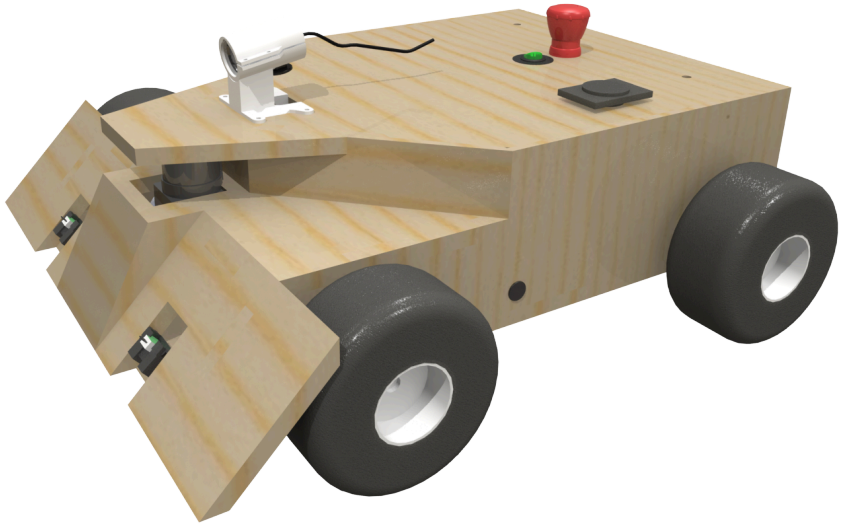
ENGR 3390 Final Project



Maximilian Schommer
12/12/2016

Table of Contents:

List of Figures:

Executive Summary

We designed, fabricated and programed an autonomous ground robot to navigate around obstacles, follow waypoints, and respond to visual cues given by the environment.

We designed a low profile, stable chassis using plywood as the primary structural component. We maximized our track width and our wheelbase in order to insure stable sensor measurements. At the same time, we minimized our height in order to lower the overall mass of the robot, and to maximize the amount of data our LIDAR could collect from our environment.

We controlled our behavioural lights, motors, and ultrasonic and IR sensors with an Arduino. We enabled our Emergency Stop to stop our motors, yet leave the computing uninterrupted, though notified. For the majority of our computing power, we used an Odroid ux4. We controlled our LIDAR, camera, IMU, and gps with the odroid.

Overall Design

Mechanical:

The robot had four major constraints that determined how it would be built:

1. The robot could not exceed the dimensions of a 2x2’ box
2. The robot needs to pass a 45 degree tilt test
3. The robot needs to protect the internals in the case of a roll (Most importantly, the LIDAR)
4. The robot can not harm other robots or beings.

The robot needed to pass each of these constraints in order for it to qualify in the races. We decided to settle on a low profile robot, in order to keep our center of mass as low as possible, with four wheels and motors to drive the robot. The shape of the robot was primarily determined by the LIDAR, as well as the battery. Since the LIDAR has a 240 degree field of vision, we wanted to use as much of the data as possible. In addition the battery was the largest component of the robot, so our height was determined by the minimum size of the battery. With this information, we built our robot to have a “shelf” on the front, that would taper off to get the maximum field of view for the LIDAR.

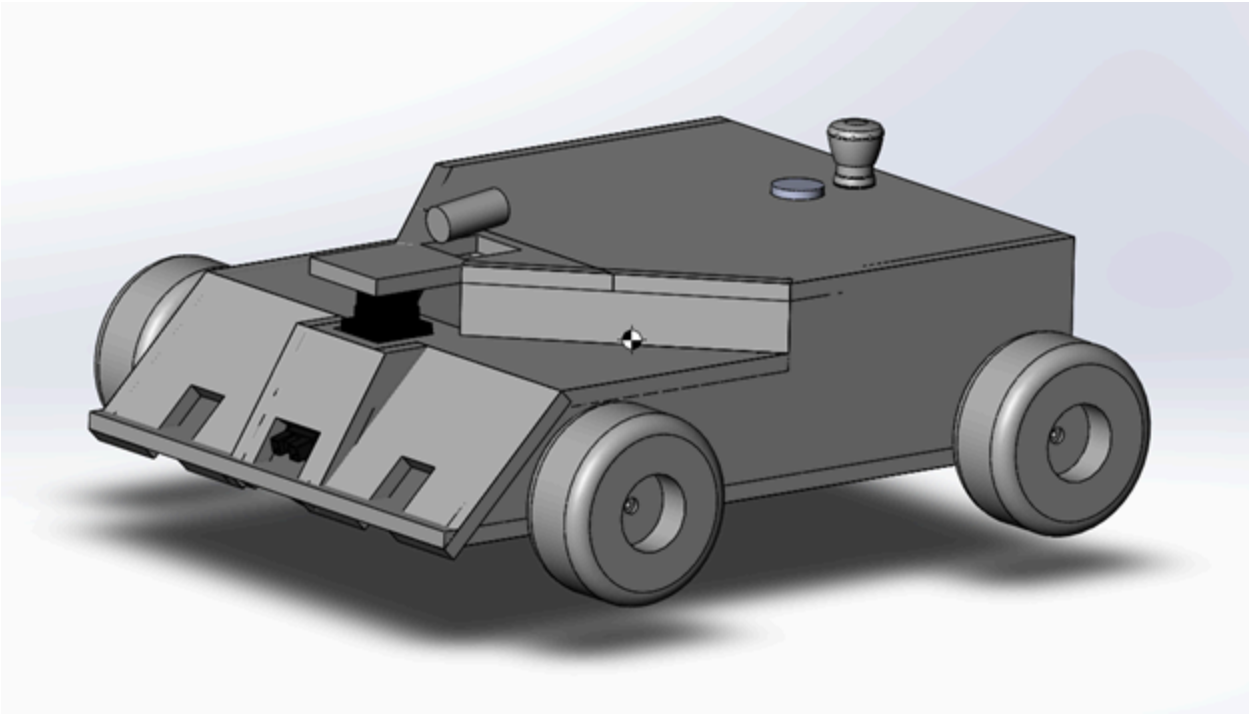


Fig 1. Overall CAD design of robot

Our hull was created out of ½” plywood, which would give us the strength to withstand the constant beatings of many programming failures. We created finger joints, glued, and filled each piece of the plywood to ensure the robot was weather proof.

Electrical:

We had various components to sense, process, and drive our robot.

Our sensors included:

Hokuyo LIDAR

The Hokuyo LIDAR (Light Detection and Ranging) URG-04LX-UG01 emits laser pulses every 100ms, measuring distances up to 6m in a 240 degree planar field of vision with +/- 30mm accuracy.

GPS

The GPS used was ameridroid usb gps module. It calculates positioning, and outputs NMEA-0183 formatted GPS messages.

INS (Inertial Navigation System)

Phidgets Precision 3/3/3 High Resolution (3 axis compass, gyroscope, accelerometer) provided spatial and motion data as our robot moved around the Olin O.

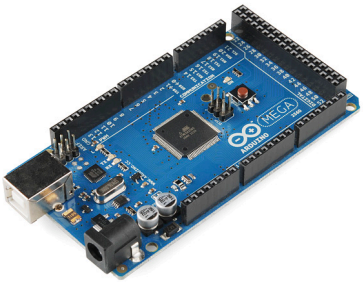
Camera

Microsoft Lifecam Cinema 720p Web camera

This devices captures video and images.

