded clockwise, while the other is counter-clockwise. This vehicle can also drive sideways, in a crab-like motion, by rotating both augers in the same direction. This type of vehicle cannot be used on roads or most other hard surfaces because it can damage both the blades and the surface. This vehicle is relatively hard to control compared to wheel and track belt vehicles. However it has the advantage on ice of minimizing the slip between the surface and augers. The augers can also be designed to provide buoyancy for the vehicle.

Decision

A decision matrix was created to aid in selecting the optimal drive configuration for this robot. The decision was made by considering six categories. The categories were cost, route flexibility, ice mobility, maintainability, part availability, and buoyancy/water mobility. Because the focus of the project is a mobile platform to drive on ice, the ice mobility category was given the highest weight. Additionally, buoyancy and water mobility were given more weight, as self-preservation was one of the major objectives.

Table 1: Decision Table for Propulsion/Locomotion Design

	Cost	Route Flexibility	Ice Mobility	Maintainability	Part Availability	Buoyancy and water mobility	Total
Weight	0.15	0.10	0.35	0.05	0.15	0.20	
Track Belt Drive	3	8	8	4	6	2	5.55
Snow Mobile Drive	4	5	7	5	7	3	5.45
Propeller Drive	6	3	9	8	8	4	6.75
Wheels	9	6	4	8	10	5	6.25
Augers	5	9	9	7	3	10	7.65

From this matrix, it was apparent that an auger drive system would be the best choice for the project. This system's superiority in both water mobility and ice mobility far outweigh its major disadvantage, lack of part availability.

Auger Design

One of the most unique aspects of this project was the use of the auger drive system. Because the design of the augers was essential to the platform's performance, special care was taken to optimize this design. First, calculations were done to determine what auger dimensions would provide sufficient buoyancy to support the weight of the robot. The calculations were completed under the assumption that the auger would consist of a cylindrical midsection with hemispherical ends as shown in Figure 10. It was determined through the calculations that if the robot weighed the target weight of 50 lbs a minimum volume of 0.40 ft^3 would be required from each auger. From this, values of 24 inches for length and 8 inches for diameter were selected. With the selected values, it was calculated that the augers would be 58% submerged if the robot had a relatively even weight distribution. These full calculations can be found in Appendix A: Initial Auger Buoyancy Calculations.

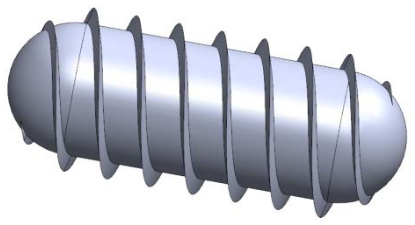


Figure 10: Auger Concept Design

Additional calculations were done to determine what thread pitch for the helix would be optimal in terms of the forces involved and the angular velocity required to achieve the desired drive speed of 5 ft/sec. A pitch of 8 inches was selected yielding an angular velocity of 450 RPM. Refer to Appendix C: Drive Speed Calculations for the full drive speed calculations.

Auger Prototype

To facilitate building a prototype quickly so that fabrication of the final design and testing of sensors and control functions could be worked on simultaneously, the decision was made to fabricate prototype augers that were less than ideal in design but easier to manufacture. The prototype augers used 22 inches of 8 inch diameter Schedule 40 PVC pipe for the main cylinder and ¼ inch diameter PVC rod for the helix. The length was shortened from the original plan of 24 inches due to limitations in the size of parts that could be cut by the laser cutter for the chassis. The helix had the planned 8 inch pitch but was changed from being double threaded, as in the concept design, to having a single thread. Hubs were machined out of 3/8 inch thick polycarbonate and 3/16 inch keyways were cut into the hubs and steel drive shaft. The keyways in the polycarbonate hubs were reinforced with 1/8 inch thick pieces of aluminum. The PVC helix was created by clamping the rod to the pipe and using a heat gun to heat up and bend the rod around the pipe. Once cooled, the rod held its helical shape and was glued in place using Weld-On #16 solvent adhesive.

There was not a taper on the ends of the cylinder (like the hemispherical ends in the initial design) because a pre-fabricated cone or sphere could not be found and it was agreed that for the sake of time, the prototype augers would be constructed faster without it. It was hypothesized that the tapered ends would be necessary for navigating loose snow and designing the prototype without them would also serve to confirm this prediction. Once the prototype was completed and tested in the snow for the first time this hypothesis was confirmed. The prototype augers would not allow the robot to drive in deep or loose snow; however they worked excellent on areas of hard packed snow and ice. Figure 11 shows the design of the completed prototype augers.

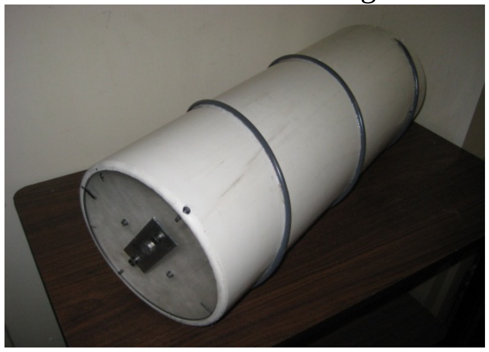


Figure 11: Prototype Auger Design

Auger: Final Design

Once some testing was completed using the prototype augers, the final design of the augers was chosen. Rather than using the originally planned hemispherical ends, the augers would have truncated conical ends. This feature was chosen primarily because the cones would be easier to fold out of sheet metal than hemispheres; however this design also allows the robot to overcome slightly taller obstacles than the hemispheres.

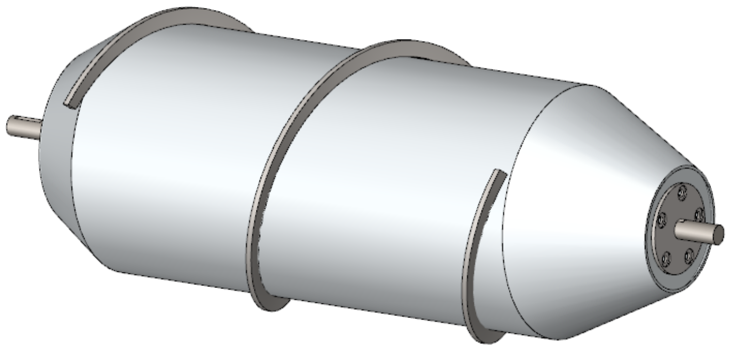


Figure 12: Model of Final Auger Design

The front cone was designed to bring the 8 inch diameter of the cylinder down to a 4 inch diameter over a length of 4 inches. The back cone had less of a taper, only taking the diameter down to 5.5 inches over a length of 2 inches. The auger was designed this way because the gear box would bottom out at a diameter less than 5.5 inches. Additionally, the robot would be driving forward more often than backwards so it made sense to have a forward advantage. Figure 12 shows the Solidworks model of the auger.

Pontoon Material Selection

Choosing the material for the pontoon portion of the auger was an important task since the material properties would directly affect the strength, weight, buoyancy of the robot, and ease of forming and welding the parts together. The target weight of the robot was 50 lbs. It was important that the augers take up as little of those 50 lbs. as possible while still maintaining the necessary strength to endure the abuse of rotating on various surfaces. Aluminum is less dense than steel but would need a greater wall thickness to yield the same strength. Another consideration was cost. The price of aluminum and galvanized steel were much cheaper than stainless steel; however both are harder to weld than stainless steel. Galvanized steel is difficult weld and can emit toxic fumes if not done properly. Aluminum requires a more skilled welder than steel, especially when working with thin walled material. To weigh these different considerations against each other the decision matrix shown in

Table 2 was created.

Table 2: Decision Matrix for Pontoon Material

Material	Price	Strength (US, YS)	Density	Usability	Total
Aluminum	9 ($90)	6 (19 ksi, 9 ksi)	10 (0.10 lb/in^3)	6	5.57
Stainless Steel	3 ($250)	10 (125 ksi, 75 ksi)	4 (0.28lb/in^3)	7	3.20
Galvanized Steel	10 ($70)	10 (125 ksi, 75 ksi)	4 (0.28lb/in^3)	2	4.0
Percentage	25%	15%	35%	25%	100

It was decided that Aluminum would be the best option for the pontoon. The pontoon was made up of seven aluminum components. There was the main 16 inch cylinder, the two inner hubs, the two cones, and the two outer hubs. The exploded view can be seen in Figure 13.

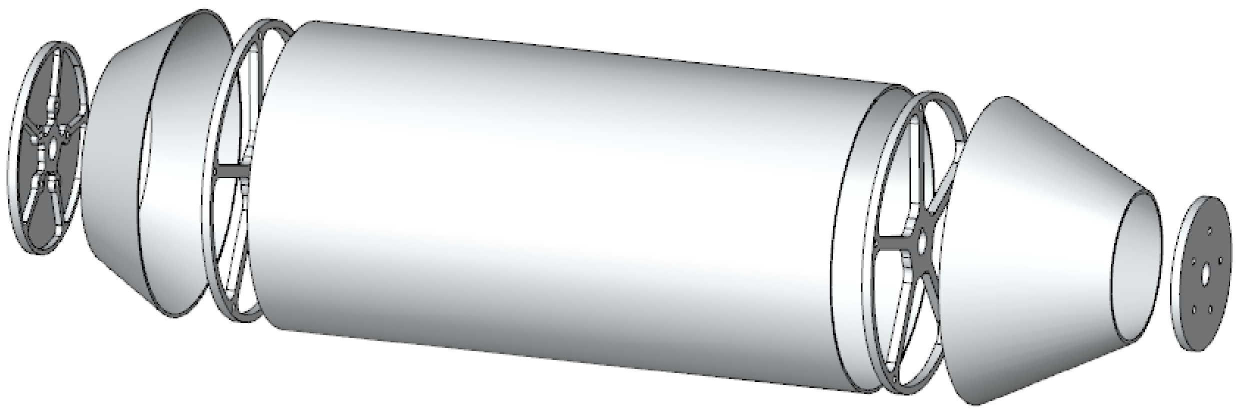


Figure 13: Exploded View of Pontoon

Helix Material Selection

After substantial consideration of materials, cost, lead times, and effort, the decision was made to outsource the fabrication of the two helixes. The cost was fairly significant ($436) however the quality of what would have been produced using on-campus facilities could not be compared to the professionally bent helicoid flighting that was ordered. Although it would be Falcon Industries' problem to form the material, the cost and ease of combining the helix with the pontoon were considered. Since aluminum would have been almost twice as expensive as stainless steel and would wear down easier since it is a softer metal, the decision was made to have the helix manufactured out of 304 stainless steel.

Combining the Components

The primary cylinder of the pontoon portion of the auger was purchased from a metal tubing supplier. The decision to purchase 8 inch diameter, 14 gauge tubing was made because the project partners agreed it would be difficult to roll sheet metal into a close-to-perfect cylinder once, not to mention needing two identical pieces. The tubing was ordered already cut to a length of 16 inches. See Figure 15.

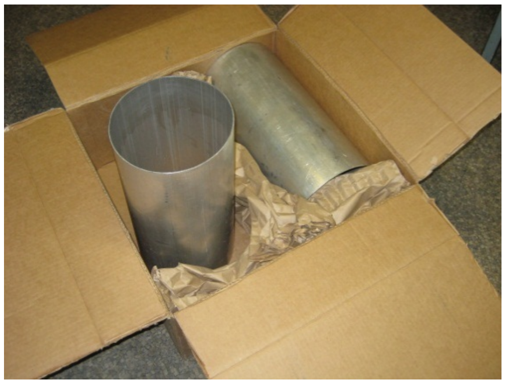


Figure 15: Aluminum Tubing

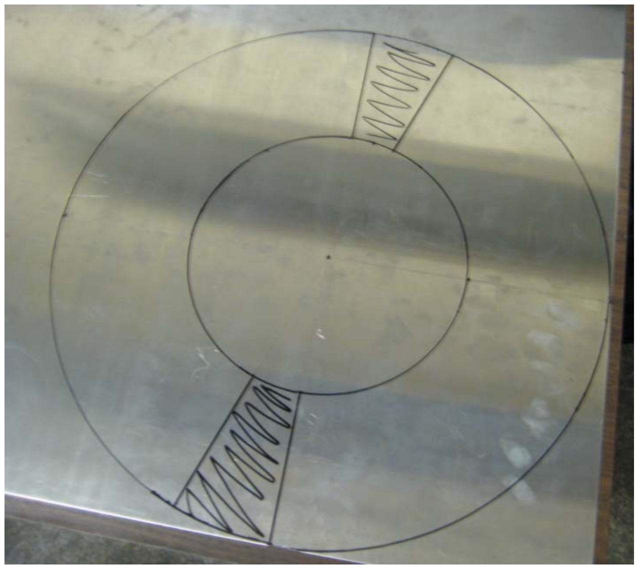


Figure 14: Front Cone Flat Geometry

The front and back cones were formed out of .050 inch thick sheet aluminum. The calculations used to draw the geometry of the cone on the flat sheet metal can be found in Appendix D: Cone Calculations. Figure 14 shows the flat geometry of the front cone and Figure 16 shows what the front cone looked like once it was formed.

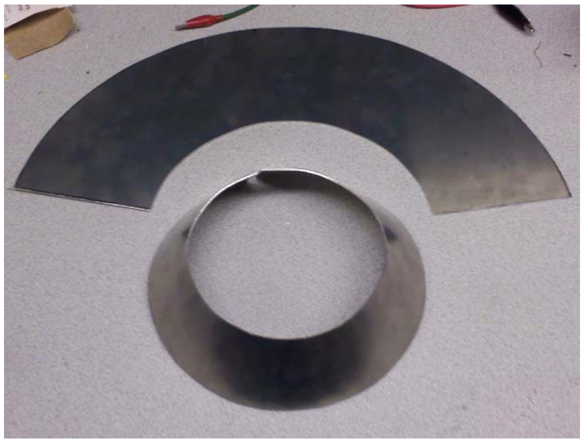


Figure 16: Front Cone Before and After

These components were all welded together by Barnstorm Cycles, a local custom motorcycle shop. Because the pontoon was aluminum and the helix was steel, they could not be welded together. JB Kwik, a steel filled epoxy, was used to attach the helix to the pontoon. Unfortunatly the epoxy and hardener were not mixed properly and the bond did not hold. The JB Kwik was ground off and replaced by Loctite E-20NS, a metal bonding epoxy with a much greater strength than JB Kwik.

Steel hubs were made to transfer the load from the steel drive shaft to the aluminum hub of the auger. The alternative that was considered required keyways. The steel hub method was chosen over keyways for two reasons: the auger would be easier to waterproof without the keyway and the forces would be distributed between five ¼-20 bolts rather than a single keyway. The final auger completely assembled can be seen in Figure 17.

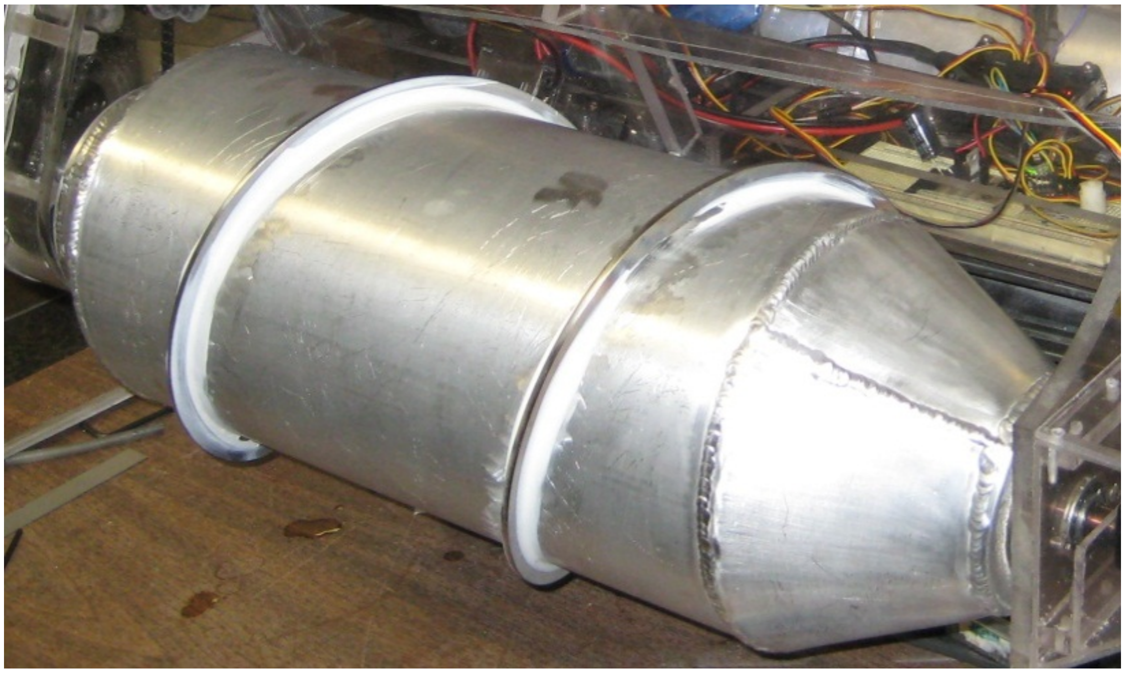


Figure 17: Fully Assembled Auger

Chassis Design

The chassis was an important aspect of this robotic platform as is the chassis with most mobile platforms. The robot would fall apart without a sturdy structure to mount everything to. The chassis was designed with the considerations that the chassis needs to: be compatible with auger drive, have an area designated for an ice sensor payload, and be rigid and capable of withstanding the forces and torques that will be experienced during operation.

Similar to the augers, the chassis went through two iterations. A prototype chassis was created out of less than ideal materials, but completed in an accelerated timeframe. The design was then modified slightly to enhance certain features and remade out of better materials.

Prototype Chassis

The prototype chassis was designed around two constraints. The first was the auger drive; the auger dimensions had already been decided on, so the chassis needed to be designed with the augers in mind. The second design constraint was the size limitation of parts that could be cut using the laser cutter. The cutting area was limited to 24 x 18 inches. A larger part could have been cut out of multiple pieces of material, however it was decided against, in order to maximize strength. Strength of the parts used in the prototype chassis was important because it was constructed out of acrylic. Polycarbonate would have been a more ideal material, however the laser cutter on campus is not capable of cutting polycarbonate and the only other way to cut complex curves on campus was using the CNC mills. Milling the parts would not have been time efficient, therefore acrylic was used.

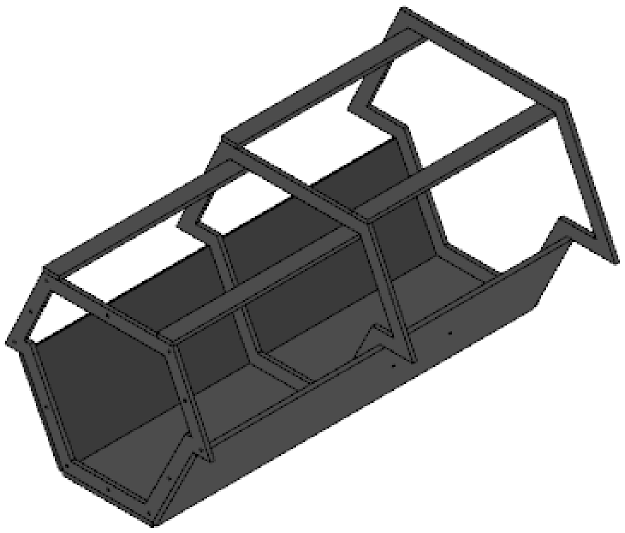


Figure 18: Prototype Chassis

The design consisted of three main ribs in a double trapezoidal shape. There were supports between the tops of the ribs, and three panels that extended the length of the bottom half of the chassis. The ribs had consistent dimensions on the bottom half, however, the top section decreased in size as they moved toward the front. This feature was added primarily for aesthetics; it made the robot look less like a rectangular box. The Solidworks model of the chassis can be seen in Figure 18. The finished prototype, assembled with the front plate, back plate and augers, can be seen in Figure 19.