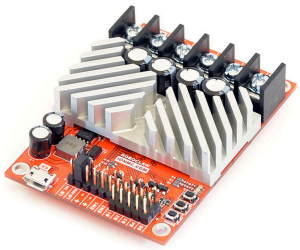
Our processors included:

Arduino Mega 2560



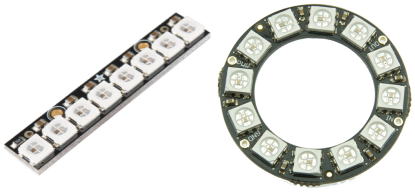
We did the majority of our processing on an Arduino Mega 2560. It has 54 digital input/output pins, 16 analog inputs, 4 UARTs, a 16 MHz crystal oscillator, and a USB power jack.Of its 54 digital pins 15 can be used as PWM outputs. We used these pins to send PWM signals to our motor controller. We also used the analog inputs to check if voltage coming through the e stop switch.

Roboclaw



Our motor controller was a 2x15A RoboClaw, in which the 2 signifies the number of channels, and the 15 signifies the number of amps each channel of the motor controller can handle.

Lights (Neopixel, green/red indicating lights)



E-stop

Battery

Motors

IR Sensors

In order to ensure that we can test everything correctly, we set up all our electronics roadkill style. We placed all our electronics on a board to individually check each component is working as intended.

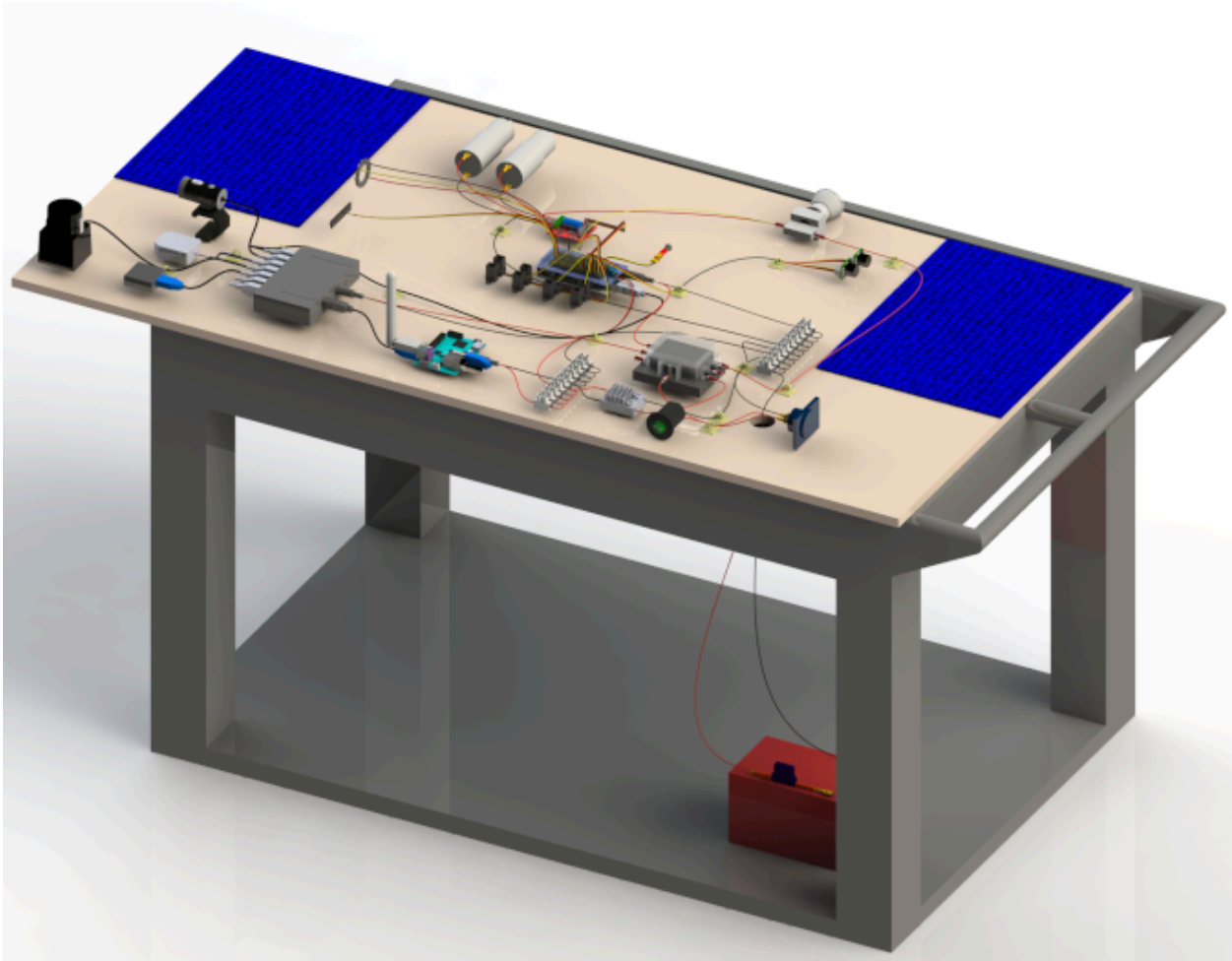


Fig 2. CAD of finished Roadkill assembly

One major safety feature on our roadkill was our fuse block, and inline fuse. The inline fuse protects the battery in the severe case of a short circuit that occurs with the main 12V power line.

The fuse block isolates several different components in a similar case of over-current draw. The motor controller, 12V usb hub, 5V DC-DC converter, and Arduino Mega each have their own over-current fuse.

We had moved all the parts of the electrical roadkill into the robot, with difficulties coming in managing the accessibility and location of parts within the chassis. Some major points of the layout is the focus on having wires be bundled together around the walls of the robot when possible in order to increase debuggability, and positioning electronics on the ceiling of the robot in order to take advantage of the vertical space more effectively.

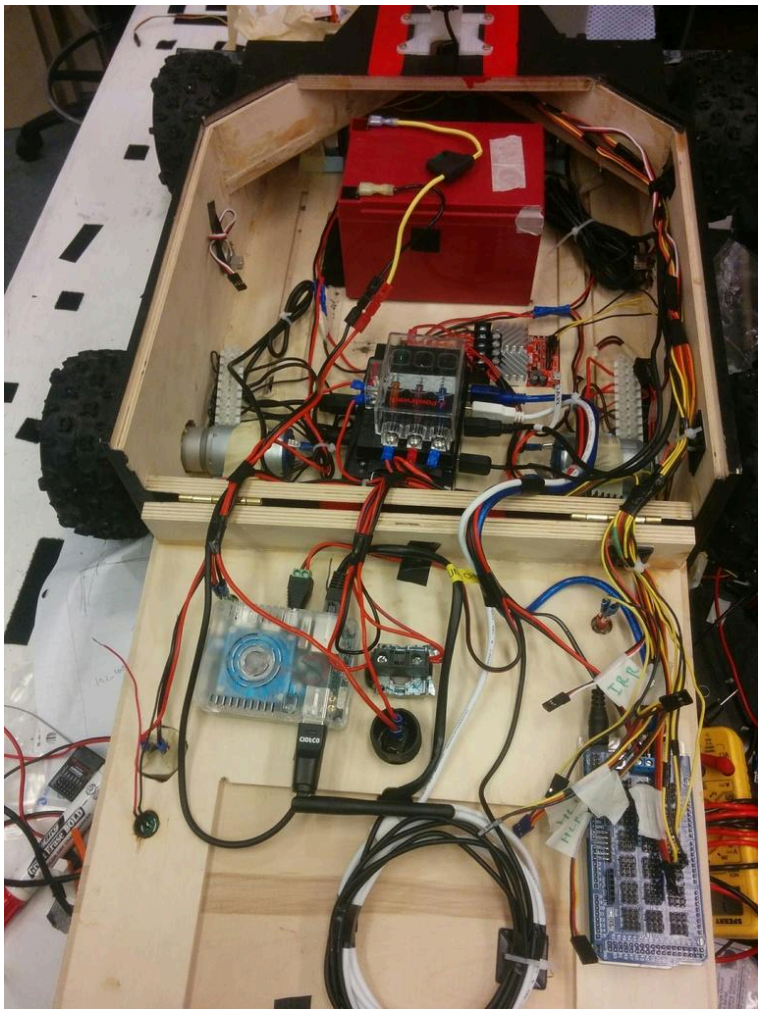


Figure 3: The electrical layout of our robot.

Code:

HINDBRAIN: MOTOR CONTROL, IR, BEHAVIOR LIGHTS, E STOP

The hindbrain of our robot is the software that controls low-level functions, such as lights, emergency stops, infrared sensing, and motor control. It exists entirely on the Arduino Mega 2560. For controlling the lights, we used the Adafruit Neopixel library. The Neopixel Library messes with the normal servo library, so we had to use Adafruit's special "TiCoServo Library." For the emergency stop control, the Arduino read a pin connected to the estop, and turned off motors only if the button was pressed. The IR sensors acted as an estop, and used basically the same code: when a certain value was returned from the sensor, the arduino would trigger an estop status and stop the robot. For the motor, we read a value from a ros topic and then converted those linear and angular velocities into speeds for the left and right motors.

MIDBRAIN: CAMERA, LIDAR, ARBITER

Our camera system had used a Microsoft Lifecam Cinema 720p for our camera. We had placed this camera in the front and elevated part of the robot in order to be able to use it to detect objects that are outside the range of our other sensors. We had interfaced with the node via the use of stock ROS nodes such as uvc_camera. An example of the output of the camera is in the figure below.

Figure 4. Picture of what the robot sees from code.

We had used this camera to do image processing such as recognition of obstacles such as orange cones. This was done via generally looking for objects of orange color, and trying to go between those objects.

Figure 5: The red circle represents desired heading. The boxes are detected cones.

Another part of the system was the use of a Phidget 3/3/3 IMU in order to get inertial measurements of the system. This was primarily used to get another set of data that could be

used to determine which way we were going, and generally trying to push the robot to go left, as per the curvature of the course. We had processed the IMU raw data via the use of a imu_filter_madgewick node that had come as part of the ROS library of nodes. This was used to derive a heading that could be fed into the arbiter as another input.

The LIDAR was utilized for object detection and general pathing around the “O”. We created three nodes using the LIDAR’s raw sensor data. One was used for obstacle avoidance. In this node, we created an algorithm that would direct the robot towards the path of the least resistance. Another node we created was a proportional wall following program. This program assisted us in our general path around the wall by trying to maintain a defined distance away from the wall around the “O”. The final node we created using the LIDAR assisted during the gaps in the wall around the “O”. This node found edges and assisted the robot in keeping its current heading to get past the gaps.

The arbiter is essentially a voting system used to help the robot make decisions on where it should go based on sensor input. The arbiter subscribes to the topics published by our midbrain nodes that format the sensor’s input. Theses midbrain nodes publish 22 element arrays that give a suggested velocity and direction they think the robot should travel based on sensor input. Each 22 element array is weighted according to how immediately the robot needs to react. For example, our obstacle avoidance was weighted much higher than our cone detection. We did this to ensure that the robot would still avoid objects while on its course to the cone.

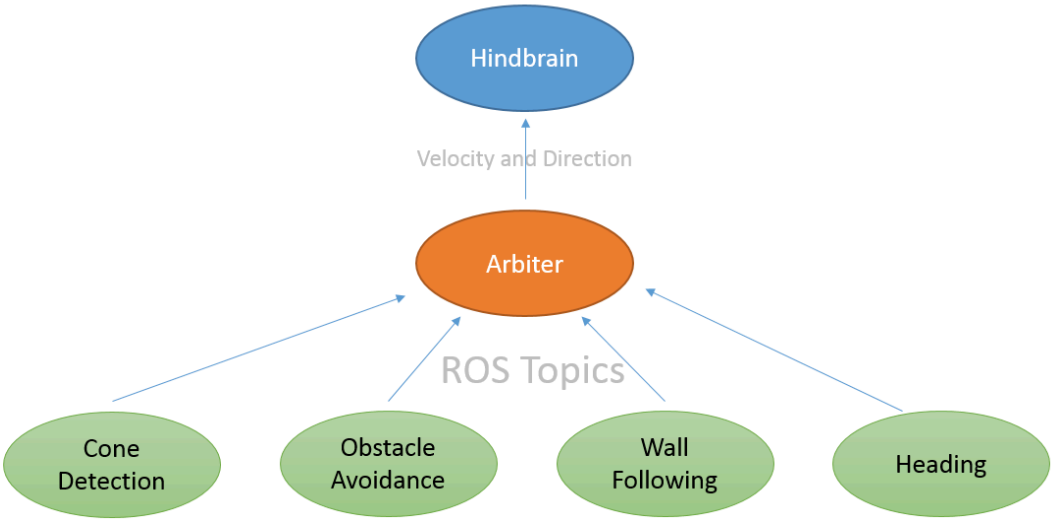


Fig 6. Software Overall Architecture

Detailed Design

Structure:

Our first task was coming up with a concept for our robot. I proposed a triangular base with swiveling caster wheel would be an optimal design for several reasons. First, it would eliminate the problem of wheel slip for high radius turns, and would thus allow us to much more accurately capture odometry data from our motors. It would also allow us to have a much more aesthetically pleasing design, as we would be able to use the roughly triangular shape to create a teardrop using plywood for the majority of structure, and 3d printed surfaces in order create the shell. We had potential problems with the center of mass, and weren't able to CAD a mock up in time for the design review, so we pivoted to a simple four wheel robot with a transmission.