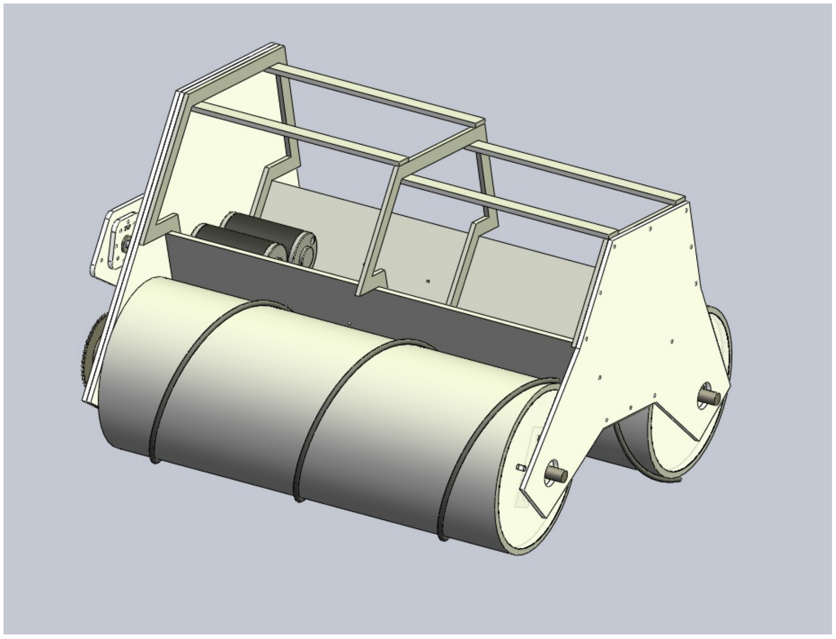


Figure 19: Assembled Prototype

Final Chassis Design

Prototyping the chassis out of acrylic worked out well. However there were a few flaws in the design besides the obvious issue of acrylic being brittle. One of these was that there was no easy way to carry the robot. Another design flaw was that the supports between the tops of the ribs were not ridged enough. The third design flaw was that the chassis would be a sealed compartment once the top was attached and sealed; this would cause the inside to heat up to dangerous temperatures. The last design flaw that was addressed was the general aesthetics of the design.

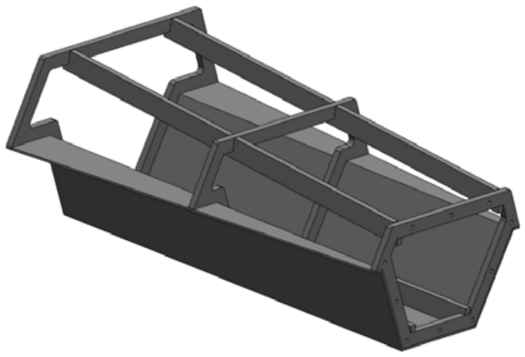


Figure 20: Final Chassis Design

The ribs and their supports were redesigned to use 3/8 inch thick polycarbonate and be solid parts, with notches cut out. Once welded together with the Weld-on solvent, the chassis was much more rigid than prototype. A solid piece of 1/8 inch thick polycarbonate was bent into the shape of the bottom plate and attached, which added additional rigidity. The final chassis design can be seen in Figure 20.

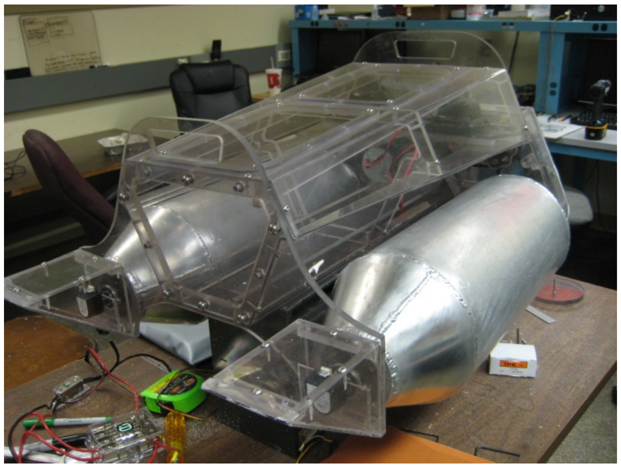


Figure 21: Assembled Final Design

The issue of carrying the robot was addressed by cutting a handle into the front and back plates. The aesthetics of the design were improved by rounding off some edges and adding wedges on the front plate. The fully assembled, final design can be seen in Figure 21. A heat sink was design to be sunken into the back plate where the drive motors attach. This would absorb some of the heat and allow it to dissipate through the external fins into the outside air. The heat sink model can be seen in Figure 22 and the finished part attached to the back plate can be seen in Figure 23. The calculations for heat dissipation can be found in Appendix F: Heat Calculations

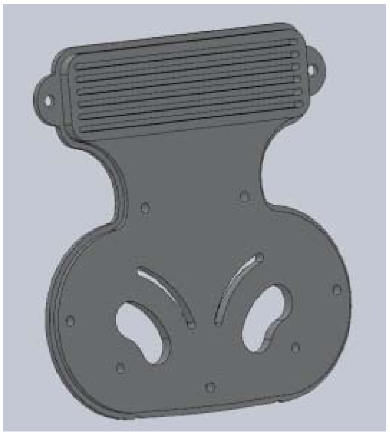


Figure 23: Heat Sink Model

All components of the robot, with the exception of the compass, were placed between the middle and rear ribs of the chassis. This made the robot very back heavy and shifted the center of gravity back a fair distance; however this allowed for the entire front section to be used for carrying a sensor payload. The volume of this area designated to the payload is approximately 9 inches wide, 11 inches deep, and 7 inches high (~.40 ft^3).

Calculations in

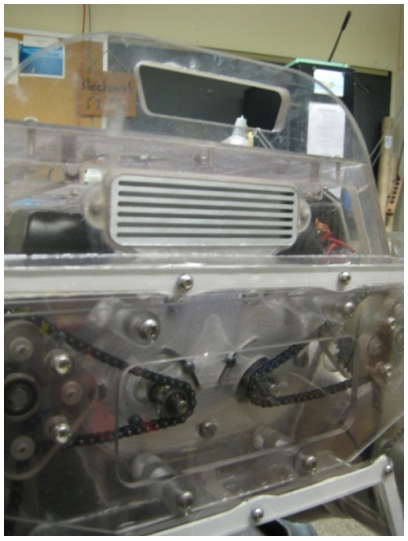


Figure 22: Installed Heat Sink

Appendix B: Final Auger Buoyancy Calculations show that that robot is capable of carrying a payload of up to 21 lbs. without sinking in water (greater payloads could be carried in environments where water will not be encountered).

Drive Train

The drive train for this system is relatively simple. CIM motors were chosen as the drive motors. AmpFlow E-150 motors were considered, however the additional performance over CIMs was not need and therefore not worth the extra expense. Calculations for the maximum torque required can be found in Appendix G: Free Body Diagram (Torque Calculations). To achieve a useable range of speeds for the auger a 12:1 reduction was implemented in two stages using ANSI #25 roller chain and sprockets. The two stages for the reduction were 12:36 and then 12:48. Chains need to be tensioned because they stretch during use; therefore a tensioning method was implemented into the design.

The bearings for the shaft housing the 12- and 32-tooth sprockets were press fit into 3/8 inch thick polycarbonate bearing blocks. The holes for the bearings were offset from center by 1/8 inch. This allows the chain in the second stage of the reduction to be tensioned by rotating the orientation of the bearing blocks. A slot was machined for one of the mounting bolts on the CIM motors so that the motor could be rotated about the other mounting bolt. This tightens the chain in the first stage of the reduction. Refer to Figure 24 for clarification.

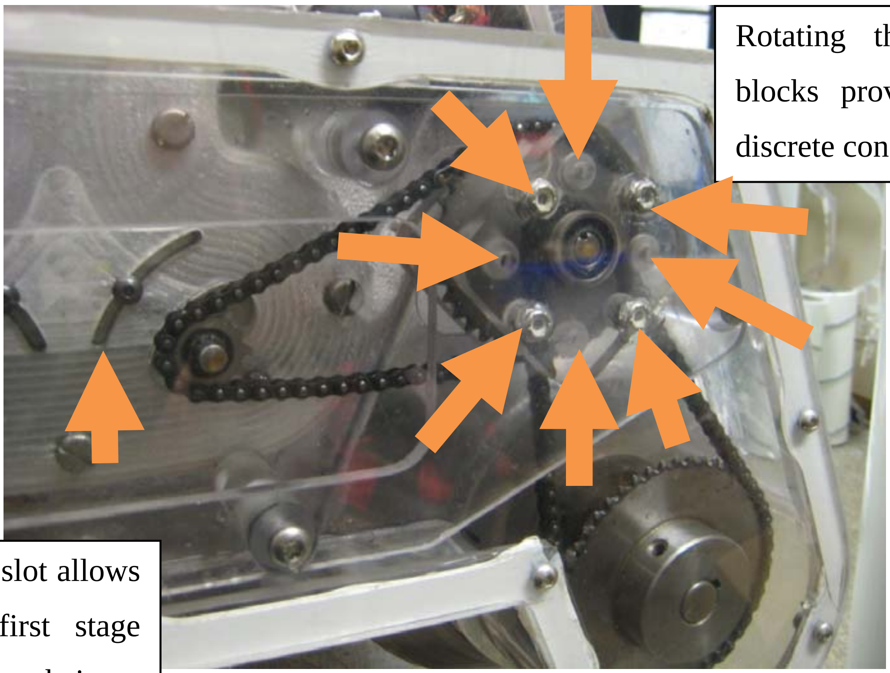

Rotating the bearing blocks provides eight discrete configurations.
This slot allows the first stage of the chain to be tensioned.

Figure 24: Chain Tensioning Explained

Control System Design

The most complex electrical component of a robotic platform is typically the microcontroller. With the obvious mechanical challenges of the ADIS design, minimizing the overhead involved in implementing the microcontroller was a primary concern. In order to achieve this, a controller with minimal low-level interfacing was chosen. Neuron Robotics' DyIO module turns a USB port into a 24 port microcontroller. The DyIO cannot be reprogrammed, but instead functions by interfacing with a computer using a library of standardized communication protocols, which means that some other device would need to handle the logic and processing of the robot.

The devised solution was to include a small computer on board the robot, from which all decisions are made from a Java program. This eliminated two time-consuming parts of robotic programming - uploading code to the controller and optimizing for slow microprocessors. The computing power of a computer is many times that of the typical microcontroller, allowing for much more complex code to be written. Furthermore, no code would ever need to be uploaded to the robot because the un-compiled code was on-board, which even allowed for changes to be made out in the field. Though there were many benefits to using this control scheme, it was far from perfect. Due to the fact that the DyIO was still in development, monthly changes to the firmware and code libraries frequently had an unforeseen impact on the previously functional code. One such issue caused a major change in the program flow of the project, when the asynchronous communication with analog sensors ceased to function as it had before. All asynchronous communication ceased to cause interrupts properly, due to a slight change in the DyIO firmware. Several other minor issues occurred, but were solved with minimal alterations.

Though the DyIO made much of the low-level control much easier, there was still a key piece of controlling the robot that was beyond its capabilities: motor control. The motors would require a supply of power far beyond the capabilities of a microcontroller, so another control option had to be considered. The motor controller needed to provide a constant supply of 20 amps, with a slightly higher peak current. This was the primary concern, though the availability and price where also considered. To meet these standards, two Jaguar MDL-BDC24 motor controllers were installed. The controller featured additional functionality in monitoring encoders and potentiometers internally, but this functionality was neither necessary nor desirable. Though it is capable of taking some processing load off of the main computer, interfacing with the device, which is known to be difficult, is not worth the reward.

Power Supply Design

A number of strict requirements were placed on battery selection. The first, and most important restriction, was cold-temperature performance. Because of the typical environmental conditions the robot would be operating in, i.e. sub-freezing temperatures and unhindered winds on open lakes, the power supply of the robot would need to not only remain unharmed, but also provide ample energy to power normal functionality. The second criterion was high energy density, which would allow high performance without compromising weight. Finally longevity was considered, both in full discharge energy and number of recharge cycles. A battery that lasts longer on the field is a must, especially when the task is performed in a place that it cannot be safely recover from, should the power fail.

Many different types of batteries were considered, from the heavy but reliable lead-acid to the powerful but high-maintenance lithium-ion. After performing a cost-benefit analysis, Lithium Polymer (LiPo) batteries were selected for their superb energy density and wide variety of form-factors. To avoid igniting a LiPo through thermal runaway, several procedures had to be strictly adhered to: charging or discharging could not exceed the specified amperage; do not charge a very cold battery; only proper chargers should be used; and charging must be frequently balanced between individual cells of each pack. To meet our power requirements, the selected battery had to output at least 14 volts at 20 amps for 30 min, with an optimized max current load and recharge time. The selected battery, of which two were used, had 4 LiPo cells to total 14.8V and 5 amp-hours, with a C value of 30. The C value determines the maximum discharge rate, which was 150 amps.

An important consideration for these types of batteries is quality, as we discovered near the end of the project. Within a LiPo battery pack, several cells are connected in series, where each cell is like an independent battery. With packs of poor to moderate quality, there are frequently instances where a single cell will cease to function properly. Continuing to use the battery as it was intended can becomes impossible, and replacing the cell can be risky. One of the batteries used in ADIS ceased to function correctly in this way. Fortunately the robot was still functional with a single battery, without noticeable side effects besides a shorter running time.

Sensor Implementation

Sensors used on the platform would need to gather accurate information about the thickness of ice, and the location of such a reading relative to some starting point. To maintain an accurate representation of this information, there would have to be several sensor subsystems that collect specific information, such as the current heading or localized position. To accomplish these tasks, the following sensors were originally chosen: a GPS to give absolute position; a gyro and accelerometers to give heading and motion information; a GPR to measure ice thickness; encoders to measure distance; and a series of ultrasonic sensors to identify obstacles. However, several flaws were found in these choices as more research and discussion followed.

A GPS system gives the global coordinates of a user at any time they have sufficient line-of-sight to the system of satellites that maintain it. To provide a global location for the robot, this type of system was considered early in the design process. However, it was soon determined that such a system wasn't as beneficial as originally perceived, due to a number of considerations about the goals of the project. In order for a GPS receiver to initialize, a significant (and often inconsistent) amount of time is required to acquire satellite information; furthermore, that information would only be accurate to within a few meters with the most affordable GPS systems, and this accuracy is highly dependent on circumstances such as time of day and weather. These conclusions led to the exclusion of GPS since our workspace will only cover a small area, within which the accuracy provided would be insubstantial even for corrections to other methods of measuring position.

Very quickly it was realized, due to the nature of frozen ponds and lakes, that there would be no obstacles or barriers to get in the way of the robots planned path, as long as that path remained on the ice. This allowed the ultrasonic sensors to be excluded, which would also make water-proofing much easier since they would have had to been exposed to the air. However, this meant that more dependence would be put on other systems to accurately track position.

An Inertial Measurement Unit (IMU), which contained both accelerometers and a gyro, was selected to fill in the gaps in sensory information. However, it wasn't until raw data from the accelerometers was collected that the complexity of such a system became apparent. Though they could give accurate data about accelerations in any direction, the meaning was highly dependent on knowing the direction of gravity at all times. A motionless IMU can be used to determine the direction of gravity, but without at least two gyros then that vector cannot be determined while in motion. Because of this, the accelerometers became useless since there was no way to differentiate between acceleration from gravity, which had x and y components when tipped slightly, and the acceleration of the platform. The gyro thus became the primary method of determining heading, but this also became a problem, as was discovered with the amount of drift from that sensor. To combat this drift, a compass was acquired. This would allow for periodic checks to be made, to determine that the perceived heading was still accurate.

One obstacle that hindered the project was implementing the compass module. The first compass purchased turned out to be incompatible with the communication protocols used on the DyIO, and couldn't be interfaced with even after extensive troubleshooting. A second compass was purchased, this time with analog output. However, two more issues became apparent after actual data was collected. The first issue was in interpreting the data from the compass, which required a function tailored for each unique compass. A trial-and-error method was devised to tune it, by making 10 degree changes in the position and recording the offset to create a function that represented the error. This provided the most meaningful feedback, and resulted in consistent and accurate interpretation of the data. The second issue was realized shortly after, when the compass was installed in the platform chassis: the magnetic fields of the motors influenced the magnometers in the compass significantly. After some research into the behavior of magnetic fields, a position in the chassis was determined as the least-influenced. However, other project goals took priority near the deadline, and the compass was never properly tuned. The same trial-and-error method can be used in the future to accurately tune the compass in its new magnetically influenced environment.

After consulting Geophysical Survey Systems, Inc. (GSSI) about GPR technology, it was discovered that using such a complicated sensor wouldn't be as easy as reading an analog value. While certainly capable of measuring the ice, the data gathered by a GPR is extremely complicated and can’t be simply interpreted by a computer system as a thickness. This was not ideal, though it would suffice to correlate the position with each reading taken. Luckily, GSSI had an analog sensor on hand that they had researched before; it was an analog ice radar once used in Russia. While unfit for their purposes, mainly from a sales point of view, it seemed to fulfill the requirements of this project nicely. They expressed an interest in aiding with the project by lending the sensor. Unfortunately, due to the sheer value of either of these sensors, it was deemed that an accurate weight and volume substitute for either sensor would suffice in the meantime, under the assumption that the actual sensor could be used with minimal trouble.

Software Design

Human-Computer Interface

Initially, the method of providing instructions to the robot was going to be in the form of a list of GPS coordinates that represented the constraints of the pond that the robot was to be inspecting. The intent was that the robot would then autonomously cover the area within these coordinates, thus surveying the entire pond.

Once GPS was removed from the scope of the project, this idea evolved into the user inputting coordinates, relative to the robots current position, on a map of the pond being inspected. The robot would then travel to each coordinate it was given, taking ice-thickness readings along the way or at each point, depending on the sensor the robot would be using.

To implement this idea, the user would need a GUI which would be capable of displaying a map and accepting mouse-clicks on the map, to represent target locations and the starting location of the robot. It would also be advantageous if this GUI would display the current location of the robot as it traveled, keep track of and display the target locations given, and have a way to represent the path that the robot had traveled. These were the features initially aimed for in the GUI.

The first iteration of the GUI was a grid of buttons, with each button representing one square foot. When this grid was created, it would set the robot’s initial position to the center of the grid, and color the button there green. When a button was pressed, the button would turn red, the distance between the robot's current position and the target position would be calculated, and a simulation of the robot driving to that point would be displayed. For this, small increments would be added to the robots current position, in the direction of the target point. When the robot reached a new square foot, the button representing the previous square foot would be colored black, and the new button would be colored green. This simulation kept track of the robot's current location, and displayed the path that the robot had traveled along.

There were a few problems with this iteration of the GUI. First, the positions given to the robot were only distances from its starting location. This meant that a person using this GUI would have to know the dimensions of the lake, and plot a course based on these numbers, not based on the actual shape of the lake. Additionally, the resolution of the grid of buttons was very limited. To represent a larger area or a higher resolution, more buttons needed to be created, but this took a heavy toll on the processor. Finally, only one target point could be added at a time, meaning the user would have to wait for the robot to get to its location before giving it a new location.

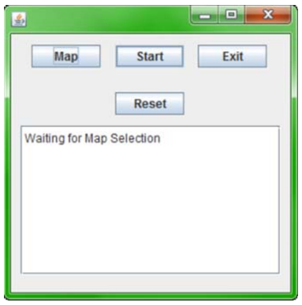
The next iteration of the GUI used an image of a map as the main display. By clicking on the map, a user could select the starting location of the robot and set target points, which were stored in an array of positions. To represent the situation, colored shapes were drawn onto the map. A yellow square represented the starting location of the robot. A red square represented one of the target positions in the list of targets. A green square represented the robot. The GUI would report locations with x and y coordinates determined by the pixels on the image that were clicked. This meant that, once the scale of the map is known, a distance can be applied to a pixel, and the edges of the lake on the map image should accurately represent the edges of the real lake.

Once this was implemented and tested, small features were added to the mapping. As the robot moved across the map, lines were drawn from the robot's current instantaneous location to its last location, therefore creating a visual, persistent path. As another feature, the robot was displayed as an isosceles triangle rather than a square, with the front pointing in the direction the robot was facing.

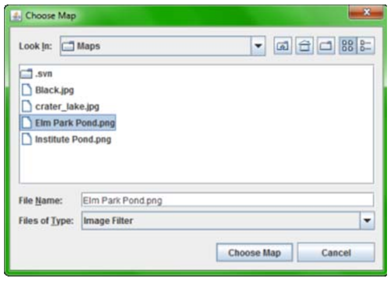
After the mapping was complete, functionality was added to the GUI to give it a broad array of uses. A textual display was added to give the user information about what the program was doing at any given time. For example, at the beginning of the program, the display states “Please select starting location”, and later in the program once the robot has finished moving through the array of target locations, the display states “Waiting for new locations.”

Another feature that was added to the GUI was the ability to select a map from anywhere on the computer. Previously, the map to be used was hard-coded. This new feature allowed the user to select an image at run-time. A third new functionality was the ability to pre-load target locations before the robot starts moving. A user can input as many targets as they want, and the robot will not begin moving until the “Start” button is pressed. To better see the orientation of the robot, a larger triangle had been added to the side of the map. An exit button was also added to the GUI, so that the user can safely exit the program at any time.

The next round of improvements to the GUI resulted in a maximum speed controller, and a mock ice thickness display. The maximum speed controller is a slider that scales down the maximum speed of the robot if necessary. To display ice thickness, the previous method of marking the path of the robot was changed. Instead of black lines, the GUI drew squares with a color that represents the thickness of the ice at that location. A black square represented ice that was positively unsafe. A red square represented ice that was dangerous. A yellow square represented ice that was questionably safe, and a green square represented safe ice. These ranges were set in the IceReading class, and could easily be adjusted. At the conclusion of this project, the ice thicknesses used to display this functionality of the GUI were randomly generated, as the robot did not have a sensor to read the thickness of ice.