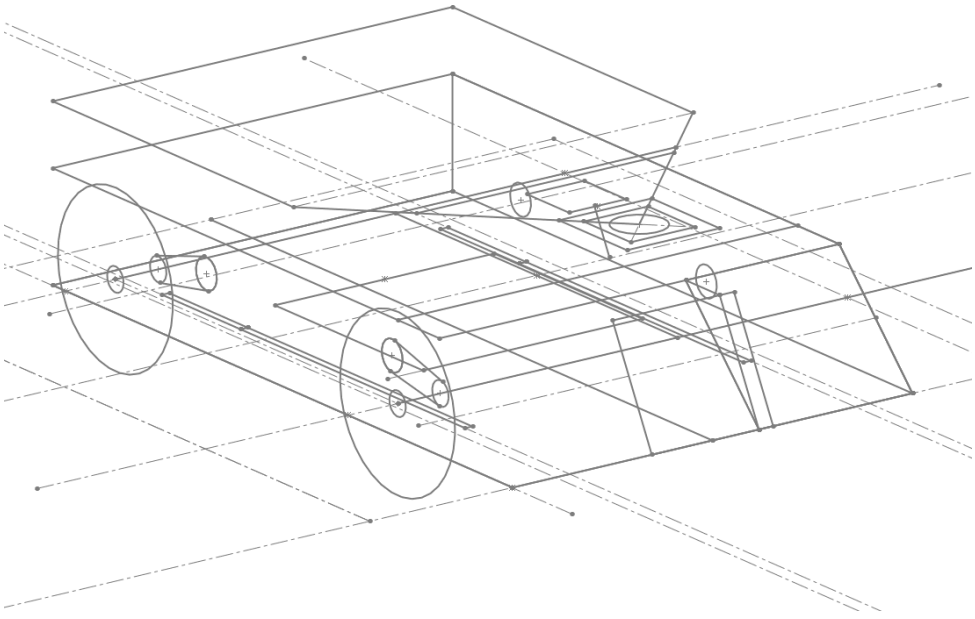


Fig 7. Master Sketch of our Robot

I designed a master sketch (Fig. 7) that contained all of the references needed to design the chassis. This part was then inserted into any new hull component, and thus everything was designed to update on this single sketch. I then designed the hull, including all joints, mostly finger joints. We decided on a structure that would have a mostly open top, so that the components would be accessible. We didn't realize how difficult servicing (Fig. 8) the front two motors would because of our low profile front design.

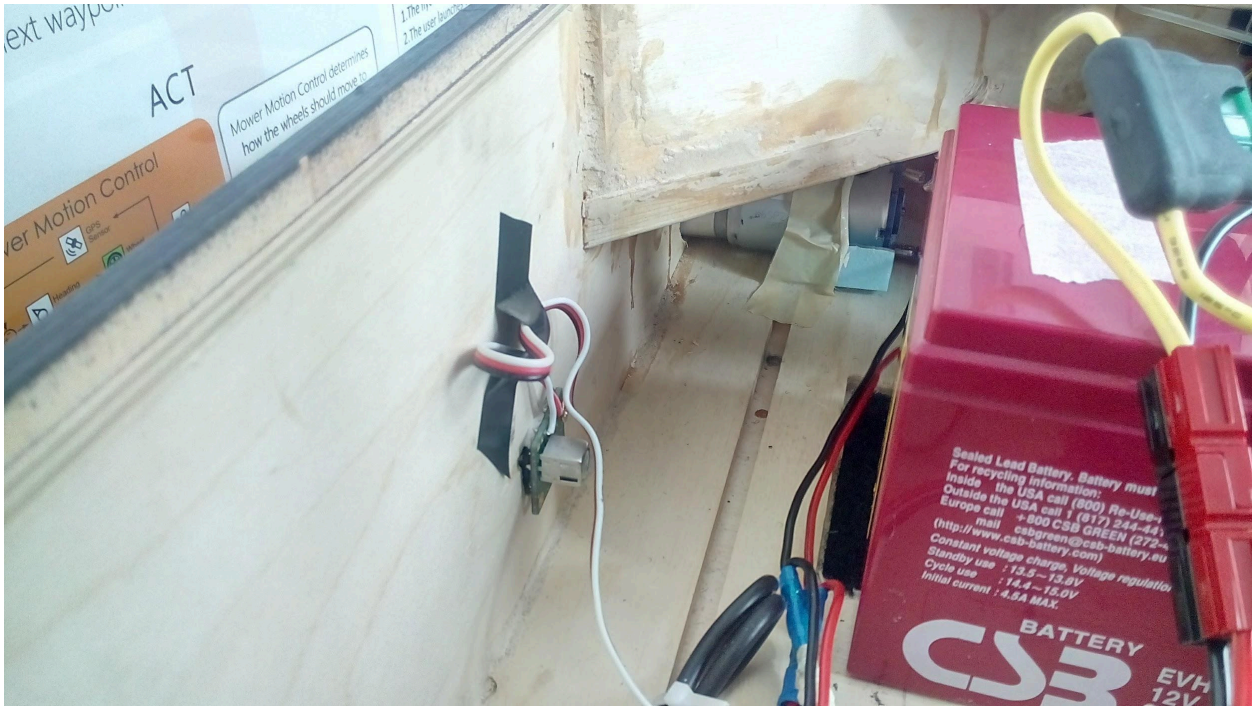


Fig 8. Difficult to service front motors.

I then began to CAD individual components for the robot, including the on-off switch, the Camera mount (Fig. 9), the wheels, wheel-reinforcement units etc. We intentionally didn't design a tilt for our LIDAR because we thought we wouldn't have enough time to program functionality that utilized the complexity.

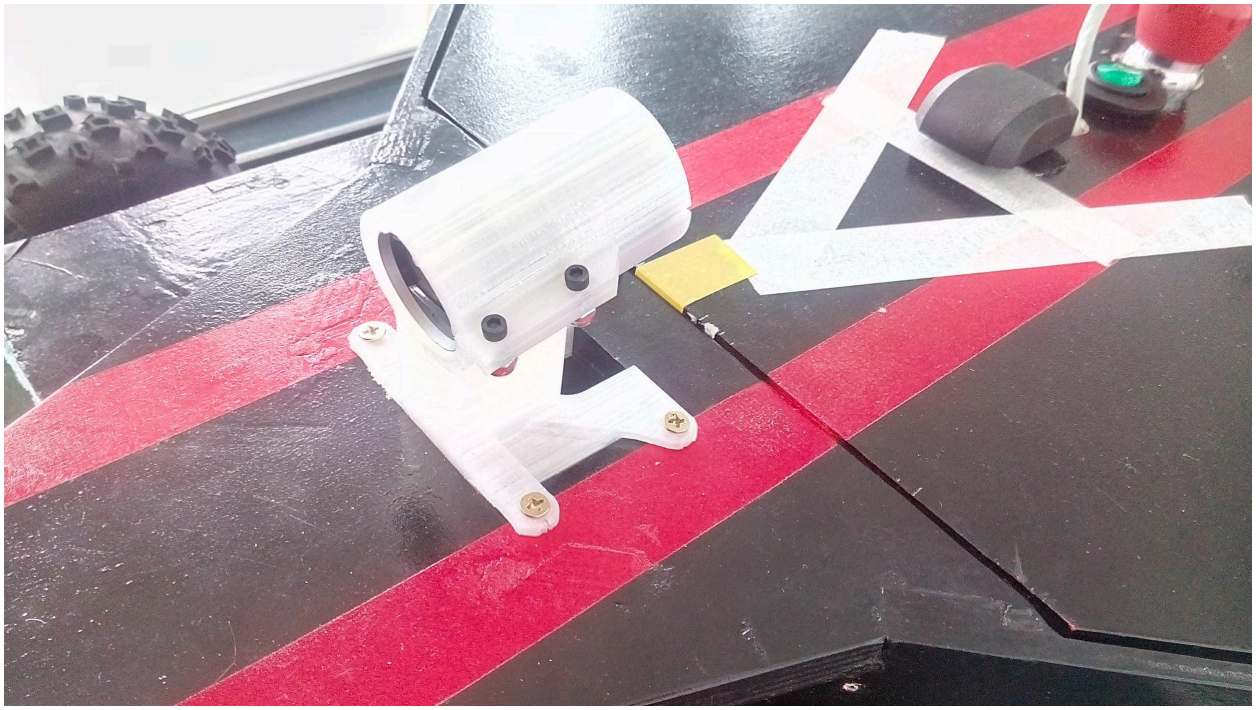


Fig 9. Camera mounting system

After the first race, our motors failed. I suspected that we shock-loaded our motors by not having a chain tensioning system. I figured that a 1.5x load on the motors shouldn't be enough to make them fail like they did, or we would start seeing every team's motors fail (Fig. 10). We decided to cut the current transmission into parts, and tension by sliding the parts apart, retained by a 3d printed spacer. After several attempts to get the right tolerances on the printed part, we successfully built the rear transmission again. When we used the remaining two motors on the rear transmission, we noticed before we put them in that they sounded broken. We decided to try to test with them anyway, since they would fail soon regardless of whether we put them on the transmission. The motors failed, and since we were now approaching the software

portion of the robot, we decided to switch to 1:1 drive so we didn't have as many mechanical problems to deal with.

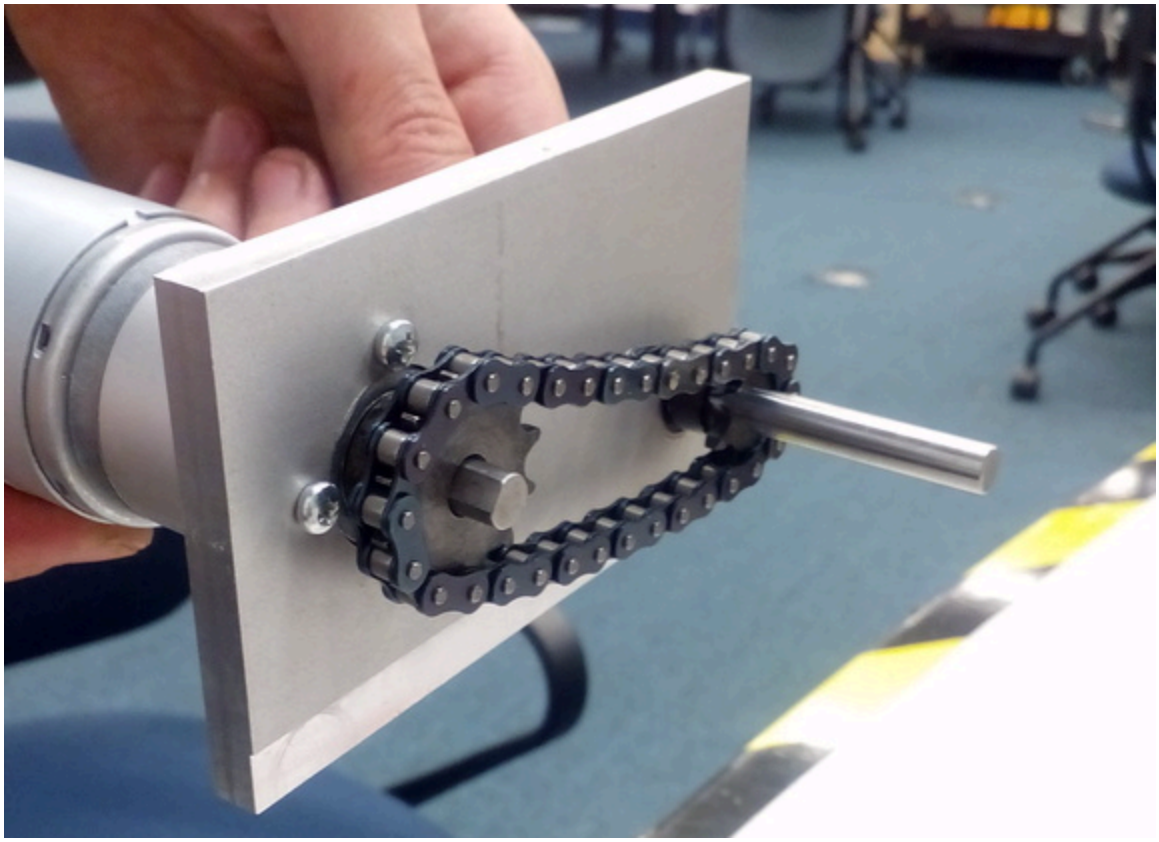


Fig 10. Initial transmission design

I designed a new 1:1 drive (Fig. 11) that could be implemented in our robot, which consisted of a sheet metal plate with mounting holes that mounted on the outside of our robot. We then dremeled out holes for the motors, and mounted them directly to the sheet metal plate.

Fig 11. 1:1 wheel drive system

We wired our electrical components and began figuring out how to use the various sensors we had available. I helped write the IMU integration script, which utilized the magnetic sensors to gather the orientation of the IMU in order to subtract the acceleration of gravity from our two dimensional data. We noticed that when using the IMU by itself, we got a significant amount of drift somewhat quickly. We ended up not using this as a standalone odometry unit, but instead used the compass functionality to get our robot to move to waypoints.

Software:

Next I began to work on the camera. I wanted to have an intelligent robot based on vision, and so decide that the best way of having our robot learn about navigating the

course would be to design a convolutional neural network that worked on the images from the camera and the associated direction vector being assigned. In order to get the direction vector, we created ros bags with the camera input and a remote control input. We then proceeded to gather data by driving the robot indoors, and later by driving outside.

The first thing I had to do was choose a network architecture. I decided on one that was used to classify MNIST digits using a recurrent network (Fig. 12).

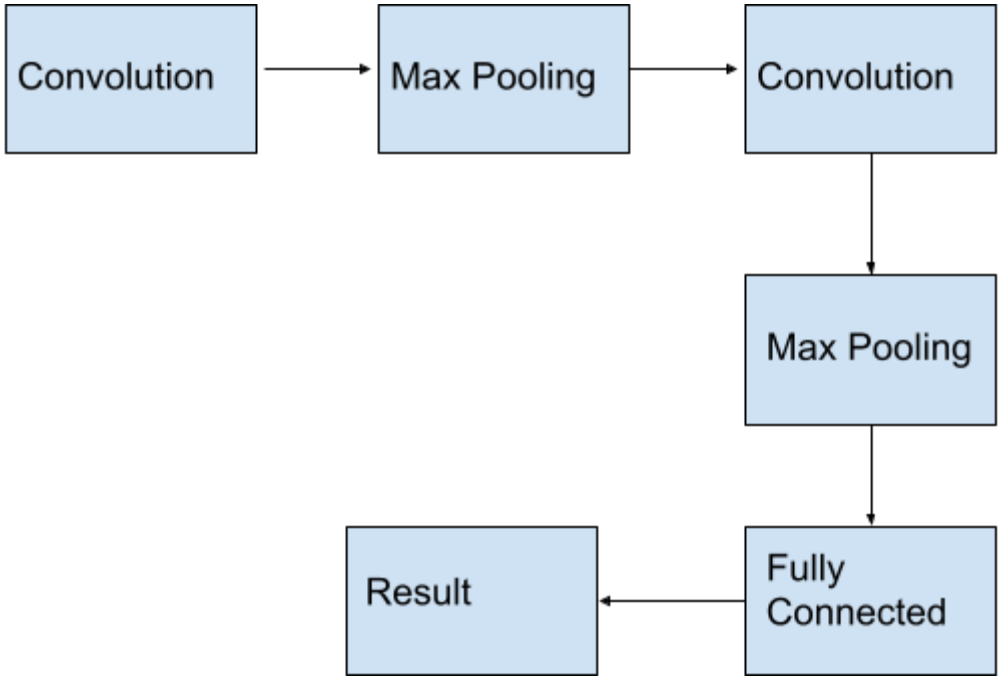


Fig 12. Network Architecture

I wrote a convolution function that would convolve any matrix on an image (Fig. 13). I used an edge detection matrix to display the functionality. The network would choose its own values for its convolution matrices based on gradient descent. I divided the output of the network into 10 classes, each one corresponding to a representative heading vector for the robot.