Waiting for Map Selection



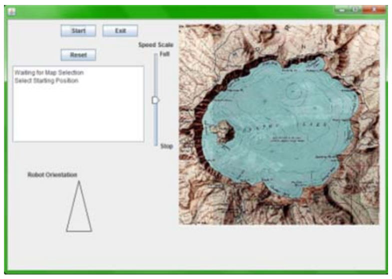
Look In: Maps

.svn
Black.jpg
crater_lake.jpg
Elm Park Pond.png
Institute Pond.png

File Name: Elm Park Pond.png
Files of Type: Image Filter

Figure 25: (Left) GUI Prior to Map Selection, (Right) Map Selection GUI

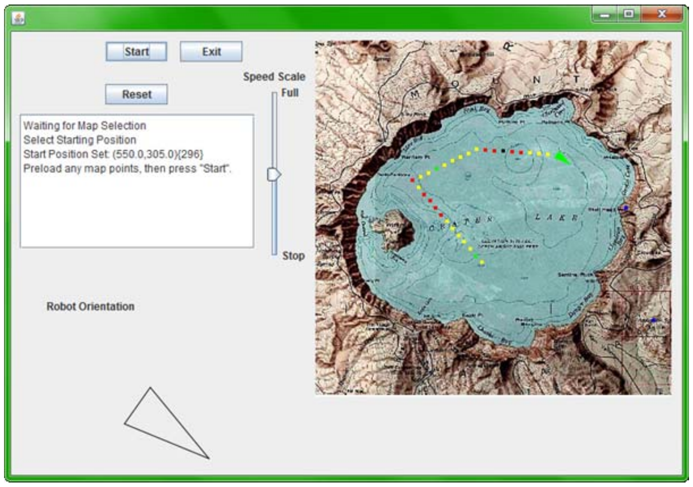
Figure 25(Left) shows the GUI before a map is selected. When the “Map” button is pressed, a file chooser pops up, in which the user selects a map, as shown in Figure 25(Right).



Waiting for Map Selection
Select Starting Position

Figure 26: The GUI after a Map Has Been Selected

Once a map is selected, the GUI inserts the map, as well as making the other components of the GUI visible. Figure 26 shows this updated GUI. Figure 27 shows The GUI while the robot has been moving. All of the functionality can be seen here.



Waiting for Map Selection
Select Starting Position
Start Position Set: (550.0,305.0)(296)
Preload any map points, then press "Start".

Figure 27: The GUI as the Robot is Moving

External Computer-Internal Computer Interface

To control the robot during its inspection, a Fujitsu U-Series Lifebook was placed inside the chassis. This computer ran the program that gave instructions to the DyIO, and displayed the GUI. To access the GUI, a system needed to be developed that allowed a user to remotely interface with this onboard computer.

The initial idea was to use a Remote Desktop connection from a laptop on shore. This would give the user complete control of the onboard computer, and the ability to start the program, give the robot commands, and see the information that the robot outputs.

To use Remote Desktop, both computers must be on the same wireless network. To make this possible on the field, a wireless ad-hoc (computer to computer) network was attempted. Unfortunately, it was found that it is not possible to establish a standard Remote Desktop through an ad-hoc connection. To circumvent this issue, software that performs similarly to Remote Desktop, which can function over ad-hoc was found. The program used was called UltraVNC.

Robot Code

Through the development of this project, a complex software system has been created to communicate with the robot. This section will describe the purpose and functionality of each Java class that is not involved in the GUI. Figure 28 shows a model of the code.

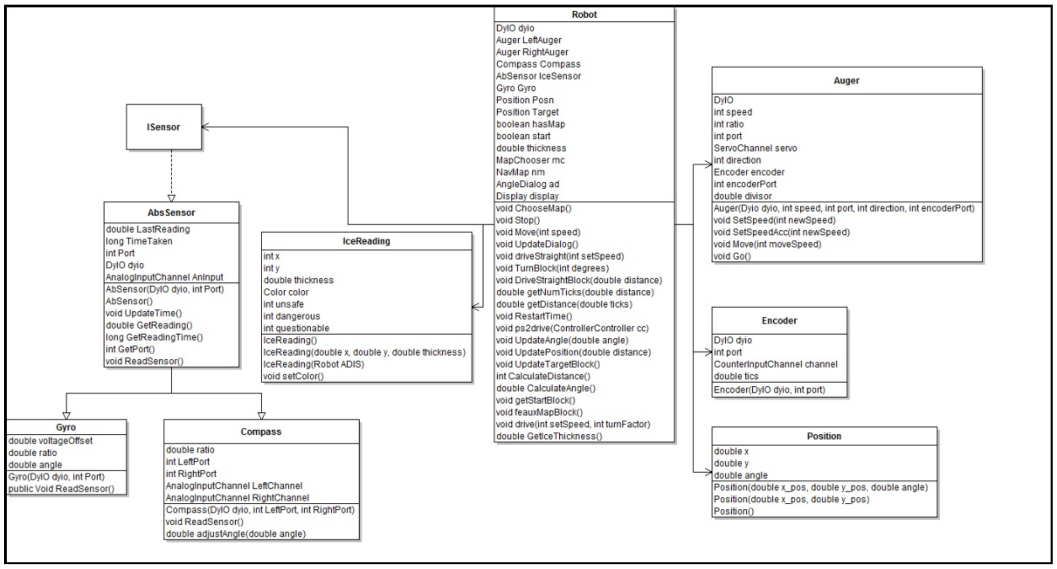


Figure 28: Model of the Code

Robot

Robot is the main class of the software. This class contains functions for communicating with each of the sensors, controlling the motors, and keeping track of the robot's position. It also contains all of the GUI elements, and a *DyIO*, for communication purposes.

The fields of *Robot* are: A *DyIO*, *dyio*, two *Augers*, *LeftAuger* and *RightAuger*, a *Compass*, an *AbSensor* called *IceSensor (*as a placeholder), a *Gyro*, two *Positions*, one called *Posn* that represents the current position of the robot, and the other called *Target* that represents the current position that the robot is moving to, and two *Booleans*, *hasMap* and *start*, used for program flow, and finally, the various GUI components discussed in the previous section.

The constructor for *Robot* contains a lot of functionality. It is in this constructor that each sensor gets its port on the DyIO. The *Augers* are created, with the port for their *Encoders,* and their direction, which represents which direction of the motor would drive that *Auger* forward. Parts of the GUI are also initialized.

The first function in *Robot* is *ChooseMap*. This function opens a *MapChooser*, and then adds the selected map to the *Robot*'s *Display*. The function also creates a new *NavMap* with the selected map.

The next function is *Stop*, which simply sets the speed of both augers to 127, stopping their rotation.

Next is *Move,* which takes in an *int* called speed. This function calls the *Auger's* function '*Go'* for both augers, which results in both augers moving at the given speed, in the correct direction for that *Auger*.

UpdateDialog is a GUI function. This function sets the angle of the *Robot*'s *AngleDialog*'s *Triangle* to the angle of the *Robot'*s *Position*. Then the function redraws the *Display,* so that any changes are shown.

The *driveStraight* function takes in an *int* called *setSpeed*. The function calls the *drive* function with the given speed, and a turn factor of zero. This function should be called from within a loop.

The next function is *TurnBlock*, which takes in an *int* called degrees. This function uses PID control to turn the robot the given number of degrees, in a clockwise direction. The function uses the Gyro to turn, and after it has turned, it waits, and then verifies with the Compass that it has turned the correct number of degrees. If it hasn't, it takes the average of what the two sensors are saying, and uses that as the current heading for the robot.

Next is the function *getNumTicks*. This function takes in a *double* called distance, and returns the number of encoder ticks the augers should turn in order to travel that distance. The function *getDistance* does the inverse operation. It returns the distance that would be traveled in the given number of ticks.

RestartTime is a function that sets the '*TimeTaken'* field of each sensor to the current time. The reason for this is so that if a sensor hasn't been used for a relatively long time, this function can be called to avoid an error with the next reading of the sensor.

The function *ps2drive* is designed to allow the user to control the robot with a Play Station 2 controller. This function is mainly used for testing, and does not sync with the normal operation of the robot.

UpdateAngle takes in a *double* called *angle*. The function adds the given angle to the robot's current angle. The function then checks to see if the robot's angle is outside the range of 1 to 360 degrees, and corrects it if it is not.

UpdatePosition takes in a *double* called distance. The function calculates the *Robot*'s new position based off the *Robot*'s previous position, assuming it traveled the given distance at its current angle. The function then adds the new position to the list of positions the robot has occupied, and updates the display to show the robot at its new position.

The next function is *UpdateTargetBlock*. If the list of target positions for the *Robot* is empty, this function waits for a position on the map to be clicked, then adds the clicked position to the list. Otherwise, the function updates the *Robot*'s current target to the next position in the list.

The function *CaluculateDistance* calculates the distance, in inches, between the *Robot*'s target position, and its current position, then returns this value. Similarly, the function *CalculateAngle* uses the *Robot*'s current heading, and returns the number of degrees that the *Robot* has to turn in order to be facing the target position.

The *getStartBlock* function, when called, waits until the user clicks on the map. The function then uses the clicked location as the *Robot*'s initial position, and uses the compass' heading as the *Robot*'s heading.

mapBlock is the main function used for the Robot's functionality. This function calls *getStartBlock, UpdateTargetBlock, TurnBlock, DriveStraightBlock*, *CalculateAngle*, and *CalculateDistance*, while updating the display accordingly. This results in the function first waiting for the *Robot*'s initial position, then accepting new target positions, orienting the robot towards the next target position, driving the correct distance to this position, then repeating the process for the next target position.

The next function, *feauxMapBlock*, is a function mainly for testing. This function operates similarly to *mapBlock*, except that instead of actually moving the *Robot*, the function feeds artificial data to the *Robot* object to simulate movement. This allows testing of the GUI mapping features without requiring that the robot actually be driving around.

The drive function takes in an int called *setSpeed*, and an int called *turnFactor*. The function sets the robot to drive at the given speed, but uses the *turnFactor* variable to set the two augers at different speeds. A higher positive turn factor causes the robot to turn more sharply in a clockwise direction. This function needs to be called in a loop to work properly, as it only sets the motors to go their given speeds once, and then does not wait any amount of time.

The final method in the *Robot* class is *GetIceThickness*. Currently, this function only generates a false ice-thickness reading, so that the functionality of the GUI can be observed. In future work on the project, once an actual ice sensor is acquired, this function will be re-written to read a measurement from the sensor.

Auger

The class *Auger* represents an auger and a motor on the robot. The fields of an *Auger* are: a *DyIO* called *dyio*, an *int*, *speed*, that represents the PWM speed that this *Auger* is currently set to drive, an *int*, *port*, which is the number of the port on the *DyIO* that the motor for this *Auger* is plugged into, a *ServoChannel* called *servo* which is the channel used by the motor, an *int* called *direction* which represents the direction the motor for this *Auger* must turn to drive the Auger forward, an *Encoder* called *encoder*, an *int*, *encoderPort*, which is the port on the DyIO the *Auger*'s encoder uses, a *double* called *divisor*, which is a number that scales down the maximum speed of this *Auger*.

The constructor for an *Auger* takes in values for the *Auger*'s *dyio*, *speed*, *port*, *direction*, and *encoderPort*. With the *port* variable, the constructor creates a new *ServoChannel* on the DyIO. Similarly, the constructor creates an *Encoder* object with the given *encoderPort*.

An *Auger* has two functions for changing its *speed* field, *SetSpeed*, and *SetSpeedAcc*. *SetSpeed* instantly changes the *Auger*'s speed to the given speed. *SetSpeedAcc* makes a small change towards the given new speed from the *Auger*'s previous speed. This has the effect of limiting the acceleration that the *Auger* can have. In order for this function to fully change the *Auger*'s speed to the given speed, it must be called repeatedly in a loop.

An *Auger* has a function called *Move* that takes in an int called *moveSpeed*. This function calls *SetSpeed* with the given speed, and then calls the function *Go*. The function Go performs multiple duties. First, the function limits the *Auger*'s speed field to a number between 0 and 255. Next, the function performs an operation on the *Auger*'s speed with the *Auger*'s *divisor*, scaling down the maximum speed. For example, a *divisor* of 2 would result in cutting the maximum speed of the *Auger* in half, both forwards and backwards. Finally, the function *Go* sets the *ServoChannel* of the *Auger* to actually move at this final PWM value.

Encoder

The *Encoder* class deals with all of the code necessary for interfacing with a shaft encoder. The fields of an encoder are: A *DyIO*, *dyio*, an *int* called *port* that represents the port number of this *encoder*, and a *ConterInputChannel* called channel that is the channel for the encoder.

The only method in the *Encoder* class is the constructor for the class. The constructor takes in a *DyIO* and an *int port*. The function creates the channel for the *Encoder* from this port. The code for reading ticks from an *Encoder* comes inherently in the code for a *CounterInputChannel*, given by the *nrdk*.

Position

A *Position* represents a set of x and y coordinates, as well as a heading. Each of these values is stored in a *double*. Three constructors exist for a *position*. The first takes in *doubles* to represent the three fields of the class. The second constructor takes in only the coordinates, and sets the heading angle to zero. Finally, a default constructor sets all three values to zero.

IceReading

The *IceReading* class is used for storing an ice thickness reading, as well as for displaying this reading in the GUI. An *IceReading* contains: A *Position* to represent where the reading was taken, a *double* called *thickness* which is the thickness of the reading, a *Color* called *color*, used for displaying the *IceReading*, and three *ints* that define the upper bounds of “unsafe”, “dangerous”, and “questionable” ice thickness ranges.

The default constructor for an *IceReading* sets all the numerical values of the object to zero, and sets the color to black. The second constructor takes in a double for an x coordinate, y coordinate, and ice thickness. The constructor then sets then calls *setColor* to determine the proper *color* for this object. A third constructor takes in *Robot*, and uses the *Position* of the *Robot* to set the *Position* of this object. The constructor then calls *setColor*.

The method *setColor* is used to determine the color of an *IceReading*, based on the thickness. The method compares the thickness to the three upper bounds of ice thickness, and then sets the color to black is the thickness is less than “unsafe”, red if it is less than “dangerous”, yellow if it is less than “questionable”, and green if it is thicker than the final range.

AbSensor

An *AbSensor* is an abstract class for the various sensors that *Robot* interfaces with. In this project, *Gyros* and *Compasses* are examples of classes that extend this class. The intention is that the ice sensor will also extend the *AbSensor* Class.

The fields of an *AbSensor* include: A *double* called *LastReading*, which is the reading that the sensor most recently returned to the *Robot*, a *long* called *TimeTaken*, which is the time that the *LastReading* was taken, an *int, Port*, for used with the DyIO, a *DyIO* for communication, and an *AnalogInputChannel* called *AnInput*, which is the channel for the sensor. The constructor for an *AbSensor* takes in a *DyIO* and a *port*, and creates the *AnalogInputChannel* on that port of the DyIO.

The method *UpdateTime* sets the *TimeTaken* field of an *AbSensor* to the current system time, in milliseconds. The class also has an abstract method called *ReadSensor*. This sensor is written by each class that extends *AbSensor*.

Gyro

A *Gyro* extends *Absensor*, and represents the code needed to interface with a gyroscope. This class contains three *doubles*, in addition to those inherited from the parent class. The field *voltageOffset* represents the number of volts that must be subtracted from the voltage read from the sensor, to center the range around zero volts. Next, *ratio* represents the ratio of angular velocity to voltage. Finally, *angle* represents the current angle that the gyro is facing, based off of a series of given angle changes.

The *ReadSensor* method of the *Gyro* class first looks at the *LastReading* of the sensor, and converts this voltage into an angular velocity. Next, the method compares the current time to the *TimeTaken*. This change in time is used with the angular velocity to determine the number of the degrees that have been traversed since the last reading was taken. The method next adds this number of degrees to the *Gyro*'s *angle*. Finally, the method reads the voltage from the sensor and the current time, and updates the appropriate fields.

Compass

A *Compass* is another class that extends *AbSensor*. The Compass used for this project uses two perpendicular magnetometers, so the *Compass* class was designed to incorporate this. The class contains: A *double*, *ratio*, two *ints* for the left and right port, and two *AnalogInputChannels* for the two Channels to the DyIO. The constructor for this class takes in a *DyIO*, and two *ints* for the ports, and then creates the channels using the ports.

The *ReadSensor* method of this class takes the voltages from both channels, and then takes the arctangent of the two values. This gives an angle that needs to be adjusted slightly, so the function calls the *adjustAngle* function. The *adjustAngle* function takes in an angle, and returns an angle based on a lookup table acquired by experimentation.

Results and Analysis

Testing

As the various systems of this project were designed, manufactured, coded, and brought together, extensive testing needed to be done in order to verify the success of the project as a whole. The tests that were conducted were chosen in accordance with the goals of each system. Testing locations included the laboratory, flat regions of snow on campus, and ice and water on Elm Park Pond.

Auger

The augers needed to be tested for two main aspects: The ability to provide reliable propulsion on ice, and the ability to maintain their structural integrity while providing this propulsion.

To test both of these aspects, the augers were first driven on carpet in the lab with a remote control system. After this was shown to be possible, the system was taken outside and tested similarly on snow. Finally, these driving tests were performed on the surface of a frozen pond.

While driving the robot on the carpet, it was observed that the robot has some ability to turn, but would very easily crab drive, as was expected. This crab driving was much faster than the forward motion of the robot.

When the robot was driven on snow, it was found that the turning ability was vastly different than it had been in the previous tests. On snow, the robot turned very easily, and had a greatly improved forward velocity. Additionally, it did not crab drive at all. The robot retained this style of motion through the ice testing.

Water testing

In water, this robot needed to be able to remain buoyant, be mobile, and stay water tight. For these tests, an open section of Elm Park Pond was used. First, the robot was gently placed into the water, and observed for signs of leaks. After it became apparent that the robot would stay afloat, the remote control system was used to drive the robot across a small segment of open water. Recovery over an ice edge from the water was also attempted at this time.

During the initial buoyancy tests, the robot stayed afloat, but was not balanced correctly. The chassis was back-heavy, so the robot tipped backwards a small amount. Prior to the water propulsion testing, a ten pound weight was added to the front to ballast the robot. During this test, the robot did show the capability to maneuver in aquatic conditions, as well as the potential to climb over the edge of the ice. Unfortunately, this testing needed to be cut short, as the augers caused a significant amount of splashing, and the robot was not sealed on the top.

Battery Life

One of the major concerns for the robot's batteries was their ability to run the system for an extended period of time. Fortunately, this concern was easily addressed by using the battery for extended periods of time during other tests, and monitoring the remaining voltage after the tests were completed. Through this testing, it was found that the batteries had enough charge to run the robot for at least 30 minutes.

Graphical User Interface (GUI)

The most important tasks that the GUI needed to perform were giving commands to the robot through the map, and displaying information from the robot on the map for the user. To first test both of these functionalities, the feuxMapBlock was created. This was a function that read input from the user, and generated a model of what the robot would be doing if it had received that input. After bugs with the GUI were worked out using this function, the real MapBlock function was used, sending signals to the actual robot. Testing the communication capability of the GUI was finished in this manner.

Computer to Computer Network

To communicate wirelessly with the robot, the user-end computer needed to reliably connect with the robot-end computer, and send information in a timely manner. To test this ability, an UltraVNC server was established on the Lifebook. A separate computer established a connection with this server through the WPI network. The robot was then controlled through this server in the lab. After this method was proven to work, the robot-end computer connected to the server through an ad hoc network. The robot was then driven outside, in realistic conditions with this connection.

With this testing, it was found that the robot can be successfully controlled through this type of connection. Commands are sent to the robot nearly instantly. The display has a small amount of lag for showing the user information, but none of the information is ever lost over the connection. After the success of this system was verified, further testing of the robot was done through a wired connection, for ease of coding.

Robot Navigation

For this project, the robot platform was required to drive straight for a given distance, and turn a specified number of degrees, allowing the robot to easily keep track of its location. To test the driving straight capability, the robot was given a long distance to travel. If the robot veered, corrections to the control system would be made. Then, the distance that was traveled was measured and compared to the distance given.

To test the turning ability of the robot, it was instructed to turn ninety degrees in one direction, then ninety degrees in the other direction. During these tests, accuracy was tested by both measuring each turn of ninety degrees, and also observing for long-term drift.

This system was the least successful during the testing phase. While the drive straight functionality did keep the augers rotating at constant speeds, this did not necessarily mean that the robot was driving forward. To account for this, the gyro was incorporated into the function. By the end of the project, this functionality was still in development, and had not been fully integrated.

The ability to both drive a specified distance and turn a specified number of degrees was achieved by the end of the project. The PID control system could have been tuned slightly more to achieve a faster response time and less oscillation, however the system as a whole was fairly consistent.

Evaluation

Overall, this project was a success. Though not every goal set at the beginning of the project had been met, the majority of them were, and some goals were even exceeded.