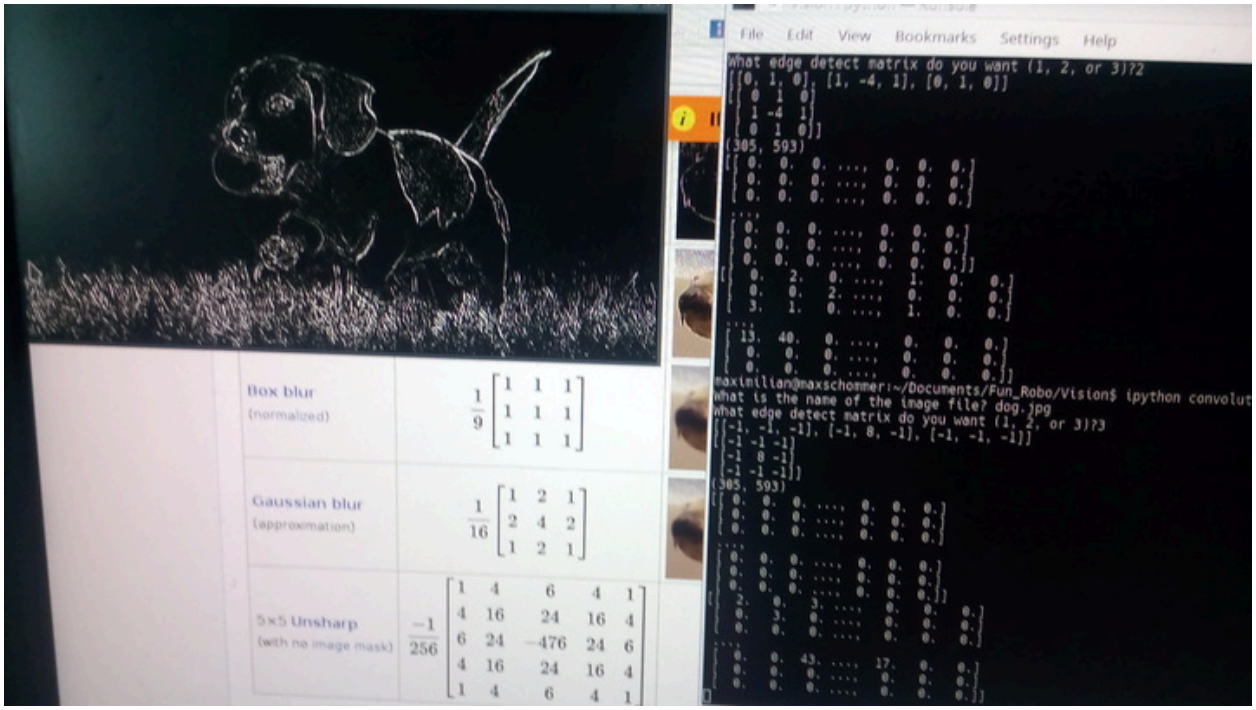


Fig. 13. Convolved matrix over an arbitrary image

In order to test the network properly, data must be divided into training and test sets. The training set is used to train the network, and the test set is used to evaluate it accurately with inputs the network never saw before. I chose to use 80% of the data we collected for training, and 20% for testing. I randomly sampled the overall dataset in order to divide the data without structure that could result from ordering. I was able to eventually get a 71% accuracy (Fig. 14) on never-seen images when looking at our indoor training set of around 900 images and vector headings. The accuracy of this was dubious, because in the training set, most of the images were associated with the default steering control (i.e. going straight). Because most of the control of the robot is in reality done in a few frames, not continuously, it would be very difficult to get the

network to not simply return the most likely result (because it wasn't able to find structure in the data since so much of it was noise).

Fig. 14 Neural network trained to an accuracy of 71% on test data.

After this network was working, I then decided to make the network more realistic by turning it into a regression network, as opposed to a classification network. This meant that the output would be a continuous range from -1 to 1 instead of a discrete one-hot vector that classifies 10 classes.

Converting to a regression model turned out to be much more difficult than anticipated, and one of the problems I ran into was that my learning rate was around 5 orders of magnitude too small. I also didn't normalize my data, which caused further rounding errors that made my model untrainable. After the bugs were worked out, I tested the model while looking at the images from the corresponding ROS bag, and found, much to my disappointment, that the network did overfit the data, and didn't learn high enough level features to understand when to make control decisions.

Since the network failed, I programmed a simpler algorithm that would take the center of mass of all of the orange cones, and aim for that. This would allow it to move between cones, weighed by their distance from the robot. Distance was measured based on the horizon, and the further a cone was away, the more it was considered since the program was meant for long term planning.

Race Summary

Before the race began, we were still tweaking some parameters. We had several nodes, including a wall following node that took a slice of LIDAR to the left and used that to approximate where the wall was. We had an edge detection node that looked at the LIDAR data and tried to find vertical edges. We had an obstacle avoidance node that moved the car based on what side an obstacle was. We had a cone detection node that used centers of mass of cones in order to weigh what direction to move. None of these nodes should have been telling the robot to swerve to the right severely, and yet the robot would do this consistently when we ran the arbiter with all of the nodes active.

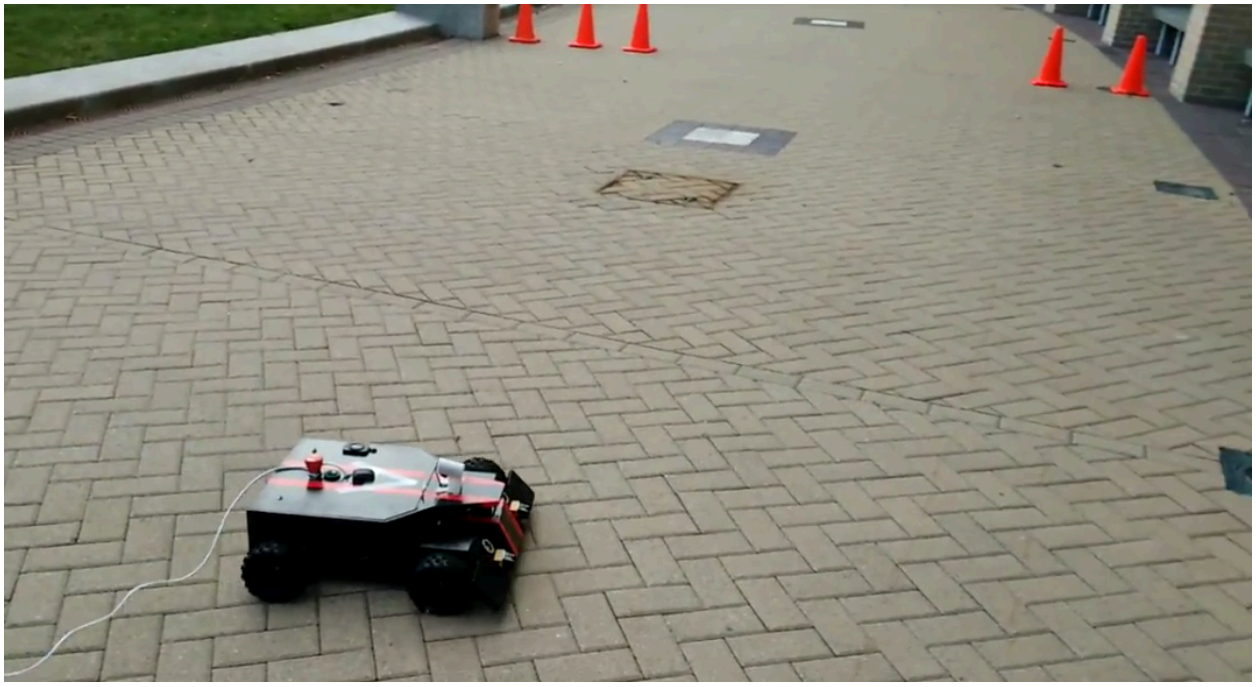


Figure 15: Alphabot swerving to the right and away from the intended path

Our first race went fairly well, mostly by chance. It so happened that the robot saw things that it normally didn't, which caused it to steer back on course and finish all twenty pillars. However, there were definitely moments when the robot almost got stuck. We noticed that the wall follow or the edge detect wasn't functioning properly because the robot would hit a wall if it approached it from a narrow enough angle. This might cause it to veer significantly to the right if it then suddenly detects wall.

Figure 16: Alphabot at the starting line for its first autonomous race.

Our second race went as expected. The robot failed to see the curb, and then suddenly saw something, swerved severely to the right, and then turned around when it went inside the pillars. We believe that the cause of this mysterious blindness was that the curb was beneath the LIDAR, so whenever the grass would come into view of the LIDAR, the wall follow or edge detect would activate suddenly and cause the robot to overshoot.

We failed the third race in a similar manner.

Figure 17: Alphabot is a drama-bot and refuses to participate in the race.

Individual Summary

The chassis design had the advantage of stability and speed. Because of our low profile, we were able to push cones out of the way if need be. The serviceability of the design had a ways to go. When replacing the transmission, we initially had to embark on a several hour disassembly process of removing sprockets and inserting spring pins. The addition of the 1:1 drive made replacing the motors a 5 minute process, and we ended up having to replace two motors during the final race.

Fig. 18. The sun gear on the motor is sheared off.

The teeth on the sun gear sheared (Fig. 18) off during regular use (without a transmission). The process of the gears weakening contributed to building up error in

our expected motor output versus our actual motor output, and resulted in a discrepancy that caused us partially to overcompensate using code.

The chassis was stable, and had a center of gravity well within tilt range (Fig. 19). This stability helped with all of the sensor information gathering in the race.

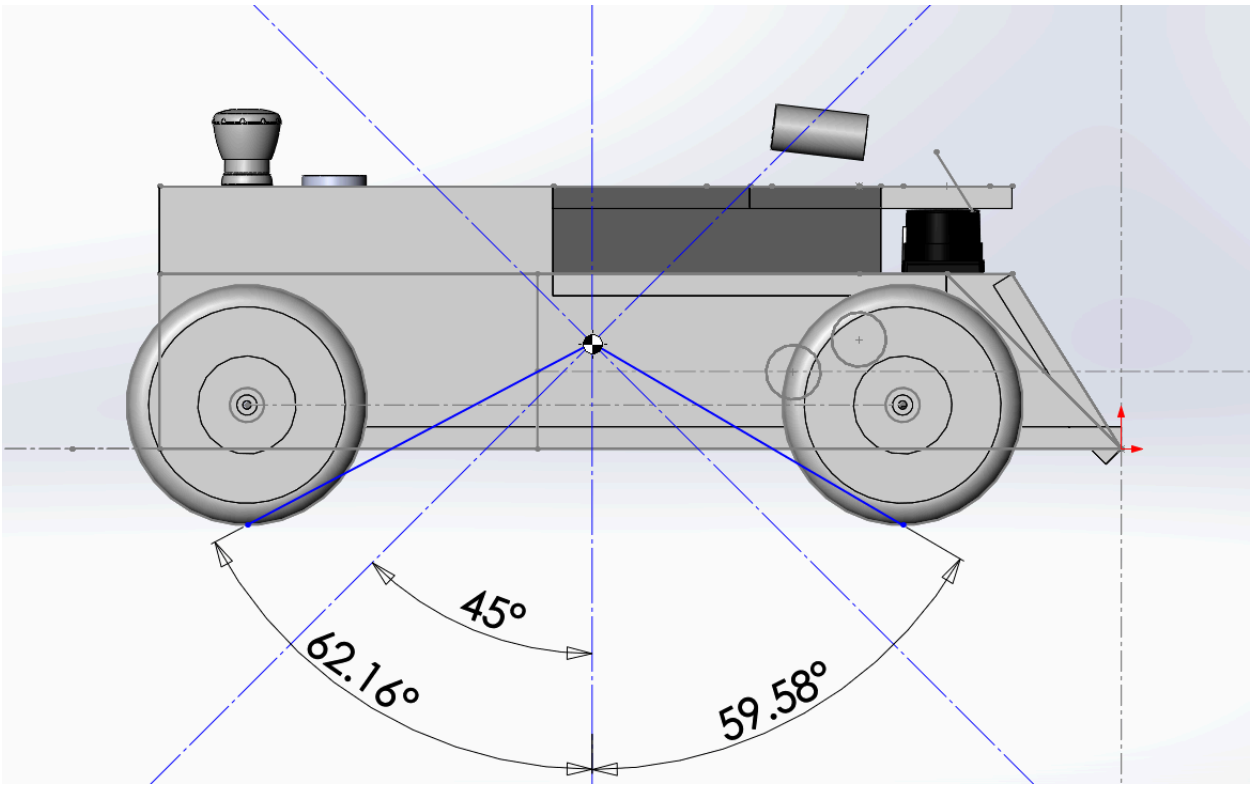


Fig. 19. CG test

The neural network ultimately failed, and didn't contribute anything to helping the robot. This is largely due to lack of experience with neural networks, and a lack of time to get such a high level learning algorithm perfected with such difficult data. We acquired several gigabytes of data, and can use that in the future for testing learning algorithms, but the dataset is far from ideal.

The cone detecting software, however, performed admirably (Fig. 20). We used OpenCV to detect bounding rectangles of orange objects using HSV color isolation. We then took the areas of the boxes, and set that equal to the mass of each object. The center of the rectangle was the center of the object. By taking the center of mass of all of the objects on screen, we were able to ignore small irregularities, and create a feature that told the arbiter to aim in the middle and to the left of cones if it saw cones to the left. If it saw a cone to the right, it should ignore it. The reason that some cones were ignored was because if our robot started predominantly listening to the cone detecting node, then it would aim for the large orange cones to the side, and not follow the wall correctly. We tested the cone software on its own, and it would indeed move directly between cones if visible, and at a cone if only one existed.