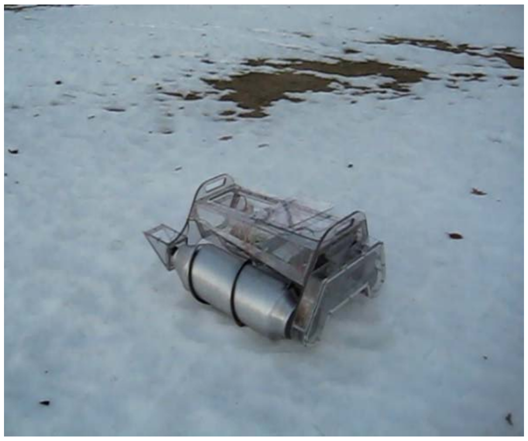


Figure 29: Final Tests on Snow

The first major goal of the project was developing a robot capable of traveling on a natural ice and snow surface. This goal has been thoroughly tested, and met. Under remote operation, the robot can easily traverse over uneven ice and snow, open water, and even patches of bare ground. Under autonomous conditions, the robot has more trouble, but still does have the capability of driving on all of these surfaces.

The next goal of the project was that the robot be able to sense its location while moving autonomously. This was one of the goals that were not quite reached. For turning and driving a set distance, the robot successfully uses the gyro and the encoders. Driving in a straight line is where the issues arise. Part of this issue may be the non-uniformity in the auger blades. While the augers are turning at the same exact rate, the difference in shape between the two blades may cause one auger to drive further than the other, causing a turn rather than a straight line. Additionally, the augers do not have quite as much grip on the ice as had originally been assumed. This may occasionally cause one auger to free-spin, while the other retains contact with the ground, causing a non straight path.

The solution that was attempted to counteract these issues, using the gyro to assist in driving straight, was not fully implemented by the end of the project. This is mainly due to a misjudgment in the amount of time e project would take. Lead times on parts ordered for the project were much greater than had been expected, which had the effect of delaying the assembly of the final robot. The final version of the robot was not completed until the last week of the project, giving very little time to test and correct the software. If this project had been spread over a larger time-span, it is likely that these issues could have been worked out.

The next goal of the project that was accomplished was the ability to receive commands from the user. This goal is demonstrated by the fact that, when a user clicks a point on a map, the system calculates the change in angle and distance needed to travel to this point. The robot then turns that number of degrees, then attempts to drive that distance while correcting for course drift.

Self-preservation upon falling through ice was the next goal of the project. This goal was not only achieved, but was also surpassed. If the ice under the robot breaks, it will float above the surface, with its own weight, and an additional 21 lbs of payload. This alone met the requirements of the project. Beyond this ability, the robot could also propel itself through the water and had the potential to drive over the edge of the ice, back onto the surface. This functionality is not fully incorporated into the autonomous mode, but with manual control of the robot, this task would likely be possible.

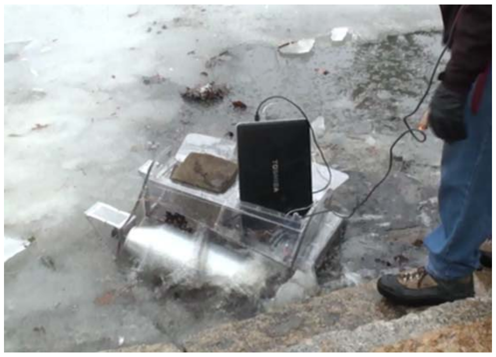


Figure 30: Final Tests in Water

The next goal of the robot was to transmit data to a user, or store data onboard. At the conclusion of the project, the robot had the ability to transmit both ice-thickness data, as well as live positional data for the robot. This more than qualified as an achievement of the goal.

The next set of requirements was for the GUI. The goals for this were the ability to receive and interpret data from the robot, create a gradient map of thicknesses, and display a map with this information on it. The GUI does all of these things, as well as showing the user the live position and orientation of the robot. In addition to all of this, the GUI has a display to prompt the user for input, and to inform the user of the current state of the program. It also provides a simple method to select a map, so that the robot can easily be used in multiple locations.

Budget Evaluation

The budget for this project was maintained rigorously using an Excel spreadsheet. This sheet kept track of all the purchases made throughout the project and recorded all the necessary details of each item purchased. These details included: description, part number, company purchased from, which system it was for, whether it was for the prototype, final version, or would be used for both, who purchased it, who paid for it, what budget it would come out of, and most importantly the total cost. Since this project was funded from multiple sources, keeping the budget spreadsheet up to date was crucial in order to ensure funds were still available for each purchase. The spreadsheet also had a feature that would keep track of who needed to be reimbursed when they paid for something that was coming out of a budget other than their personal contribution to the project.

In the project proposal, a budget of $2250 was projected; this was broken down into six system categories with individual allowances shown below in Table 3.

Table 3: Proposed Budget Breakdown

Category	Allowance
Drive Train	**$600**
Motors	**$400**
Sensors	**$400**
Chassis	**$300**
Battery/Power	**$300**
Controller	**$250**
Total	**$2250**

This budget was used as a guide while considering certain part and material decisions that were made. However it was used lightly since all of the allowances were rough estimates and no design decisions had been made at that time.

The final cost of the robot proved to be more than anticipated. Some of the categories worked out to be close to the proposed amount; however others were grossly underestimated. The “Drive Train” budget for example was estimated to be $600 in the proposal but once the final design was complete, more than $1,100 had been spent on the drive train. This was because the design decision was made to use auger drive; the materials and manufacturing fees that went into the augers alone were over $900.

The pie charts in Table 4 and Table 5 show the breakdown of the budget by each of the six systems and by the version of the robot (prototype or final design).

As can be seen in Table 6, the drive train was the most expensive system of the robot. This system was composed of nearly $200 worth of components such as chain, sprockets, bearings, and shafts, and over $900 that went into the materials and manufacturing of both the prototype and final augers. The chassis was composed of three primary expenses: materials, nuts & bolts (which included all the materials such as bolts, solvent glue, and gasket tape require to assemble and waterproof the robot), and the fees to have the parts of the chassis waterjet. This distribution is shown in Table 5.

Table 5: Budget Breakdown by System

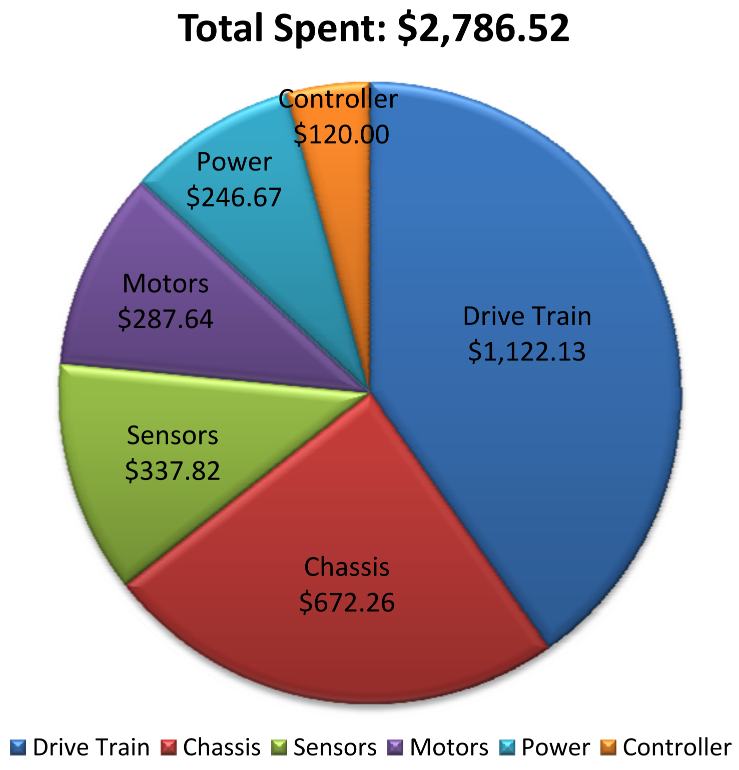


Table 6: Chassis Budget Breakdown

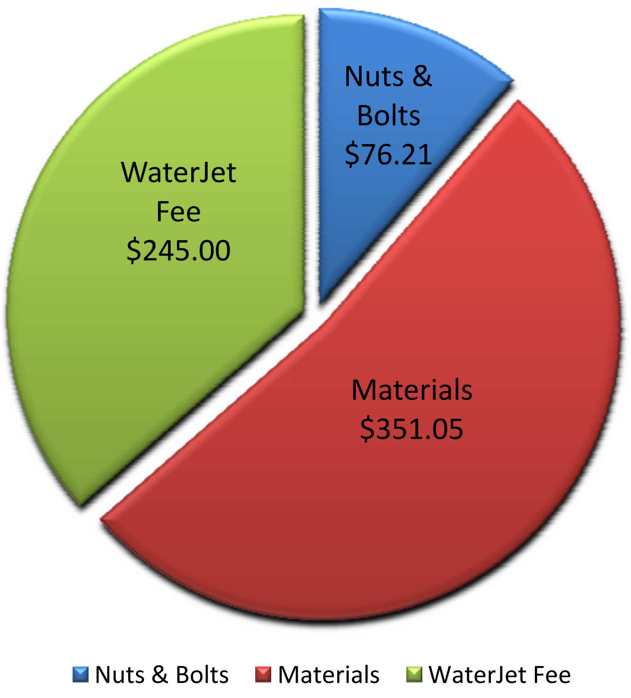
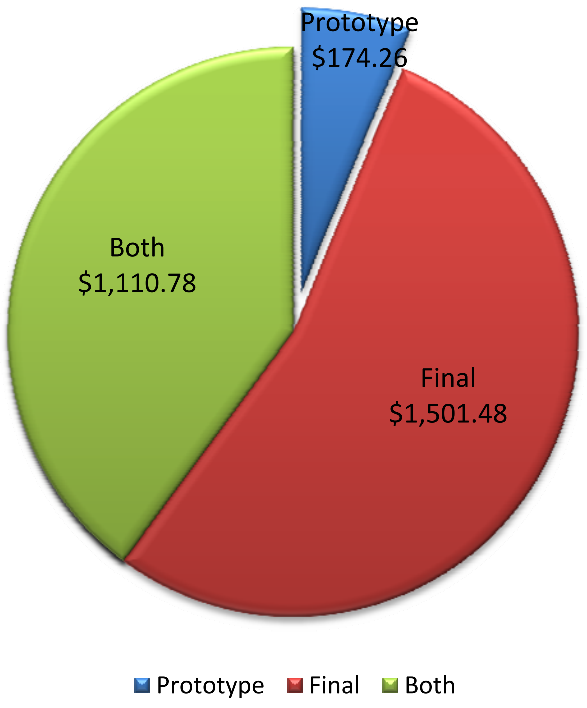


Table 4: Budget Breakdown by Version



Looking at the budget broken down by robot version (prototype or final design) you can see that there was a significant amount of money spent on the prototype that was recycled into the final version. There was only about $175 spent on parts for the prototype that were not reused in the final version. $110 of that $175 went into the materials for the prototype augers. The full bill of materials can be found in Appendix H.

Social Implications

All inventions, including robots, have some effect on society. It is important to predict these effects to gauge their impact, whether good or bad, intended or unintended, preventable or unpreventable. As part of this project, the social implications of the Auger Driven Ice Surveyor (ADIS) have been predicted in terms of safety, ethical, and moral implications.

Safety Issues

Safety issues are a vital part of any project that should be considered and identified before they cause a problem. The ADIS robot has some safety issues that must be considered before being safely operated. The first thing the operator must note is that the auger can be dangerous because the helicoid flighting can be relatively sharp, not to mention that it rotates very fast to accomplish the desired driving velocity. The second critical safety issue is that the robot is heavy enough to hurt the operator or itself when it is dropped accidentally.

To minimize these risks, some features of the mechanical design should be noted. The frame includes handles to provide ADIS with a simple and discrete method of being carried to help prevent dropping the robot accidentally. Furthermore, the blades of the auger have not been sharpened, and thus are relatively flat, much like a hockey skate, which helps prevent injury while the auger is motionless.

Figure 31: Handle of the Robot, and Auger Blade

Occupational Issues

Many robots are built with specific objectives or tasks in mind. These roles can have the potential to eliminate jobs, which is an issue that should be discussed before development. For example, where individuals once answered calls for large businesses, many have replaced them with voice recognition software. On the other hand, the servers that maintain those recognition systems generate new work for technicians and programmers.

The purpose of the ADIS robot is to measure ice thickness. Each city with a body of water that freezes annually has their own way of protecting people from the dangers of thin ice, usually by marking off dangerous areas and keeping records of ice thickness, as measured by the ruler-methods mentioned earlier. In most cases, the task of measuring the ice thickness is performed by an individual that is employed by the city, usually one that has other duties as well. ADIS is designed to make the task of measuring ice thickness easier, safer, and more complete, but it is not capable of replacing the role of the operator. The robot is merely a tool, one which still must have an individual that wields it. There is the possibility of depreciating the value of an ice measuring specialist, if there is any such individual, but there is no evidence that there are a significant number of people with this career.

Conclusion and Recommendations

Suggestions for Future Work

Though this project has been judged a success, it is far from a finished, marketable product. Future project groups could pick up the project where this group left off and make changes that significantly improve the performance of the robot.

An example of one improvement that could be made to the project would be either designing or acquiring an actual sensor to measure ice thickness, then incorporating that sensor into the platform. Another useful improvement could be incorporating GPS or some other advanced localization system, to improve upon the robot's ability to record and report its position, and the position of ice thickness readings that it makes. Fully autonomous self-recovery would be another improvement that could be made.

Another use for this project could be the repurposing of various systems of the project. For example, the auger drive system is versatile enough that another use could be found for it, potentially in a mud, sand, or fully aquatic environment. Similarly, the GUI could be repurposed to work with another robotic systems that requires mapping or positional display.

Accomplishments

The completion of this project resulted in, in addition to a semi-autonomous mobile platform for an ice thickness sensor, a group of experienced engineers. While completing the Auger Driven Ice Surveyor, the students involved in this project learned many lessons about the engineering conceptualization, design, and fabrication processes. Such lessons include setting reasonable goals for a project, accounting for lead times in early project time-lines, using specializations within a group effectively, applying weights to project goals and spending appropriate amounts of time on each, and time management with an impending

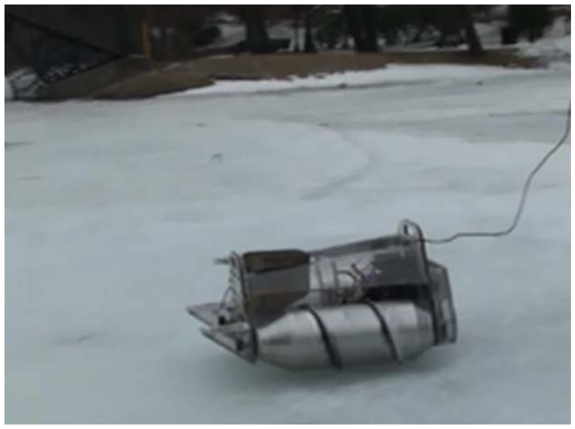


Figure 32: Final Testing on Ice

deadline. All of these lessons are important for professional engineers to have learned, as they all are applicable to the vast majority of engineering projects in both academic and professional settings.

References

1. http://2010.census.gov/news/releases/operations/big-form.html : Census – Government research about population, Number of city with more than 100000 populations.
2. http://www.dnr.state.mn.us/safety/ice/index.html : Minnesota Department of Natural Resources (DNR) “Ice safety”
3. http://files.dnr.state.mn.us/education_safety/safety/ice/ice_stats.pdf : Minnesota DNR Boat and Water Safety Section “Ice-Related Fatalities 1976-2009”
4. http://gf.nd.gov/education/ice-brochure.html: North Dakota Game and Fish Department, “Safety on Ice”
5. http://www.malags.com/Case-Studies/Case-Studies-1 : MALÅ, “Quality Inspections of Ice Roads using GPR”
6. http://www.esa.int/esaLP/ESAOMH1VMOC_LPcryosat_0.html : European Space Agency, “CryoSat-ESA's ice mission”
7. http://www.geophysical.com/: Geophysical Surveying Systems, Inc. “World Leader in Ground Penetrating Radar and Electromagnetic Induction Instruments”

Appendices

Appendix A: Initial Auger Buoyancy Calculations

These are the calculations used to determine the initial length and diameter of the pontoon augers assuming hemispherical ends.
Green values indicate user defined variables.

>

>

Density of water:

> $p := 62.4 \frac{[\![lb]\!]}{[\![ft]\!]^3}$:

>

Mass of robot:

> $m := 50 [\![lb]\!]$:

Gravitational Acceleration:

> $g := 32.17 \frac{[\![ft]\!]}{[\![s]\!]^2}$:

Minimum Buoyancy required for robot to float:

> $B_{\min} := combine(m \cdot g, 'units')$;

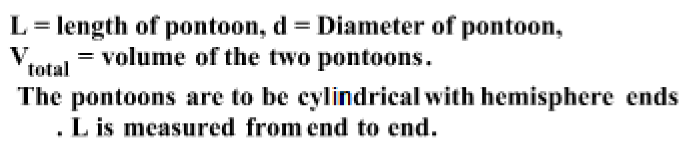
$$B_{\min} := 222.383 [\![N]\!]$$

Minumum volume required to achieve buoyancy at specified mass:

> $V_{\min} := solve\left(\frac{2 \cdot g \cdot m \cdot p \cdot v}{m + p \cdot v} = m \cdot g, v\right)$;

$$V_{\min} := 0.801288 [\![ft]\!]^3$$

L = length of pontoon, d = Diameter of pontoon,



L = length of pontoon, d = Diameter of pontoon,
V_{total} = volume of the two pontoons.
The pontoons are to be cylindrical with hemisphere ends
. L is measured from end to end.

>

$L := 2[[ft]]:$

$d := \frac{8}{12}[[ft]]:$

$V_{total} := 2 \cdot evalf\left((L-d)\cdot\left(\frac{d}{2}\right)^2 \cdot \mathrm{Pi} + \left(\frac{4}{3}\right)\cdot \mathrm{Pi} \cdot \left(\frac{d}{2}\right)^3\right);$

$$V_{total} := 1.24112\,[[ft]]^3$$

These calculations use the provided values of m, L, and d to determine the percentage of the pontoon that will be submerged:

V_{CylSub} = volume of the cylinder section of a pontoon submerged as a function of the percentage submerged (P)

V_{ShpSub} = volume of the sphere section of a pontoon submerged as a function of the percentage submerged (P)

V_{Sub} = total volume of a single pontoon submerged as a function of the percentage submerged (P)

$V_{TotalSub}$ = total volume of both pontoons submerged as a function of the percentage submerged (P)

>

$V_{CylSub} := evalf\left(\frac{((L-d)\cdot d^2)}{8}\cdot\left(2\cdot \mathrm{Pi}\cdot\frac{P}{100} - \sin\left(2\cdot \mathrm{Pi} \cdot\frac{P}{100}\right)\right)\right):$

$V_{ShpSub} := evalf\left(\left(\frac{\mathrm{Pi}}{3}\right)\cdot\left(d\cdot\frac{P}{100}\right)^2\cdot\left(1.5\cdot d - \left(d\cdot\frac{P}{100}\right)\right)\right):$

$V_{Sub} := V_{CylSub} + V_{ShpSub}:$

$V_{TotalSub} := 2\cdot V_{Sub}:$

$P_{sub} := solve\left(V_{TotalSub} = V_{\min}, P\right);$

$$P_{sub} := 57.908($$

Appendix B: Final Auger Buoyancy Calculations

These are the calculations to determine if the actual volumes of the prototype and final auger designs were sufficient to support the weight of the robot.

>

Density of water:

> $p := 62.4 \frac{[\![lb]\!]}{[\![ft]\!]^3}$:

Gravitational Acceleration:

> $g := 32.17 \frac{[\![ft]\!]}{[\![s]\!]^2}$:

Mass of robot:

> $m_{proto} := 45[\![lb]\!] : m_{final} := 55[\![lb]\!]$:

Minimum Buoyancy required for robot to float:

> $B_{proto} := combine(m_{proto} \cdot g, 'units');$
$B_{final} := combine(m_{final} \cdot g, 'units');$

$$B_{proto} := 200.145 [\![N]\!]$$
$$B_{final} := 244.621 [\![N]\!]$$

Minumum volume required to achieve buoyancy at specified robot mass:

>

$V_{proto} := solve\left(\frac{2 \cdot g \cdot m_{proto} \cdot p \cdot v}{m_{proto} + p \cdot v} = m_{proto} \cdot g, v\right);$

$V_{final} := solve\left(\frac{2 \cdot g \cdot m_{final} \cdot p \cdot v}{m_{final} + p \cdot v} = m_{final} \cdot g, v\right);$

$$V_{proto} := 0.721152 [\![ft]\!]^3$$
$$V_{final} := 0.881417 [\![ft]\!]^3$$

Actual volume of prototype auger:
22 inch long cyclinder with 8.625 inch diameter

>
$$L := \frac{22}{12}[\![ft]\!] :$$
$$d := \frac{8.625}{12}[\![ft]\!] :$$

$$TotalVol := 2\cdot evalf\left(L\cdot\left(\frac{d}{2}\right)^2\cdot \mathrm{Pi}\right);$$

$$TotalVol := 1.48771\,[\![ft]\!]^3$$

Actual volume of final auger design:
16 inch long cylinder with 8.0 inch diameter and conical ends.
Front cone base diameter of 8 inches, top diameter of 4 inches, and height of 4 inches.
Back cone base diameter of 8 inches, top diameter of 5.5 inches, and height of 2 inches.