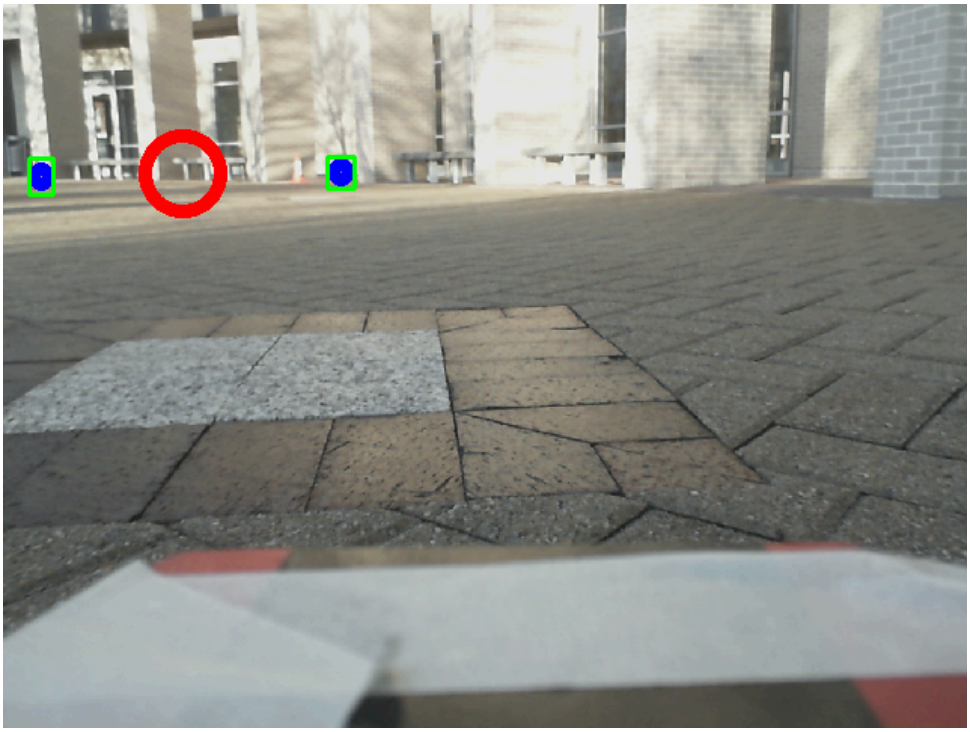


Fig. 20. Cone detection finds the center between cones.

The lidar mount designed ended up not being ideal (Fig. 21). It scanned objects just above the curb, and thus, when it saw something come into range, it often came very close, and caused the robot to overcompensate and veer to the right. This could have been solved by putting our LIDAR on a tilt, so we could see the curb at all time.

Fig. 21. LIDAR is horizontal, and not on a tilt system.

We didn't end up using ultrasonic sensor (Fig. 22), which could have helped a great deal with wall following. We relied on the LIDAR for much of our sensing, and when the LIDAR data was incomplete, we were stuck. If we would have implemented wall following using a simple ultrasonic, then the problem of veering right could have been mitigated greatly.

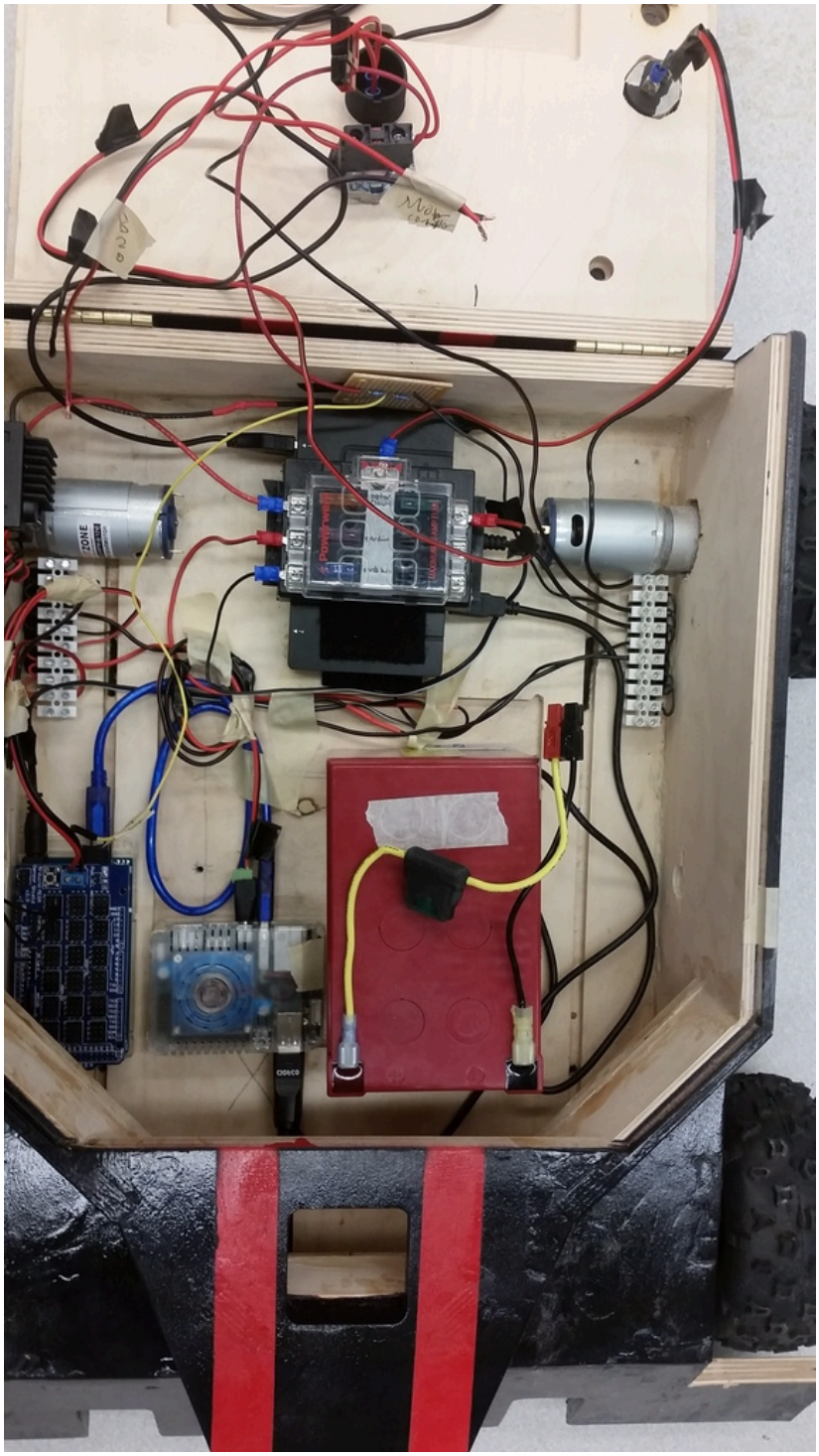


Fig 22. Ultrasonic sensors are not used.

In conclusion, the robot performed decently, although it could have performed significantly better given a little more time to write software and test. I believe a large reason we had such difficulty navigating was that we weren't able to create robust software that could properly handle edge cases.