Cylinder Volume:

> $L := \frac{16}{12}[\![ft]\!] : d := \frac{8.0}{12}[\![ft]\!] : \; CylVol := evalf\left(L\cdot\left(\frac{d}{2}\right)^2\cdot \mathrm{Pi}\right);$

$$CylVol := 0.465420\,[\![ft]\!]^3$$

Cone Volumes:

>
$$\mathrm{Diameter}_{Base} := \frac{8}{12}[\![ft]\!] :$$
$$\mathrm{Diameter}_{Top1} := \frac{4}{12}[\![ft]\!] :$$
$$\mathrm{Diameter}_{Top2} := \frac{5.5}{12}[\![ft]\!] :$$
$$H1 := \frac{4}{12}[\![ft]\!] :$$
$$H2 := \frac{2}{12}[\![ft]\!] :$$
$$ConeVol_{front} := evalf\left(\frac{1}{3}\cdot\mathrm{Pi}\cdot\left(\left(\frac{\mathrm{Diameter}_{Base}}{2}\right)^2 + \left(\frac{\mathrm{Diameter}_{Base}\cdot\mathrm{Diameter}_{Top1}}{2}\right) + \left(\frac{\mathrm{Diameter}_{Top1}}{2}\right)^2\right)\cdot H1\right);$$
$$ConeVol_{back} := evalf\left(\frac{1}{3}\cdot\mathrm{Pi}\cdot\left(\left(\frac{\mathrm{Diameter}_{Base}}{2}\right)^2 + \left(\frac{\mathrm{Diameter}_{Base}\cdot\mathrm{Diameter}_{Top2}}{2}\right) + \left(\frac{\mathrm{Diameter}_{Top2}}{2}\right)^2\right)\cdot H2\right);$$

$$ConeVol_{front} := 0.0872665\,[\![ft]\!]^3$$
$$ConeVol_{back} := 0.0552232\,[\![ft]\!]^3$$

Helix Volume

>
$r := \frac{4}{12} [\![ft]\!] :$

$pitch := \frac{8}{12} [\![ft]\!] :$

$Height := \frac{.5}{12} [\![ft]\!] :$

$Width := \frac{.1875}{12} [\![ft]\!] :$

$Length := combine\left(evalf\left(\text{sqrt}\left((4 \cdot \text{Pi} \cdot r)^2 + pitch^2\right)\right), 'units', 'system' = 'FPS'\right) :$

$HelixVol := Length \cdot Width \cdot Height;$

$$HelixVol := 0.00276140 [\![ft]\!]^3$$

Auger Volume:

> $AugerVol := CylVol + ConeVol_{front} + ConeVol_{back} + HelixVol;$
$TotalVol := 2 \cdot AugerVol;$

$$AugerVol := 0.610670 [\![ft]\!]^3$$

$$TotalVol := 1.22134 [\![ft]\!]^3$$

Maximum robot weight that can be supported by final auger dimensions

> $Weight_{Max} := solve\left(\frac{2 \cdot g \cdot m_{\max} \cdot p \cdot TotalVol}{m_{\max} + p \cdot TotalVol} = m_{\max} \cdot g, m_{\max}\right)[2];$

$$Weight_{Max} := 76.2118 [\![lb]\!]$$

Max Payload

> $Payload_{Max} := Weight_{Max} - m_{final};$

$$Payload_{Max} := 21.2118 [\![lb]\!]$$

Appendix C: Drive Speed Calculations

These are the calculations used to determine the speed the augers need to be driven to move at the desired speed.
Green values indicate user defined variables.

```
>
  with(Units) :
  AddUnit('revolution','spelling'='rev','plural'='revolutions') :
  AddUnit('minute','spelling'='min','plural'='minutes') :
  AddUnit('mile','plural'='miles') :
  AddUnit('hour','spelling'='h','plural'='hours') :
```

Desired Drive Speed of Robot.

> $S_d := 5 \frac{[\![ft]\!]}{[\![s]\!]}$:

Drive Speed converted to Mph.

> $S_{mph} := evalf\left(S_d \cdot \frac{3600 [\![s]\!]}{5280 [\![ft]\!]} \cdot 1 \frac{[\![mile]\!]}{[\![hour]\!]} \right);$

$$S_{mph} := \frac{3.41 [\![mi]\!]}{[\![h]\!]}$$

Auger Properties

Pitch (P_a)
Diameter (d)
Length (L)
Thread Height (h)

>

$P_a := \frac{8}{12} \frac{[\![ft]\!]}{[\![rev]\!]}$:

$d := \frac{8.625}{12} [\![ft]\!]$:

$L := \frac{(24 - d)}{12} [\![ft]\!]$:

$h := \frac{.75}{12} [\![ft]\!]$:

Required RPM of Pontoon Augers.

> $RPM := evalf\left(solve\left(x = S_d \cdot \left(\frac{1}{P_a} \right), x \right) \cdot 60 \frac{[\![s]\!]}{[\![min]\!]} \right);$

$$RPM := \frac{450. [\![rev]\!]}{[\![min]\!]}$$

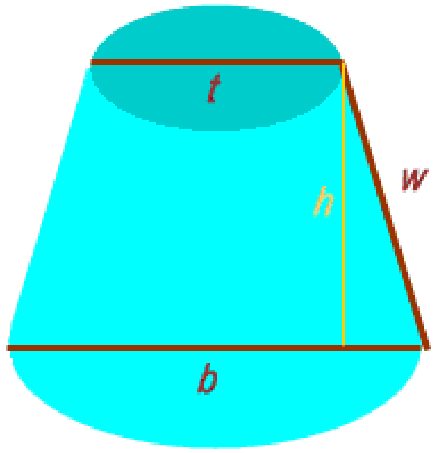
>

Appendix D: Cone Calculations

Front Cone

These are the calculations to determine the geometry needed to be folded into the front cone

Equations and pictures found at: http://mathcentral.uregina.ca/QQ/database/QQ.02.06/phil1.html



for the example these pictures correspond to: t = 290, b = 550, and h = 250.

> $t := 4 : b := 8 : h := 4 :$

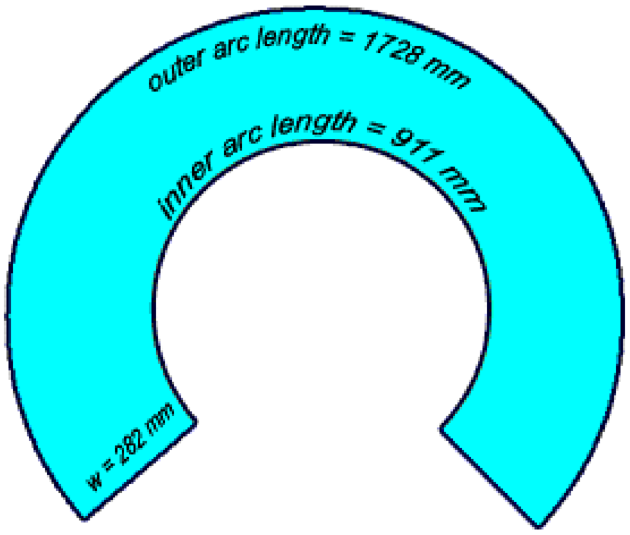
> $ArcLenght_{inner} := evalf(\mathrm{Pi} \cdot t);\ ArcLength_{outer} := evalf(\mathrm{Pi} \cdot b);$

$$ArcLenght_{inner} := 12.566$$

$$ArcLength_{outer} := 25.132$$

> $w := evalf\left(\mathrm{sqrt}\left(\left(\frac{(b-t)}{2}\right)^2 + h^2\right)\right);$

$$w := 4.4721$$



> $ArcRatio := \dfrac{ArcLenght_{inner}}{ArcLength_{outer}};$

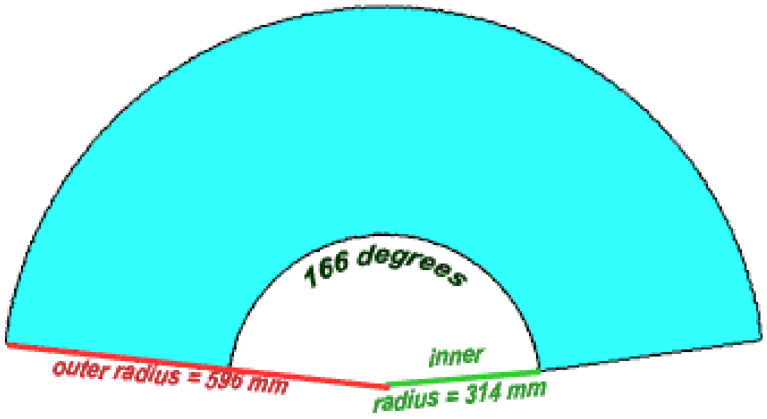
$$ArcRatio := 0.500002$$

> $r := solve\left(\dfrac{x}{(x+w)} = ArcRatio, x\right);$

$$r := 4.47218$$

> $R := r + w;$

$$R := 8.94432$$



> $Circumference_{inner} := evalf\,(2 \cdot \mathrm{Pi} \cdot r);$

$$Circumference_{inner} := 28.0995$$

> $PercentOfCircle := \dfrac{ArcLenght_{inner}}{Circumference_{inner}};$

$$PercentOfCircle := 0.447211$$

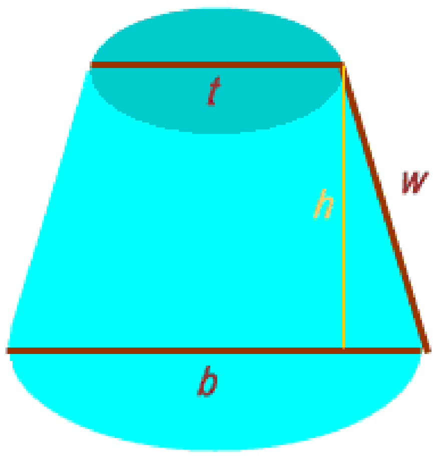
> $Degrees := PercentOfCircle \cdot 360;$

$$Degrees := 160.996$$

Back Cone

These are the calculations to determine the geometry needed to be folded into the back cone

Equations and pictures found at: http://mathcentral.uregina.ca/QQ/database/QQ.02.06/phil1.html



for the example these pictures correspond to: t = 290, b = 550, and h = 250.

> $t := 5.5 : b := 8 : h := 2 :$

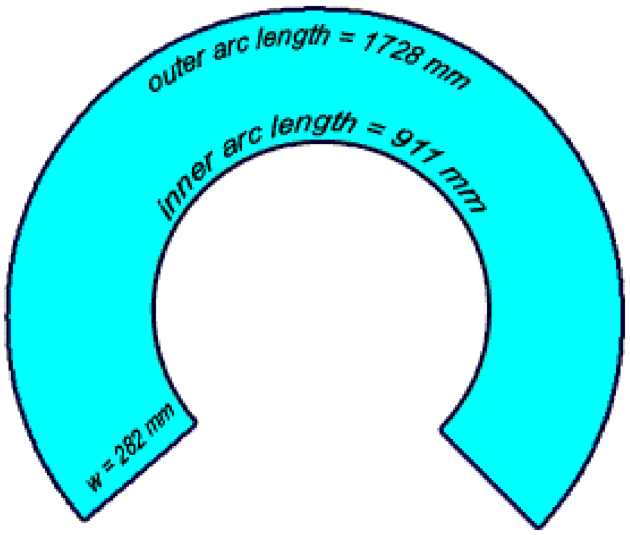
> $ArcLenght_{inner} := evalf(\text{Pi} \cdot t);\; ArcLength_{outer} := evalf(\text{Pi} \cdot b);$

$$ArcLenght_{inner} := 17.2787$$

$$ArcLength_{outer} := 25.1327$$

> $w := evalf\left(\text{sqrt}\left(\left(\frac{(b-t)}{2}\right)^2 + h^2\right)\right);$

$$w := 2.35850$$



> $ArcRatio := \frac{ArcLenght_{inner}}{ArcLength_{outer}};$

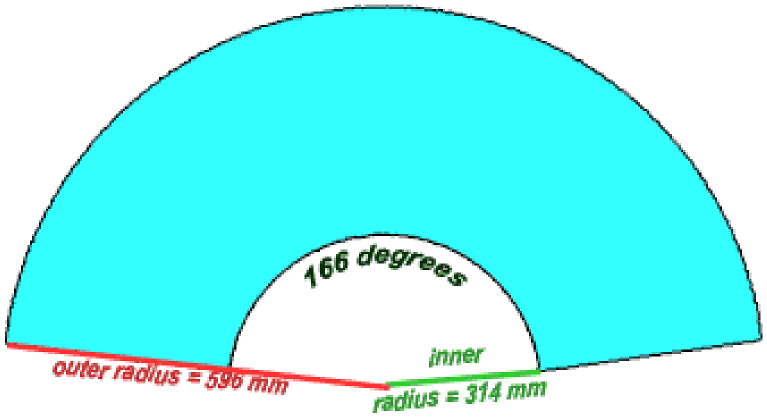
$$ArcRatio := 0.687499$$

> $r := solve\left(\frac{x}{(x+w)} = ArcRatio, x\right);$

$$r := 5.18868$$

> $R := r + w;$

$$R := 7.54718$$



> $Circumference_{inner} := evalf(2 \cdot \mathrm{Pi} \cdot r);$

$$Circumference_{inner} := 32.6015$$

> $PercentOfCircle := \frac{ArcLenght_{inner}}{Circumference_{inner}};$

$$PercentOfCircle := 0.529997$$

> $Degrees := PercentOfCircle \cdot 360;$

$$Degrees := 190.799$$

Appendix E: Stability Calculations

Spherical end-shape (Example)

The robot will flip over when center of gravity goes over the last point of contacting ground. Reason is that moment generated from gravity force does not have reaction moment to balance the moment.

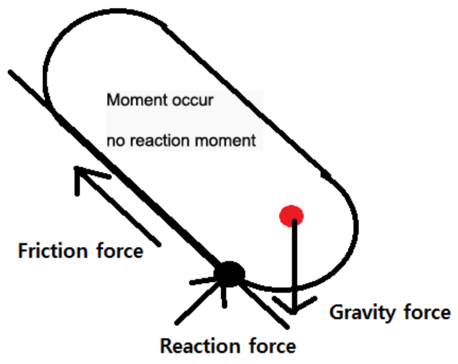


Figure 33: Free Body Diagram of The Robot on Angled Terrain.

Due to irregular shape of augers, last point of contacting ground changes while The robot is flipping. I assumed auger will not slip since auger thread will create enough friction force between ice and The robot.

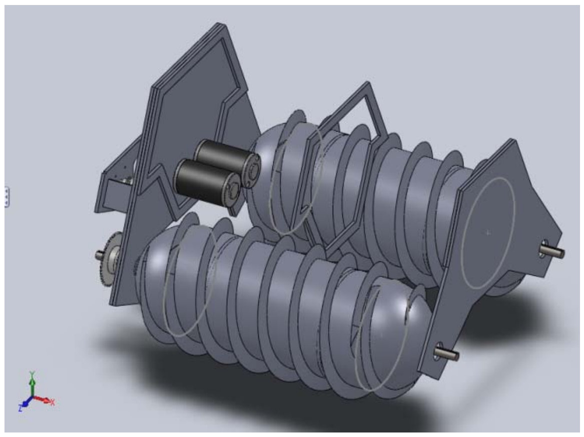
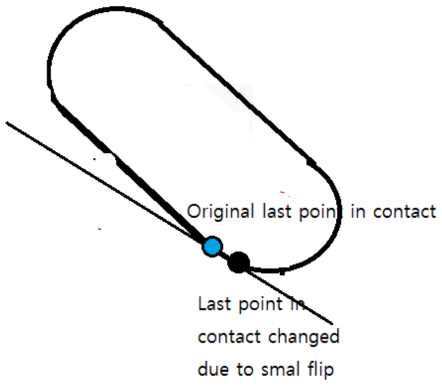


Original last point in contact
Last point in contact changed due to smal flip

Figure 34: Auger Model with X, Y, and Z Coordinates and Explanation

By using coordinate system to define the changes in last point in contact, I was able to make a graph of angle of terrain versus last point in contact changes.

Stability for x dir

X and Y are achieved from Excel sheet that list all the weights

$X := 4in$

$Y := 6in$

A is distance from 0.0 to center of gravity

$A := \left(\sqrt{X^2 + Y^2}\right) = 7.211 \cdot in$

B is distance from 0.0 to last point contacting the ground in x direction

$y(x) := 4in - \sqrt{-(x \cdot in)^2 + 16in^2}$

$B(x) := \sqrt{(x \cdot in + 12in)^2 + \left[4in - \sqrt{-(x \cdot in)^2 + 16in^2}\right]^2}$

Range of x is from 0in to 4in

Finding θ when center of gravity goes over last point contacting ground

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right) - \theta\right) \cdot A = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x + 12}\right) - \theta\right) \cdot B(x)$$

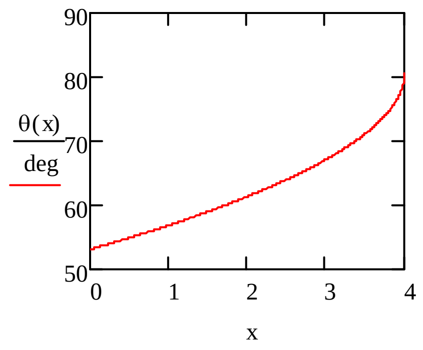
Law of cosine

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) \cdot \cos(-\theta) \cdot A + \sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) \cdot \sin(-\theta) \cdot A = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x + 12}\right)\right) \cdot \cos(-\theta) \cdot B(x) \ldots$$
$$+ \sin\left(\operatorname{atan}\left(\frac{y(x)}{x + 12}\right)\right) \cdot \sin(-\theta) \cdot B(x)$$

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) + \sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) \cdot \tan(-\theta) = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x + 12}\right)\right) \cdot \frac{B(x)}{A} \ldots$$
$$+ \sin\left(\operatorname{atan}\left(\frac{y(x)}{x + 12}\right)\right) \cdot \tan(-\theta) \cdot \frac{B(x)}{A}$$

Using law of cosine, I was able to make equation in form of θ(x)

$$\theta(x) := -\operatorname{atan}\left(\frac{\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) - \cos\left(\operatorname{atan}\left(\frac{y(x)}{x \cdot in + 12in}\right)\right) \cdot \frac{B(x)}{A}}{\sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) - \sin\left(\operatorname{atan}\left(\frac{y(x)}{x \cdot in + 12 \cdot in}\right)\right) \cdot \frac{B(x)}{A}}\right)$$



Maximum angle of stability is

$\theta(4) = 80.538\text{deg}$

Opposite direction Calculation

Finding θ when center of gravity goes over last point contacting ground

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)+\theta\right)\cdot A = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+12}\right)-\theta\right)\cdot B(x)$$

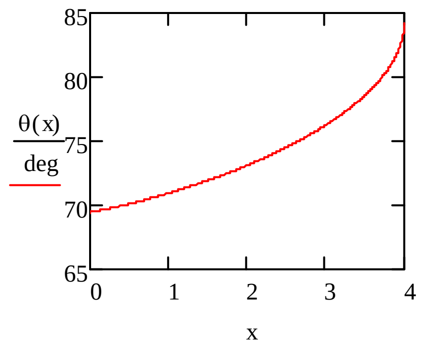
Law of cosine

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right)\cdot\cos(\theta)\cdot A + \sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right)\cdot\sin(\theta)\cdot A = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+12}\right)\right)\cdot\cos(-\theta)\cdot B(x)\ \ldots$$
$$+\left(-\sin\left(\operatorname{atan}\left(\frac{y(x)}{x+12}\right)\right)\right)\cdot\sin(-\theta)\cdot B(x)$$

$$\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right) + \sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right)\cdot\tan(\theta) = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+12}\right)\right)\cdot\frac{B(x)}{A}\ \ldots$$
$$+\left(-\sin\left(\operatorname{atan}\left(\frac{y(x)}{x+12}\right)\right)\right)\cdot\tan(\theta)\cdot\frac{B(x)}{A}$$

Using law of cosine, I was able to make equation in form of θ(x)

$$\theta(x) := \operatorname{atan}\left(\frac{\cos\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right)+\cos\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in+12in}\right)\right)\cdot\frac{B(x)}{A}}{\sin\left(\operatorname{atan}\left(\frac{Y}{X}\right)\right)-\sin\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in+12\cdot in}\right)\right)\cdot\frac{B(x)}{A}}\right)$$



Maximum angle of stability is

$\theta(4) = 84.289\text{deg}$

Stability for z dir

Z is achieved from Excel sheet that list all the weights

$Z := 0.000001\text{in}$

A is distance from 0.0 to center of gravity in

Z direction

$A := \left(\sqrt{Z^2 + Y^2}\right) = 6\cdot\text{in}$

B is distance from 0.0 to last point contacting the ground in z direction

$B(x) := \sqrt{(x\cdot\text{in} + 8\text{in})^2 + \left[4\text{in} - \sqrt{-(x\cdot\text{in})^2 + 16\text{in}^2}\right]^2}$

Range of x is from 0in to 4in

Finding θ when center of gravity goes over last point contacting ground

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right) - \theta\right)\cdot A = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right) - \theta\right)\cdot B(x)$$

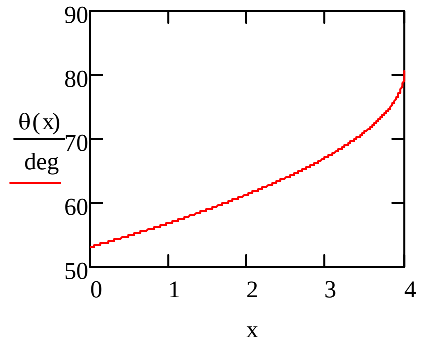
Law of cosine

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\cos(-\theta)\cdot A + \sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\sin(-\theta)\cdot A = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\cos(-\theta)\cdot B(x) \ldots$$
$$+ \sin\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\sin(-\theta)\cdot B(x)$$

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) + \sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\tan(-\theta) = \cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\frac{B(x)}{A} \;\ldots$$
$$+\sin\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\tan(-\theta)\cdot\frac{B(x)}{A}$$

Using law of cosine, I was able to make equation in form of θ(x)

$$\theta(x) := -\operatorname{atan}\left(\frac{\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) - \cos\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in + 8in}\right)\right)\cdot\frac{B(x)}{A}}{\sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) - \sin\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in + 8\cdot in}\right)\right)\cdot\frac{B(x)}{A}}\right)$$



Maximum angle of stability is

$$\theta(4) = 80.538\text{deg}$$

Opposite direction Calculation

Finding θ when center of gravity goes over last point contacting ground

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right) + \theta\right)\cdot A = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right) - \theta\right)\cdot B(x)$$

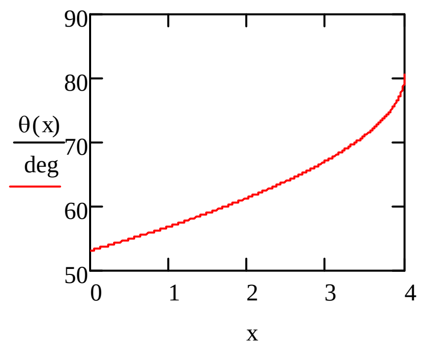
Law of cosine

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\cos(\theta)\cdot A + \sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\sin(\theta)\cdot A = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\cos(-\theta)\cdot B(x) \;\ldots$$
$$+\left(-\sin\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\right)\cdot\sin(-\theta)\cdot B(x)$$

$$\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) + \sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right)\cdot\tan(\theta) = -\cos\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\cdot\frac{B(x)}{A} \;\ldots$$
$$+\left(-\sin\left(\operatorname{atan}\left(\frac{y(x)}{x+8}\right)\right)\right)\cdot\tan(\theta)\cdot\frac{B(x)}{A}$$

Using law of cosine, I was able to make equation in form of θ(x)

$$\theta(x) := \operatorname{atan}\left(\frac{\cos\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) + \cos\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in + 8in}\right)\right)\cdot\frac{B(x)}{A}}{\sin\left(\operatorname{atan}\left(\frac{Y}{Z}\right)\right) - \sin\left(\operatorname{atan}\left(\frac{y(x)}{x\cdot in + 8\cdot in}\right)\right)\cdot\frac{B(x)}{A}}\right)$$



Maximum angle of stability is

$\theta(4) = 80.538\text{deg}$

Appendix F: Heat Calculations

Heat dissipation rate of 2 motor

$V_{input} := 15\,V$

$15\,V$

$A_{input} := 10\,A$

$10\,A$

$P_{input} := V_{input} \cdot A_{input} \cdot \frac{W}{V \cdot A}$

$\frac{150\,J}{s}$

$P_{output} := 0.65 \cdot P_{input}$

$\frac{97.50\,J}{s}$

$Q_{dot} := P_{input} - P_{output}$

$\frac{52.50\,J}{s}$

$Q_{total} := 2 \cdot Q_{dot}$

$\frac{105.00\,J}{s}$

Assume temperature of motor (Aluminum heat sink) is 60 degree Celsius and Ambient temperature is 10 degree Celsius

$T_c := (60 + 273)\,K$

$333\,K$

$h_{out} := 30\,\frac{W}{m^2 \cdot K}$

$\frac{30\,J}{s\,m^2\,K}$

$T_{inf} := (10 + 273)\,K$

$283\,K$

$A_{sink} := 0.01\,m^2$

$0.01\,m^2$

$q_{conv} := A_{sink} \cdot h_{out} \cdot (T_c - T_{inf})$

$\frac{15.00\,J}{s}$

Lexan conduction from inside to outside

$k_{Lexan} := \frac{0.18\,W}{m \cdot K}$

$\frac{0.18\,J}{s\,m\,K}$

$q_{lexan} := Q_{total} - q_{conv}$

$$\frac{90.00\,J}{s}$$

$A_{lexan} := 0.3\,m^2$

$$0.3\,m^2$$

$dx := 0.01\,m$

$$0.01\,m$$

$$dT := \frac{q_{lexan}}{A_{lexan} \cdot k_{Lexan}} \cdot dx$$

$$16.66666667\,K$$

It goes into steady state when inside temperature and outside temperature is 17 degree different inside temperature and outside temperature. So if inside temperature is 37 degree Celsius and outside surface temperature will be 20 degree.

$T_{Lexan} := (20 + 273)\,K$

$$293\,K$$

$$h_{out} := 30\,\frac{W}{m^2 \cdot K}$$

$$\frac{30\,J}{s\,m^2\,K}$$

$T_{inf} := (10 + 273)\,K$

$$283\,K$$

$A_{Lexan} := 0.3\,m^2$

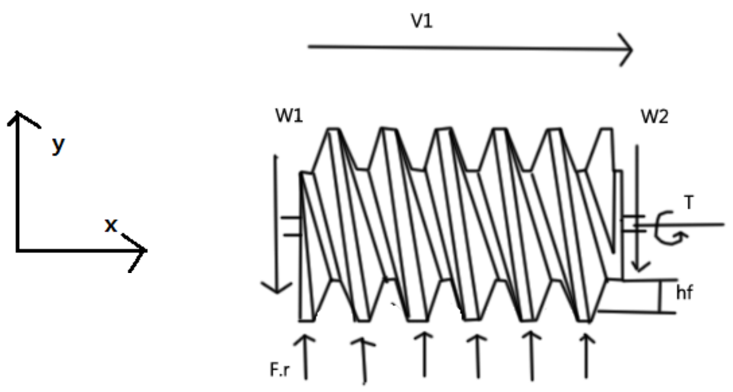
$q_{conv} := A_{Lexan} \cdot h_{out} \cdot \left(T_{Lexan} - T_{inf}\right)$

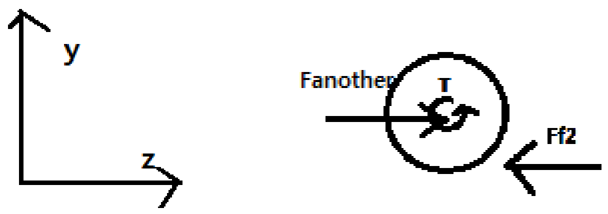
$$\frac{90.0\,J}{s}$$

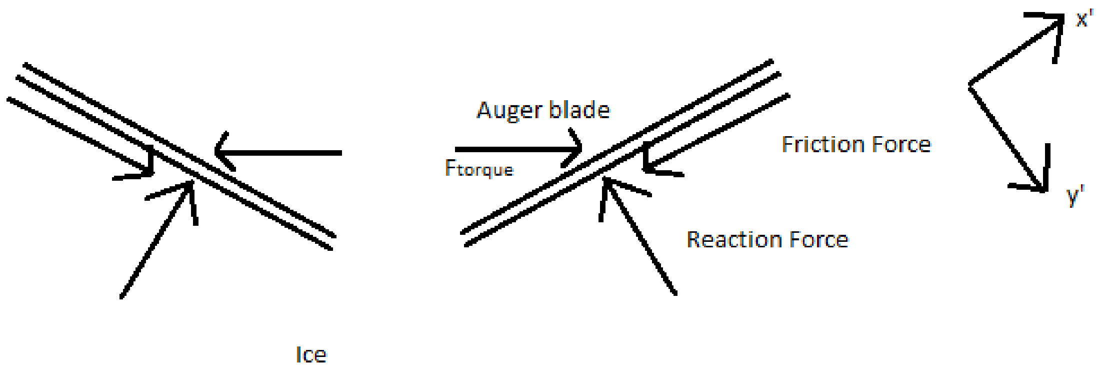
With robot 37 degree Celsius, it became steady state.

Appendix G: Free Body Diagram (Torque Calculations)

Free Body Diagram







Assume Weight is distributed evenly, Friction between bottom of blade and ice is negligible

Distance from location of Reaction force to neutral $h_f := 0.95 \cdot h = 0.712 \cdot \text{in}$

Weight of robot $W := 50 \text{lbf}$

Length of Wheel $L := 16 \text{in}$

Each Reaction force from Ground $F_r := \dfrac{\dfrac{W}{2}}{\dfrac{2L}{L_{pitch}}} = 6.25 \cdot \text{lbf}$

Torque at peak efficiency for motor $T_{rotate}(x) := x\,lbf \cdot in$

Coefficient of Friction between ice and steel $\mu := 0.03$

Reaction force $R_{blade}(x) := \dfrac{T_{rotate}(x) \cdot \cos(\alpha)}{h_f}$

Friction force of blade and ice $F_{f\_blade}(x) := R_{blade}(x) \cdot \mu + F_r \cdot \mu$

$F_{torque\_yprime}(x) := R_{blade}(x)$ $F_{torque\_xprime}(x) := \dfrac{T_{rotate}(x) \cdot \sin(\alpha)}{h_f}$

Force cancel each other by left over

$F_{left}(x) := F_{torque\_xprime}(x) - F_{f\_blade}(x)$ This force is cancel out by other auger force

$x := 0, 0.1 .. 30$

All the forces cancel out

Torque requirement can be calculated by F.torque_xprime = F.f_blade

$F_{torque\_xprime\_min} := 0.834 lbf$

$Torque_{minimum} := 0.4 \dfrac{lbf}{in} = 6.4 \cdot \dfrac{ozf}{in}$

Appendix H: Bill of Materials

Item	Quantity	Part Number	System	Purchased From	Total Cost
1/8 x 3 x 36 inch Stainless Steel	1	8992K383	Drive Train	McMASTER-CARR	39.58
1/8" Hex Insert Bit	1	8526A64	Chassis	McMASTER-CARR	9.86
1/8" T-Handel Hex Key	1	7391A53	Chassis	McMASTER-CARR	29.26
10' x 0.25" pvc rod	3	n/a	Drive Train	Plastics Unlimited	21.34
10' x 0.25" pvc rod	1	n/a	Drive Train	Plastics Unlimited	251.01
10' x 8" Schedule 40 PVC pipe	1	n/a	Drive Train	Washburn-Garfield	245
100 Pack 10-24 x 1/2" Button Head Screw	1	92949A242	Chassis	McMASTER-CARR	40.16
100 Pack Ny-Locknut 1/4-20	1	91831A029	Chassis	McMASTER-CARR	6.98
100 Pack Ny-Locknut 10-24	1	91831A011	Chassis	McMASTER-CARR	1.4
10x36x.25 Aluminum	1	8975K117	Drive Train	McMASTER-CARR	3.8
12 gauge wire - 2 strand Red/Black	25	9697T4	Power	McMASTER-CARR	1.09
12-Tooth ANSI 25 Sprocket 1/4 inch Bore	2	2737T101	Drive Train	McMASTER-CARR	5
12-Tooth ANSI 25 Sprocket 3/8 inch Bore	2	2737T102	Drive Train	McMASTER-CARR	6.24
12x12x.25 Aluminum	1	9246K13	Drive Train	McMASTER-CARR	6.53
12x12x1/4 6061 Aluminum	1	9246K13	Drive Train	McMASTER-CARR	5.01
12x24x.05 Aluminum	1	88895K44	Drive Train	McMASTER-CARR	120
24x24x.05 Aluminum	1	88895K54	Drive Train	McMASTER-CARR	37.64
25 Pack 1/4-20 x 3/4" Torx Pan Head Screw	1	96710A737	Chassis	McMASTER-CARR	436
3 Pair of 4mm Gold Plated Bullet Connectors	1	B000X4RZ2E	Power	Amazon	89.52
3/16" x 1/2" x 6' Stainless Steel	1	8992K17	Drive Train	McMASTER-CARR	58.36
3/4 Dimeter Polycarbonate Rod (1 foot)	2	8571K15	Chassis	McMASTER-CARR	41.38
3/8 keyed Stainless Steel Shaft - 3"	2	1497K4	Drive Train	McMASTER-CARR	27.17
36-Tooth ANSI 25 Sprocket 3/8 inch Bore	2	2737T261	Drive Train	McMASTER-CARR	14.21
3x1 Connector Housing	20	JS-1108-03-R	Sensors	Jameco Electronics	11.82
48-Tooth ANSI 25 Sprocket 5/8 inch Bore	2	2737T322	Drive Train	McMASTER-CARR	14.1
4x1 Connector Housing	3	100803	Sensors	Jameco Electronics	24.15
5 inch OD Polycarb Tube (1 foot)	1	8585K45	Chassis	McMASTER-CARR	19
5/8 non-key Auger Drive Shaft - 5ft	1	1346K28	Drive Train	McMASTER-CARR	20
50' 3-Color Heavy Gauge Servo Wire	1	57417.00	Sensors	Jameco Electronics	30
5x1 Connector Housing	4	163686	Sensors	Jameco Electronics	16.93998
60 Amp Fuses	1	EAGU60-4	Power	Amazon	13.75
8" Diamerter 16" Aluminum Pipe	2		Drive Train	Global Technology & Engineering	6.48
8" x 8" X 3/8" 6061 Aluminum	1	9246K21	Chassis	McMASTER-CARR	71.93
8ft ANSI 25 Roller Chain	1	6261K288	Drive Train	McMASTER-CARR	218
Analog Compass	1	1525.00	Sensors	Images Scientific Instruments	3.32
ANSI 25 Master Link	4	6261K108	Drive Train	McMASTER-CARR	8.74

ArduIMU	1	n/a	Sensors	Spark Fun	11.52
Bearing 3/8 ID 7/8 OD Double Sealed	4	6384K23	Drive Train	McMASTER-CARR	27.6
Bearings 5/8 ID Flanged, Double Sealed	4	6384K365	Drive Train	McMASTER-CARR	14.24
Blue Lipo 4-Cell 5000mAh 14.8v 4S1P 30C RC Battery	2	83P-5000mAh-4S1P-148-40C	Power	HobbyPartz	38.64
Brushed DC Motor Controllers (Jaguar Black)	2	MDL-BDC24	Motors	DigiKey	20.66
CIM Motors	2	M4-R0062-12	Motors	Trossen Robotics	26.74
Compass Module (not just the component)	1	HMC6352	Sensors	Spark Fun	27.2
Crimper	1	HT-202A-R	Sensors	Jameco Electronics	69.64
DyIO Controller	1	n/a	Controller	Neuron Robotics	11.02
Female Pins	30	100766	Sensors	Jameco Electronics	8.95
Fused Distribution Block	1	Scosche EADB4	Power	Amazon	4.5
Ground Distribution Block	1	EDB	Power	Amazon	11.4
Helicoid Flighting	2	n/a	Drive Train	Falcon Industries	4
I2c to UART board	1	BOB-09981	Sensors	Spark Fun	0.87
J-B Weld Epoxy 2 oz	2	7605A13	Drive Train	McMASTER-CARR	1.4
Li-Po GUARD Safety Battery Storage Bag	1	n/a	Power	HobbyPartz	15.95
Male Pins	60	Y-1800-TX-R	Sensors	Jameco Electronics	52.06
Panel Mount 120A Circuit Breaker	1	CB3-PM-120	Power	Terminal Supply Co	133.05
Polycarbonate for Chassis	1	n/a	Chassis	Plastics Unlimited	51.5
Quadrature Encoder	2	SP-16	Sensors	US-Digital	19.19
scrap lexan	5	n/a	Drive Train	Plastics Unlimited	34.95
Thunder AC6 Charger	1	n/a	Power	HobbyPartz	4.95
Torx T30 Insert Bit	1	7013A29	Chassis	McMASTER-CARR	9.44
Torx T30 Screwdriver	1	5756A19	Chassis	McMASTER-CARR	9.48
Two-Piece Clamp-On Shaft Collar	2	6436K136	Drive Train	McMASTER-CARR	13.95
Value Seal Gasket Tape 1/2 wide 50ft roll	1	9477K21	Chassis	McMASTER-CARR	99.98
WaterJet Fees for Cutting Polycarb Parts	1	n/a	Chassis	Vangy Tool	5.75
Welding fees for Augers	1	n/a	Drive Train	Barnstorm Cycles	54.7
Welding fees for Augers	1	n/a	Drive Train	Barnstorm Cycles	16.25
Weld-On Solvent and Applicator supplies	1	IPS16-PT	Chassis	Ridout Plactics	32.17
				Grand Total:	2786.52

Appendix I: Robot Code

AbSensor

```
package package1;


import com.neuronrobotics.sdk.dyio.DyIO;
import com.neuronrobotics.sdk.dyio.peripherals.AnalogInputChannel;




// An Abstract Sensor
abstract public class AbSensor implements ISensor{
	//FIELDS
	public double LastReading = -1;//The most recent reading from the 
sensor
	public long TimeTaken = System.currentTimeMillis();//The time the last 
reading was taken
	public int Port;//The port of the DyIO that this sensor is plugged into
	public DyIO dyio;//The dyio
	public AnalogInputChannel AnInput;//The channel that controls this 
sensor


	//CONSTRUCORS
	AbSensor(DyIO dyio,int Port){
		this.dyio = dyio;
		this.Port = Port;
		this.AnInput = new AnalogInputChannel(dyio.getChannel(Port), 
true);

	}
	AbSensor(){
	
	}


	//METHODS
	//Updates TimeTaken to the current time
	public void UpdateTime(){
		TimeTaken = System.currentTimeMillis();
		}

	//get the last reading
	public double GetReading(){
		return this.LastReading;
	}

	//get the time of the last reading
	public long GetReadingTime(){
		return this.TimeTaken;
	}
	//get the port of the sensor
	public int GetPort(){
		return this.Port;
	}

	//Reads the sensor and updates the LastReading and TimeTaken
	abstract public void ReadSensor();
}
```

AngleDialog

```
package package1;

import java.awt.Graphics;

//An abstract Sensor
public class AngleDialog{

	//FIELDS
	double angle=0;
	Triangle triangle = new Triangle();
	Display display;
	JLabel label;

	//CONSTRUCTOR
	public AngleDialog(Display d) {
		this.display = d;
		triangle.setBounds(0, 16, 284, 246);
		 label = new JLabel("Robot Orientation");
		label.setBounds(40, 300, 284, 16);

	}

	//METHODS
	public void paint(Graphics g){
		if(display.ADIS.hasMap){
		triangle.paint(g);
		}
	}

}
```

Auger

```
package package1;

import com.neuronrobotics.sdk.dyio.DyIO;
import com.neuronrobotics.sdk.dyio.peripherals.ServoChannel;

//A motor, auger, and drive system
public class Auger {

	//FIELDS
	DyIO dyio; //The DyIO
	int speed; //The set rotational speed for the motor
	int ratio = 1; //The ratio of speed:PWM
	int port; //The port of the dyio this auger uses
	ServoChannel servo; //The Channel for controlling the motor
	int direction;//Whether the auger is a left or a right
	Encoder encoder;
	int encoderPort;
	double divisor=3;



	//CONSTRUCTORS
	Auger (DyIO dyio, int speed, int port, int direction, int encoderPort){
		this.speed = speed;
		this.port = port;
		this.servo = new ServoChannel(dyio.getChannel(port));
		if (Math.abs(direction) != 1)
			System.out.println("Auger direction must be 1 or -1. 
Unexpected results may follow.");
		this.direction = direction;
		this.encoderPort = encoderPort;
		this.encoder = new Encoder(dyio, encoderPort);
	}

	//METHODS


	//instantly sets the speed at which this Auger should turn and 
compensates for auger direction
	public void SetSpeed(int newSpeed){
		if (this.direction == 1) this.speed = newSpeed;
		if(this.direction ==-1) this.speed = 255-newSpeed;
	}

	//sets the speed at which this auger should turn, with acceleration 
control
	public void SetSpeedAcc(int newSpeed){
		int range= 10;//The amount of acceleration control. A higher 
number is less control.

		if(this.direction ==-1) newSpeed = 255-newSpeed;

		if (speed != newSpeed){
			//if the actual speed is less than the set speed range, go 
slightly faster
			if (speed < newSpeed - (range-1)) speed += range;
			//if the actual speed is more than the set speed range, go 
slightly slower
			else if (speed > newSpeed + (range-1)) speed -= range;
			//if the actual speed is in range, set it to the set speed
			else speed = newSpeed;
		}
	}

	//set the motor to move at the given speed
	public void Move(int moveSpeed){
		SetSpeed(moveSpeed);
		Go();
```

```
      }

      //set the motor to move at its set speed
      public void Go(){
            int pwm = speed * ratio;
            if(pwm >252) pwm = 252;//upper PWM limit with the DyIO
            if(pwm < 2) pwm = 2;//Lower PWM limit with the DyIO

            //this scales the maximum and minimum outputs, and centers them
around 127
            pwm =(int)((pwm/divisor)+((divisor-1)/(divisor)*127));

            servo.SetPosition(pwm);
      }


}
```

Clicker

```
package package1;

import java.awt.Point;
import java.awt.event.MouseEvent;
import javax.swing.event.MouseInputAdapter;
import package1.Position;

//A class for accepting inputs on an image
class Clicker extends MouseInputAdapter{
      //FIELDS
      NavMap map;
      Position first = null;
      Position start = null;
      Position lastClicked = null;
      int MAX_DESTINATIONS = 100;
      Position destinations[]=new Position[MAX_DESTINATIONS];
      IceReading path[] = new IceReading[10000];
      int head=0;
      int tail = 0;
      int ppt = 0;

      //CONSTRUCTOR
  public Clicker(NavMap c){
    map = c;
  }

  //METHODS
  public void mousePressed(MouseEvent e){
      if (tail >= MAX_DESTINATIONS) return;//Don't overflow the array of
positions

      Point p = e.getPoint();//Find out where the user clicked

      if(         (p.x < 355)||
                  (p.x>345+map.image.getWidth())||
                  (p.y<45)||
                  (p.y>34+map.image.getHeight())) return;
```

```
            Position clicked = new Position(p.x, p.y);
            if (first == null){
                  first = new Position(p.x,p.y);
                  map.ADIS.display.repaint();
                  //DRAW SOMETHING GREEN AT CLICKED_POS
                  lastClicked = clicked;
                  return;
            }

            if (lastClicked.x==clicked.x && lastClicked.y == clicked.y)
return;
            lastClicked = clicked;
            //MAKE THE POSITION AT CLICKED_POS RED
            destinations[tail] = clicked;
            map.ADIS.display.repaint();
            tail ++;

  }
}
```

Compass

```
package package1;

import com.neuronrobotics.sdk.dyio.DyIO;
import com.neuronrobotics.sdk.dyio.peripherals.AnalogInputChannel;

//The Compass
public class Compass extends AbSensor {

      //FIELDS
      double ratio = 1;//the ratio of voltage:Heading
      int LeftPort;
      int RightPort;
      AnalogInputChannel LeftChannel;
      AnalogInputChannel RightChannel;

      //CONSTRUCTORS
      Compass(DyIO dyio, int LeftPort, int RightPort){
            super();
            this.dyio = dyio;
            this.LeftPort = LeftPort;
            this.RightPort = RightPort;
            this.LeftChannel = new
AnalogInputChannel(dyio.getChannel(LeftPort));
            this.RightChannel = new
AnalogInputChannel(dyio.getChannel(RightPort));
      }


      //METHODS
      //Takes the analog voltage reading of the Compass,
      //converts it to a heading, then updates the proper fields
```

```
	public void ReadSensor (){
		int LeftVal = (int)(LeftChannel.getVoltage()*100);
		int RightVal = (int)(RightChannel.getVoltage()*100);
		LeftVal= (LeftVal-254);
		RightVal = (RightVal - 254);
		LastReading = 180-Math.toDegrees(Math.atan2(LeftVal,RightVal));
		//System.out.print("Before Adjust: "+LastReading);
		LastReading = adjustAngle(LastReading);
		//System.out.println(" After Adjust: "+LastReading);

		UpdateTime();
	}

	//Linearizes the angle, based on a lookup table of recorded difference
between actual and read data
	public double adjustAngle(double angle){
		double diff,slope,offset;
		//Angles that the difference from true angle were recorded for
		double angles[] =
{0,24,39,58,84,115,155,183,202,221,240,258,276,286,301,314,324,342,360};
		//Differences from true angle recorded for the above angles
		double diffs[] = {0,4,-1,-
2,4,15,35,43,42,41,40,38,36,26,21,14,4,2,0};
		int a,b;
		for(a=1;a<19;a++){
			b=a-1;
			if(angle<angles[a]){
				//Once the correct range is found, find the linear
value for difference
				slope = (diffs[a]-diffs[b])/(angles[a]-angles[b]);
				offset = angle-angles[b];
				diff = slope*offset + diffs[b];
				return angle - diff;
			}
		}
		System.out.println("ERROR: Angle not Adjusted: "+angle);
		return angle;

	}






}
```

ControllerController

```
package package1;


import net.java.games.input.*;
```

```
//A class for controling a USB controller
public class ControllerController {
	//FIELDS
	public Controller controller;
	public DirectController dc;
	public JoyStickDialog jsd;

	//CONSTRUCTOR
	public ControllerController(){

		Controller[] ca =
ControllerEnvironment.getDefaultEnvironment().getControllers();


		for(int i =0;i<ca.length;i++){
			/* Get the name of the controller */
			if(ca[i].getType() == Controller.Type.STICK){
				controller = ca[i];
				break;
			}
		}
		dc = new DirectController(controller);
		jsd = new JoyStickDialog(dc);

	}
}
```

DirectController

```
package package1;

import net.java.games.input.*;

//A class for recieving data from a USB PS2 Controller
public class DirectController {
	//FIELDS
	int LeftStickX;
	int LeftStickY;
	int RightStickX;
	int RightStickY;
	Controller device;
	Component[] input;

	//CONSTRUCTOR
	DirectController(Controller device){
		this.device = device;
		this.input = device.getComponents();
	}

	//METHODS
	public void Update(){
		device.poll();

		LeftStickY = (int) (255 -(input[0].getPollData()*127 + 127));
		LeftStickX = (int) (input[1].getPollData()*127 + 127);
```

```
            RightStickX = (int) (input[2].getPollData()*127 + 127);
            RightStickY = (int) (255 - (input[3].getPollData()*127 + 127));
      }


      public void CheckComponents(){
            if(input[0].isAnalog()){
                  System.out.println("Input 0 is Analog");
            }else{
                  System.out.println("Input 0 is Digital");
            }

            if(input[1].isAnalog()){
                  System.out.println("Input 1 is Analog");
            }else{
                  System.out.println("Input 1 is Digital");
            }

            if(input[2].isAnalog()){
                  System.out.println("Input 2 is Analog");
            }else{
                  System.out.println("Input 2 is Digital");
            }

            if(input[3].isAnalog()){
                  System.out.println("Input 3 is Analog");
            }else{
                  System.out.println("Input 3 is Digital");
            }

      }

}
```

Display

```
package package1;

import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JSlider;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.io.IOException;
import java.util.Hashtable;

import javax.swing.JTextPane;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

//A display that contains an output window, various buttons to interact with
the robot
```

```
// and various displays to show the robot's state.
@SuppressWarnings("serial")
public class Display extends JFrame{

      //FIELDS
      Robot ADIS;
      JTextPane text;
      JScrollPane scrollPane;
      boolean paused = false;
      JButton mapButton;
      JButton startButton;
      JSlider speedControl;
      JLabel speedLabel;
      int xSize=300;
      int ySize=300;

      //CONSTRUCTOR
      @SuppressWarnings("unchecked")
      public Display(final Robot x) {

            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            ADIS = x;
            getContentPane().setLayout(null);
            JButton resetButton = new JButton("Reset");
            JButton exitButton = new JButton("Exit");
            mapButton = new JButton("Map");
            text = new JTextPane();
            scrollPane = new JScrollPane(text);
            startButton = new JButton("Start");
            speedControl = new JSlider(JSlider.VERTICAL, 1, 5, 3);
            speedLabel = new JLabel("Speed Scale");

            exitButton.setBounds(190,11, 70, 23);
            resetButton.setBounds(106, 59, 70, 23);
            scrollPane.setBounds(10, 93, 264, 151);
            mapButton.setBounds(21, 11, 70,23);
            startButton.setBounds(106,11,70,23);
            speedControl.setBounds(280,60,60,200);
            speedLabel.setBounds(260,40,80,20);

            getContentPane().add(exitButton);
            getContentPane().add(resetButton);
            getContentPane().add(scrollPane);
            getContentPane().add(mapButton);
            getContentPane().add(startButton);


            speedControl.addChangeListener(new ChangeListener(){
                  public void stateChanged(ChangeEvent e){
                        int value = 6 - speedControl.getValue();
                        x.LeftAuger.divisor = value;
                        x.RightAuger.divisor = value;
                  }
            });
            exitButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){
                        x.Stop();
                        try {
                              Thread.sleep(100);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        System.exit(1);
                  }
            });
            resetButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent arg0) {
                        ADIS.Posn = null;
                        ADIS.nm.clicker.destinations = new
Position[ADIS.nm.clicker.MAX_DESTINATIONS];
                        ADIS.nm.clicker.path = new IceReading[1000];
                        ADIS.display.repaint();
                        ADIS.nm.clicker.head = 0;
                        ADIS.nm.clicker.tail = 0;
                        ADIS.nm.clicker.ppt = 0;
                        System.out.println("Resetting");
                        ADIS.getStartBlock();


                  }
            });
            mapButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){

                        try {
                              x.chooseMap();
                        } catch (IOException e) {
                              println("Cannot Choose Map");
                              e.printStackTrace();
                        }
                        x.hasMap = true;
                        speedControl.setPaintLabels(true);
                        getContentPane().add(speedControl);
                  }
            });
            startButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){
                        x.start = true;
                  }
            });


            text.setEditable(false);
            Hashtable labelTable = new Hashtable();
            labelTable.put( new Integer( 1 ), new JLabel("1/5th") );
            labelTable.put( new Integer( 5), new JLabel("Full") );
            speedControl.setLabelTable( labelTable );
            speedControl.setMajorTickSpacing(1);

            this.setBounds(50, 50, xSize, ySize);
            setVisible(true);
      }
```

```
	//METHODS

	//print the given string to the console, and this display's output 
window
	void print(String t){
		System.out.print(t);
		text.setText(text.getText()+t);
		
	scrollPane.getVerticalScrollBar().setValue(scrollPane.getVerticalScroll
Bar().getMaximum());
		try {
			Thread.sleep(10);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	//print the given string with a line break at the end
	void println(String t){
		print(t+"\n");
	}

	//print the given int
	public void print(int t){
		print(Integer.toString(t));
	}

	//print the given int with a line break at the end
	public void println(int t){
		print(Integer.toString(t)+"\n");
	}

	public void paint(Graphics g){
		paintComponents(g);
		if(ADIS.nm != null)ADIS.nm.paint(g);
		if(ADIS.ad != null)	ADIS.ad.paint(g);
		
	}

}//end class
```

Encoder

```
package package1;

import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JSlider;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.io.IOException;
import java.util.Hashtable;
import javax.swing.JTextPane;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

//A display that contains an output window, various buttons to interact with
the robot
// and various displays to show the robot's state.
@SuppressWarnings("serial")
public class Display extends JFrame{

      //FIELDS
      Robot ADIS;
      JTextPane text;
      JScrollPane scrollPane;
      boolean paused = false;
      JButton mapButton;
      JButton startButton;
      JSlider speedControl;
      JLabel speedLabel;
      int xSize=300;
      int ySize=300;

      //CONSTRUCTOR
      @SuppressWarnings("unchecked")
      public Display(final Robot x) {

            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            ADIS = x;
            getContentPane().setLayout(null);
            JButton resetButton = new JButton("Reset");
            JButton exitButton = new JButton("Exit");
            mapButton = new JButton("Map");
            text = new JTextPane();
            scrollPane = new JScrollPane(text);
            startButton = new JButton("Start");
            speedControl = new JSlider(JSlider.VERTICAL, 1, 5, 3);
            speedLabel = new JLabel("Speed Scale");

            exitButton.setBounds(190,11, 70, 23);
            resetButton.setBounds(106, 59, 70, 23);
            scrollPane.setBounds(10, 93, 264, 151);
            mapButton.setBounds(21, 11, 70,23);
            startButton.setBounds(106,11,70,23);
            speedControl.setBounds(280,60,60,200);
            speedLabel.setBounds(260,40,80,20);

            getContentPane().add(exitButton);
            getContentPane().add(resetButton);
            getContentPane().add(scrollPane);
            getContentPane().add(mapButton);
            getContentPane().add(startButton);


            speedControl.addChangeListener(new ChangeListener(){
                  public void stateChanged(ChangeEvent e){
                        int value = 6 - speedControl.getValue();
                        x.LeftAuger.divisor = value;
                        x.RightAuger.divisor = value;
                  }
            });
            exitButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){
                        x.Stop();
                        try {
                              Thread.sleep(100);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        System.exit(1);
                  }
            });
            resetButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent arg0) {
                        ADIS.Posn = null;
                        ADIS.nm.clicker.destinations = new
Position[ADIS.nm.clicker.MAX_DESTINATIONS];
                        ADIS.nm.clicker.path = new IceReading[1000];
                        ADIS.display.repaint();
                        ADIS.nm.clicker.head = 0;
                        ADIS.nm.clicker.tail = 0;
                        ADIS.nm.clicker.ppt = 0;
                        System.out.println("Resetting");
                        ADIS.getStartBlock();


                  }
            });
            mapButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){

                        try {
                              x.chooseMap();
                        } catch (IOException e) {
                              println("Cannot Choose Map");
                              e.printStackTrace();
                        }
                        x.hasMap = true;
                        speedControl.setPaintLabels(true);
                        getContentPane().add(speedControl);
                  }
            });
            startButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent arg0){
                        x.start = true;
                  }
            });


            text.setEditable(false);
            Hashtable labelTable = new Hashtable();
            labelTable.put( new Integer( 1 ), new JLabel("1/5th") );
            labelTable.put( new Integer( 5), new JLabel("Full") );
            speedControl.setLabelTable( labelTable );
            speedControl.setMajorTickSpacing(1);

            this.setBounds(50, 50, xSize, ySize);
            setVisible(true);
      }


      //METHODS

      //print the given string to the console, and this display's output
window
      void print(String t){
            System.out.print(t);
            text.setText(text.getText()+t);

      scrollPane.getVerticalScrollBar().setValue(scrollPane.getVerticalScroll
Bar().getMaximum());
            try {
                  Thread.sleep(10);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }

      //print the given string with a line break at the end
      void println(String t){
            print(t+"\n");
      }

      //print the given int
      public void print(int t){
            print(Integer.toString(t));
      }

      //print the given int with a line break at the end
      public void println(int t){
            print(Integer.toString(t)+"\n");
      }

      public void paint(Graphics g){
            paintComponents(g);
            if(ADIS.nm != null)ADIS.nm.paint(g);
            if(ADIS.ad != null)     ADIS.ad.paint(g);

      }

}//end class
```

Gyro

```
package package1;

import com.neuronrobotics.sdk.dyio.DyIO;

//The Gyro
```

```
public class Gyro extends AbSensor {

	//FIELDS
	double voltageOffset;//
	double ratio = .8;//the ratio of Angular Acceleration:Voltage
(1.76470588)
	public double angle = 0;//The angle that the Gyro is at


	//CONSTRUCTORS
	Gyro(DyIO dyio, int Port){
		super(dyio, Port);
	}


	//METHODS


	//Takes the analog voltage reading of the gyro,
	//converts it to an angular acceleration, then updates the proper
fields
	public void ReadSensor(){
		double voltage = LastReading;// voltage in volts
		voltage = voltage - voltageOffset;//center the voltage around
zero
		voltage = voltage * 100;//voltage in milivolts
		double angVel = -voltage * ratio;//anglular velocity in a
clockwise direction
		if (-1<angVel && angVel<1) angVel = 0;
		double timeChange = System.currentTimeMillis() - TimeTaken;
		double degMoved = angVel* (timeChange);
		degMoved = degMoved/(15*14);
		if(-.2<degMoved && degMoved<.2)degMoved = 0;
		angle = angle +degMoved;

		LastReading = AnInput.getVoltage();//get the reading for the next
iteration
		TimeTaken = System.currentTimeMillis();//get the time the reading
was taken
	}
}
```

IceReading

```
package package1;

import java.awt.Color;

//A class representing a reading of ice thickness
public class IceReading {
	//FIELDS
	Position position = new Position();//The position of the reading
	double thickness;//Thickness of the ice read at the position
	Color color;//Color representing the thickness
	//Variables representing the levels of safety of different ice-
thicknesses in inches
```

```
	int unsafe = 1;//less than this is unsafe
	int dangerous = 3;//less than this is dangerous
	int questionable = 5;//less than this is questionable
	//otherwise, the ice is safe.

	//CONSTRUCTORS
	public IceReading(){
		position.x=0;
		position.y=0;
		thickness=0;
		color= Color.black;
	}
	public IceReading(double x, double y, double thickness){
		this.position.x = x;
		this.position.y = y;
		this.thickness = thickness;
		setColor();
	}
	public IceReading(Robot ADIS){
		position.x = ADIS.Posn.x;
		position.y = ADIS.Posn.y;
		thickness = ADIS.GetIceThickness();
		setColor();
	}

	//METHODS
	public void setColor(){
		if (thickness < unsafe) color = Color.black;
		else if (thickness < dangerous) color = Color.red;
		else if (thickness < questionable) color = Color.yellow;
		else color = Color.green;
	}
}
```

ISensor

```
package package1;

//An interface for Sensors
public interface ISensor {
	void UpdateTime();
	void ReadSensor();
	double GetReading();
	long GetReadingTime();
	int GetPort();

}
```

JoyStickDialog

```
package package1;

import javax.swing.JDialog;
import javax.swing.JLabel;
import java.awt.Color;
import java.awt.Font;

//A class to display information from a USB controller
@SuppressWarnings("serial")
public class JoyStickDialog extends JDialog {
      //FIELDS
      DirectController sticks;
      JLabel LX, LY, RX, RY, Right, Left, X, Y;

      //CONSTRUCTOR
      public JoyStickDialog(DirectController sticks){
            this.sticks = sticks;

            setBounds(100, 100, 254, 209);
            getContentPane().setLayout(null);

            LX = new JLabel("0");
            LX.setBounds(48, 60, 46, 14);
            getContentPane().add(LX);

            RX = new JLabel("0");
            RX.setBounds(146, 60, 46, 14);
            getContentPane().add(RX);

            LY = new JLabel("0");
            LY.setBounds(48, 110, 46, 14);
            getContentPane().add(LY);

            RY = new JLabel("0");
            RY.setBounds(146, 110, 46, 14);
            getContentPane().add(RY);

            Left = new JLabel("Left-Stick");
            Left.setFont(new Font("Tahoma", Font.BOLD, 11));
            Left.setForeground(Color.RED);
            Left.setBounds(32, 33, 62, 14);
            getContentPane().add(Left);

            Right = new JLabel("Right-Stick");
            Right.setFont(new Font("Tahoma", Font.BOLD, 11));
            Right.setForeground(Color.RED);
            Right.setBounds(124, 33, 68, 14);
            getContentPane().add(Right);

            X = new JLabel("X");
            X.setFont(new Font("Tahoma", Font.BOLD, 11));
            X.setForeground(Color.RED);
            X.setBounds(12, 60, 26, 14);
            getContentPane().add(X);

            Y = new JLabel("Y");
            Y.setFont(new Font("Tahoma", Font.BOLD, 11));
            Y.setForeground(Color.RED);
            Y.setBounds(12, 110, 17, 14);
            getContentPane().add(Y);
            setVisible(true);
      }

      //METHODS
      public void Update(){
            sticks.Update();

            LX.setText(Integer.toString(sticks.LeftStickX));
            LY.setText(Integer.toString(sticks.LeftStickY));
            RX.setText(Integer.toString(sticks.RightStickX));
            RY.setText(Integer.toString(sticks.RightStickY));
            repaint();
      }
}
```

Main

```
package package1;


import java.io.IOException;
import com.neuronrobotics.sdk.dyio.DyIO;
import com.neuronrobotics.sdk.ui.ConnectionDialog;


public class Main {
      public static void main(String[] args) throws InterruptedException,
IOException{
            //Set up the DyIO
            DyIO dyio = new DyIO();
            if (!ConnectionDialog.getBowlerDevice(dyio)){
                  System.exit(1);
            }
            //Create the robot
            Robot ADIS = new Robot(dyio);

            ADIS.RestartTime();//a function for sensor timing

            /*//Optional code for controling the robot with the PS2
controller.
             *simply remove the block comment to use this functionality.

            //Create the PS2 Controller Controller
            ControllerController cc = new ControllerController();
            boolean what = true;
            ADIS.hasMap = true;
            ADIS.start = true;
            while(what){
                  ADIS.ps2drive(cc);
                  Thread.sleep(10);
                  }
            //*/

            //wait for a map to be selected
            ADIS.display.println("Waiting for Map Selection");
            while(!ADIS.hasMap);

            ADIS.getStartBlock();
            ADIS.Gyro.ReadSensor();
            ADIS.Gyro.angle = 180;

            while(true){
                  ADIS.mapBlock();
            }//end while
      }//end main
}//end class
```

MapChooser

```
package package1;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileNameExtensionFilter;

//A class for choosing an image file to represent a map
@SuppressWarnings("serial")
public class MapChooser extends JFrame{
      //FIELDS
      JFileChooser fc;
      File file;
      BufferedImage map;

      //CONSTRUCTOR
      public MapChooser() throws IOException{
            fc = new JFileChooser("src/Maps");
            FileFilter ff = new FileNameExtensionFilter("Image Filter",
"png","jpg");
            fc.setFileFilter(ff);
            int retreval;
            retreval = fc.showDialog(this, "Choose Map");
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            if(retreval == JFileChooser.APPROVE_OPTION){
                  file = fc.getSelectedFile();
                  System.out.println(file.getName());
                  map = ImageIO.read(file);
                  //Robot x = null;
                  //NavMap nm = new NavMap(x, map);
                  //nm.repaint();
            }
            else System.exit(1);;

      }
}
```

NavMap

```
package package1;


import java.awt.*;
import java.awt.image.*;

//A class for managing a map, components on the map, and a clicker to accept
input from the map
public class NavMap{
	//FIELDS
	Robot ADIS;
	BufferedImage image;
	Dimension size;
	int x_pos;
	int y_pos;
	Graphics2D g2;
	Clicker clicker = new Clicker(this);
	int SIZE=4;

	//CONSTRUCTOR
	public NavMap(Robot ADIS, BufferedImage image)
	{
		this.image = image;
		size = new Dimension(image.getWidth(), image.getHeight());

		this.ADIS = ADIS;

		ADIS.display.addMouseListener(clicker);
		ADIS.display.addMouseMotionListener(clicker);
	}

	//METHODS
	public void paint(Graphics g){
		g2 = (Graphics2D)g;
		g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
				RenderingHints.VALUE_ANTIALIAS_ON);
		g2.drawImage(image, 350, 40, ADIS.display);

		//if (clicker.first != null){
		//	g2.setColor(Color.yellow);
		//	g2.fillRect((int)((clicker.first.x)-.5*SIZE),
(int)((clicker.first.y)-.5*SIZE), SIZE, SIZE);

		//}
		int i;
		if(clicker.destinations[clicker.head] !=null){
			g2.setColor(Color.blue);
			for(i=clicker.head;i<clicker.tail;i++){
				g2.fillRect((int)(clicker.destinations[i].x-.5*SIZE),
(int)(clicker.destinations[i].y-.5*SIZE), SIZE,SIZE);
			}
		}
```

```
            if(ADIS.Posn != null){
                  g2.setColor(Color.green);
                  //g2.fillRect((int)(ADIS.Posn.x-.5*SIZE),
(int)(ADIS.Posn.y-.5*SIZE), SIZE,SIZE);
                  double angle = (ADIS.Posn.angle);
                  double xx = ADIS.Posn.x;
                  double yy = ADIS.Posn.y;
                  double L = 20;
                  angle=Math.toRadians(angle);
                  //do the math only once, to avoid bogging down the system
                  double sin = Math.sin(angle);
                  double cos = Math.cos(angle);
                  //left point
                  double x1 = xx+((L/4)*cos);
                  double y1 = yy+((L/4)*sin);
                  //right point
                  double x2 = xx-((L/4)*cos);
                  double y2 = yy-((L/4)*sin);
                  //top
                  double x3 = xx+(L*sin);
                  double y3 = yy-(L*cos);

                  int xPts[] = {(int)x1,(int)x2,(int)x3};
                  int yPts[] = {(int)y1,(int)y2,(int)y3};
                  g2.fillPolygon(xPts, yPts, 3);

            }
            for(i=2;i<clicker.ppt;i++){
                  g2.setColor(clicker.path[i].color);
                  g2.fillRect((int)(clicker.path[i].position.x-.5*SIZE),
(int)(clicker.path[i].position.y-.5*SIZE), SIZE,SIZE);
                  //g2.setColor(Color.black);
                  //g2.drawLine((int)clicker.path[i].x,
(int)clicker.path[i].y, (int)clicker.path[i-1].x, (int)clicker.path[i-1].y);

            }

      }

}
```

Position

```
package package1;

//A location and orientation
public class Position {

      //FIELDS
      public double x;
      public double y;
      double angle;


      //CONSTRUCTORS
      Position(double x_pos, double y_pos, double angle){
            this.x = x_pos;
            this.y = y_pos;
            this.angle = angle;
      }
      public Position(double x_pos, double y_pos){
            this.x = x_pos;
            this.y = y_pos;
            this.angle = 0;
      }
      Position(){
            this(0,0,0);
      }

}
```

Robot

```
package package1;

import com.neuronrobotics.sdk.dyio.DyIO;
import java.io.IOException;
import java.lang.Math;
import java.util.Random;


//A robot
public class Robot{
      //FIELDS
      DyIO dyio;

      Auger LeftAuger;
      Auger RightAuger;

      Compass Compass;
      AbSensor IceSensor;
      Gyro Gyro;

      Position Posn;
      Position Target;

      boolean hasMap = false;
      boolean start = false;

      double thickness = 3;//A fake initial thickness. See "GetIceThickness"

      MapChooser mc;
      NavMap nm;
      AngleDialog ad;//The dialog box that displays the relative position and
orientation of the robot
      Display display;


      //CONSTRUCTOR
      Robot(DyIO dyio) throws IOException, InterruptedException{
            this.dyio = dyio;
            this.LeftAuger = new Auger(dyio, 127, 2, -1, 21);
            this.RightAuger = new Auger(dyio, 127, 3, 1, 23);

            this.Gyro = new Gyro(dyio,12);
            this.IceSensor = null;
            this.Compass = new Compass(dyio, 10, 11);

            this.Posn = null;

            display = new Display(this);
            ad = new AngleDialog(display);//The dialog box that displays the 
relative position and orientation of the robot

            //Find the offset of the Gyro
            double voltages = 0;
            Gyro.ReadSensor();
            for (int i=0;i<5;i++){
                  Gyro.ReadSensor();
                  voltages += Gyro.LastReading;
                  Thread.sleep(5);
            }
            Gyro.voltageOffset = voltages/5;



      }

      //METHODS

      //choose and display the map
      public void chooseMap() throws IOException{
            mc = new MapChooser();
            nm = new NavMap(this, mc.map);

            display.getContentPane().add(ad.label);
            display.getContentPane().add(ad.triangle);
            display.getContentPane().add(display.speedLabel);

            display.getContentPane().remove(display.mapButton);

            //Resize the Display to account for new components
            display.xSize = 370 + nm.image.getWidth();
            if((60+nm.image.getHeight())<600)display.ySize = 600;
            else display.ySize = 60 + nm.image.getHeight();
            display.setSize(display.xSize,display.ySize);

            display.repaint();
      }

      //stops the robot
      public void Stop(){
            for (int i=0;i<10;i++){
                  LeftAuger.SetSpeedAcc(127);
                  RightAuger.SetSpeedAcc(127);
                  LeftAuger.Go();
                  RightAuger.Go();
```

```
		}
	}

	//sets both motors to the same speed
	public void Move(int speed){
		LeftAuger.Move(speed);
		RightAuger.Move(speed);
	}

	//set the angle the triangle shoudl be facing, and repaint the display
	public void UpdateDialog(){
		ad.triangle.angle= Posn.angle;
		display.repaint();
	}

	//drives both augers at the given speed, using proportional control.
	//Must be called in a loop.
	public void driveStraight(int setSpeed){
		drive(setSpeed, 0);

	}//end driveStraight

	//block that sets the motors to turn the given amount of degrees 
clockwise
	public void TurnBlock(int degrees) throws InterruptedException{

		display.println("Target Angle: "+ degrees);

		int Kp=5;
		int Ki = 0;
		int Kd = 900;
		double error = 0;//a positive error means turn clockwise
		double errorSum=0;
		double errorPrev =0;
		double rate = 0;
		int speed = 0;
		int done = 3;
		long time = System.currentTimeMillis();
		long timeChange = 0;
		double prevAngle=0;

		Gyro.ReadSensor();//Read the sensor, to reset the time since last 
reading
		Gyro.angle = 0;//Reset the angle, so that turn angle is based of 
current heading
		while (done > 0){
			error = degrees - (int)Gyro.angle;
			System.out.println(error);
			errorSum += error*timeChange;
			rate = error - errorPrev;
			timeChange = System.currentTimeMillis()-time;
			speed = (int) ((Kp*error) + (Ki*errorSum) + 
(Kd*rate/timeChange));
			LeftAuger.SetSpeedAcc(127 + speed);//to turn clockwise, 
increase this
			RightAuger.SetSpeedAcc(127 - speed);//to turn clockwise, 
decrease this
```

```
                  LeftAuger.Go();   RightAuger.Go();
                  errorPrev = error;//Set previous error for next iteration
                  time = System.currentTimeMillis();
                  Thread.sleep(20);
                  Gyro.ReadSensor();//to update the angle
                  UpdateAngle(Gyro.angle-prevAngle);//Add the degrees
traveled to the current heading of the robot
                  UpdateDialog();
                  display.repaint();
                  if ((Gyro.angle >= degrees - 5)&&(Gyro.angle<=degrees +
5)){
                        display.println("Done in "+done);
                        done--;
                  }
                  else {
                        prevAngle = Gyro.angle;
                        done = 3;
                  }
            }//end while
            Stop();
            display.println("DONE");
            //Verify with Compass
            display.println("Veryifying with Compass...");
            Thread.sleep(3000);
            Compass.ReadSensor();
            if (Posn.angle>Compass.LastReading-10 &&
Posn.angle<Compass.LastReading+10){
                  display.println("Verified");
            }
            else {
                  display.println("Not Verified. Heading Adjusted.");
            }
      }

      //block that sets the motor to drive straight the given distance in
inches
      public void DriveStraightBlock(double distance) throws
InterruptedException{
            double ticks = getNumTicks(distance);//Find the number of ticks
equivalent to the given distance
            double avg=0;//the average of the encoder values
            double distTraveled = 0;//the total distance, in inches, the
robot has traveled so far
            double lastDist = 0;//The distance the robot had traveled the
last iteration through the loop
            double Kp = 6;//Proportional Constant that scales the value
            double Ki = 0;//Integral Constant
            double Kd = 600;//Derivative Constant
            double error=0;
            double errorSum = 0;
            double errorPrev = 0;
            double rate = 0;
            int speed=0;
            int done = 3;
            long time = System.currentTimeMillis();
            long timeChange = 0;
            LeftAuger.encoder.channel.setValue(0);
            RightAuger.encoder.channel.setValue(0);
            Gyro.angle = 0;
            while(done>0){
                  error = (ticks - avg);
                  errorSum += (error*timeChange);
                  rate = error - errorPrev;
                  timeChange = System.currentTimeMillis()-time;
                  //Move the motors slower based on error
                  speed = 127;
                  speed += (Kp*error) + (Ki*errorSum) + (Kd*rate/timeChange);
                  driveStraight(speed);
                  errorPrev = error;
                  time = System.currentTimeMillis();
                  avg = -(-
LeftAuger.encoder.channel.getValue()+RightAuger.encoder.channel.getValue())/2
;
                  distTraveled = getDistance(avg);
                  UpdatePosition(distTraveled- lastDist);
                  lastDist = distTraveled;
                  display.repaint();
                  if ((avg >= ticks - 5)&&(avg<=ticks+5)){
                        display.println("Done in "+Integer.toString(done));
                        done--;
                  }
                  else {
                        done = 3;
                  }

            }
            //slow the motors to a complete stop
            Stop();
      }

      //converts distance in inches to a number of encoder ticks
      public double getNumTicks(double distance){
            return distance*2;//the robot moves a half-inch for every tick
      }

      //converts number of encoder ticks to a distance in inches
      public double getDistance(double ticks){
            return ticks/2;//the robot moves a half-inch for every tick
      }

      //restart the time, as if the robot has just begun taking sensor
readings
      public void RestartTime(){
            long time = System.currentTimeMillis();
            Gyro.TimeTaken = time;
            Compass.TimeTaken = time;
      }

      //drive the robot with the ps2 controller
      public void ps2drive(ControllerController cc){
            int Left = (cc.dc.LeftStickY);
            int Right= (cc.dc.RightStickY);

            //if the set speed is close to 127, set it to 127
            if (112 < Left && Left < 142) Left = 127;
            if (112 < Right && Right < 142) Right = 127;
            System.out.println("Start the motors moving at "+Right+" and 
"+Left);
            RightAuger.Move(Right);
            LeftAuger.Move(Left);
            cc.jsd.Update();//Update the Dialog box for the PS2 Controller

      }

      //Calculates the new angle of the robot from a given change in angle
      public void UpdateAngle(double angle){
            Posn.angle += angle;
            while ((Posn.angle>360)||(Posn.angle<0)){
                  if (Posn.angle > 360)Posn.angle -= 360;
                  if (Posn.angle < 0)Posn.angle +=360;
            }
      }

      //calculates the new x/y position of the robot, from the given distance 
traveled in inches
      public void UpdatePosition(double distance){
            double angle = Posn.angle;
            angle = Math.toRadians(angle);
            double x_change= (Math.sin(angle)*distance);
            double y_change= (Math.cos(angle)*distance);
            Posn.x += x_change;
            Posn.y -= y_change;


            nm.clicker.ppt++;
            Position current = new Position();
            current.x = Posn.x;
            current.y = Posn.y;
            nm.clicker.path[nm.clicker.ppt] = new IceReading(this);
            display.repaint();
      }

      //waits for a button to be pressed on the map, then updates the target 
position
      public void UpdateTargetBlock() throws InterruptedException{
            if (nm.clicker.head == nm.clicker.tail){
                  nm.clicker.head = 0;
                  nm.clicker.tail = 0;
                  display.println("Waiting for input");
                  while (nm.clicker.head == nm.clicker.tail){
                        Thread.sleep(10);
                  }
            }

            Target = nm.clicker.destinations[nm.clicker.head];

      }

      //calculates the distance between the current and target positions
      public int CalculateDistance(){
            int distance;
            double x, x2, y, y2;
            x = (Target.x - Posn.x);
            x2 = x*x;
            y = (Target.y - Posn.y);
            y2 = y*y;
            distance = (int) (Math.sqrt((x2+y2)));
            return distance;
      }

      //calculates the angle change needed to get to the target position
      //returns an angle from -180 to 180
      public double CalculateAngle(){
            double angleGlobal, angleChange, x, y;
            x = (int)Target.x - (int)Posn.x;
            y = (int)Target.y - (int)Posn.y;
            //first, find the global angle from the current position to the
target
            angleGlobal = Math.toDegrees(Math.atan(x/y));
            if (y<0){
                  if (x<0) angleGlobal= -180+angleGlobal;
                  if (x>0) angleGlobal = 180+angleGlobal;
                  if (x==0)angleGlobal = 180;
            }
            angleGlobal = 180-angleGlobal;//Translates this angle to a zero-
up, positive-clockwise angle
            //now, find the angle the robot needs to turn, based on it's
current heading
            angleChange = angleGlobal - Posn.angle;
            //adjust the number to a number between +/-180
            while ((angleChange>180)||(angleChange<-180)){
                  if(angleChange > 180)angleChange -= 360;
                  if(angleChange <-180) angleChange +=360;
            }
            return angleChange;
      }

      //waits for the user to input a starting position
      public void getStartBlock(){
            while(Posn == null){
                  display.println("Select Starting Position");
                  while(nm.clicker.first == null);
                  Compass.ReadSensor();
                  Posn = new Position(nm.clicker.first.x,nm.clicker.first.y,
0);

                  nm.clicker.path[0] = new IceReading(this);
                  display.println("Start Position Set:
("+Posn.x+","+Posn.y+"){"+(int)Posn.angle+"}");
            }
            display.repaint();
      }

      //drive the robot according to instructions given from the GUI
      public void mapBlock() throws InterruptedException{
            getStartBlock();
            if(!start)display.println("Preload any map points, then press
\"Start\".");
            while(!start);
            UpdateTargetBlock();

            double angleChange =CalculateAngle();

            display.println("Turning...");
            TurnBlock((int)angleChange);//*dir
            display.println("Success!");

            UpdateDialog();
            display.repaint();

            int distance = CalculateDistance();

            display.println("Driving...");
            DriveStraightBlock(distance);
            display.println("Success!");

            nm.clicker.head ++;
            display.repaint();
      }

      //draw the map, assuming the robot would actually turn and drive the
correct distance
      public void feauxMapBlock() throws InterruptedException{
            int i;
            getStartBlock();
            if(!start)display.println("Preload any map points, then press
\"Start\".");
            while(!start);
            UpdateTargetBlock();
            double angleChange =CalculateAngle();
            int dir;
            if (angleChange>0) dir=1;else dir = -1;
            for(i=0;i!=(int)angleChange;i+=dir){
                  UpdateAngle(dir);
                  UpdateDialog();
                  Thread.sleep(5);
                  display.repaint();
            }
            int distance = CalculateDistance();
            for(i=0;i<distance;i+=10){
                  UpdatePosition(10);
                  Thread.sleep(100);
            }
            nm.clicker.head ++;
            display.repaint();
      }

      //drive the robot the given speed, with the given turning factor (zero
is straight, positive is clock-wise)
      public void drive(int setSpeed, int turnFactor){
            double leftTicks;
            double rightTicks;
            int angle;
            int error;
            //get the ticks from each encoder
            leftTicks = -LeftAuger.encoder.channel.getValue();
            rightTicks = RightAuger.encoder.channel.getValue();
            //a positive error means the right auger has been moving too fast

            error = (int) (rightTicks - leftTicks);

            //Assume the gyro was initially set to 0
            Gyro.ReadSensor();
            display.print("Error: "+error);
            angle = (int) (Gyro.angle);//if the robot has turned clockwise,
subtract more from the error, to turn counter-clockwise
            display.println(", angle"+angle);

            //set the corrected set speed for each auger
            if (setSpeed >= 127){//forwards
                  LeftAuger.SetSpeedAcc(setSpeed + (error) + turnFactor);//a
positive error speed this up
                  RightAuger.SetSpeedAcc(setSpeed - (error) - turnFactor);//
a positive error slows this down
            }
            else {//backwards
                  LeftAuger.SetSpeedAcc(setSpeed - error - turnFactor);//a
positive error slows this down
                  RightAuger.SetSpeedAcc(setSpeed + error + turnFactor);// a
positive error speeds this up
            }
            //Start the augers driving at the corrected speed.
            LeftAuger.Go();
            RightAuger.Go();
      }

      //get the thickness of the ice at the current position
      public double GetIceThickness(){

            //THIS IS FAKE. IT IS A TEMPORARY FIX SO THAT THE MAP SHOWS SOME
SORT OF ICE READING.
            //ONCE AN ICE SENSOR IS ADDED, THIS SHOULD BE CHANGED TO READING
FROM THAT SENSOR.
            Random r = new Random();
            int adjust = r.nextInt(3)-1;
            thickness = thickness + (adjust);
            if (thickness < 0)thickness = 1;
            if (thickness > 7)thickness = 6;
            //END OF THE FAKE PART

            return thickness;

      }

}//end class
```

Triangle

```
package package1;
```

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JComponent;

//A class that draws a triangle
@SuppressWarnings("serial")
public class Triangle extends JComponent{
	//FIELDS
	double x = 150;//The X coordinate around which the triangle is centered
	double y = 450;//The Y coordinate around which the triangle is centered
	double L = 100;//the height of the triangle
	double angle = 0;//the angle at which the triangle is drawn, in degrees
	Graphics2D g2g;//some kind of graphics thing. It's necessary.

	//CONSTRUCTOR
	//empty constructor
	public Triangle() {

	}

	//METHODS
	//paint the triangle
	public void paint(Graphics g){
		g2g = (Graphics2D)g;
		g2g.setColor(Color.black);
		//g2g.drawLine(150, 0, 150, 300);
		//g2g.drawLine(0, 150, 300, 150);
		angle=Math.toRadians(angle);
		//do the math only once, to avoid bogging down the system
		double sin = Math.sin(angle);
		double cos = Math.cos(angle);

		//left point
		double x1 = x+((L/4)*cos);
		double y1 = y+((L/4)*sin);
		//right point
		double x2 = x-((L/4)*cos);
		double y2 = y-((L/4)*sin);
		//top
		double x3 = x+(L*sin);
		double y3 = y-(L*cos);

		angle = Math.toDegrees(angle);


		//draw the lines between the points
		g2g.drawLine((int)x1, (int)y1, (int)x2, (int)y2);
		g2g.drawLine((int)x2, (int)y2, (int)x3, (int)y3);
		g2g.drawLine((int)x3, (int)y3, (int)x1, (int)y1);


	}
}
```