Florida Solar Beach Buggy Challenge

Group 22:

A

D

V

College of Engineering & Computer Science

University of Central Florida

September 14, 2020

Contents

1 Executive Summary **1**

2 Interests and Motivations **3**

2.1 D 3

2.2 V 4

2.3 A 5

3 Goals and Objectives **6**

4 Function of the Project **8**

5 Broader Impact **9**

6 Legal, Ethical, and Privacy Issues **11**

7 Specifications and Requirements **12**

8 Ideas and Research **13**

8.1 Seeing With Cameras 13

8.2 Solutions to Object Recognition 13

8.3 Human Training Data 14

8.4 Powerful Embedded Boards 14

8.5 Heat Vision for Better Identification 14

8.6 LiDAR 15

8.7 Camera Vision 16

8.8 Ultrasonic as a Last Line of Defense 17

8.9 Mapping Objects using Point Clouds 17

8.10 GPS 18

8.11 Using Depth to Plan Paths 19

8.12 Traversing an Unknown Environment 19

8.13 Approaching the Multi-Team Problem 19

8.13.1 Creating Multiple Different Modules 20

8.13.2 Creating Multiple Identical Modules 20
8.13.3 Creating a Single Module 20
8.14 End-to-End Deep Learning Approach 21
8.15 Multitask Learning Approach 21
8.16 Convolutional Neural Network 22
8.16.1 Unfreeze Convolutional Layers for Fine-Tuning 23
8.17 Image Classification 23
8.18 Cloud Computing in Place of Board 27
8.19 Waypoint Path Planning 27
8.20 ROS 30
8.20.1 rviz 31
8.20.2 rosbag 32
8.20.3 rqt 33
8.20.4 rosserial arduino 34
8.21 Perception Subsystem 34
8.21.1 Semantic Segmentation 35
8.21.2 Simultaneous Localization and Mapping 36
8.22 Planning Subsystem 37
8.22.1 Global Positioning System 37
8.22.2 Hybrid A* 40
8.23 Control Subsystem 42
8.23.1 Control Neural Network 42

9 Budget and Financing **44**

10 Project Milestones **46**

11 Design Summary **47**
11.1 RC Test Vehicle 47
11.2 Simulation 48
11.3 Autonomy System 48

12 Interface Control Document **49**

12.1 The Box 49

12.2 Ultrasonic Sensors 52

12.3 Power 53

12.4 Control Instructions 53

13 Simulations **55**

13.1 Simulation Software 55

13.1.1 Gazebo 55

13.2 Simulated Ultrasonic Sensor 56

13.3 Simulated Camera Vision 57

13.4 Simulated Environment 57

13.5 Beach Buggy Simulator 58

14 Sensor Suite **60**

14.1 Ultrasonic 60

14.1.1 Ultrasonic Sensor Placement 61

14.1.2 Depth Camera 65

15 Autonomy System **67**

15.1 Joystick Node 67

15.2 Autonomous Driver Node 67

15.2.1 User Drive Mode 67

15.2.2 Autonomous Drive Mode 68

15.3 Filter Node 69

15.4 Arduino Node 69

16 Training **70**

16.1 Real World Data Collection 70

16.2 Data Pre-processing 72

16.3 Blocked and Direction Models 73

16.3.1 Blocked Model 73

16.3.2 Direction Model 73
16.4 Data Augmentation 74
16.4.1 Reflection 74
16.4.2 Rotation 75
16.4.3 Pepper Noise 75
16.4.4 Gaussion Noise 76
16.4.5 Augmentation Process 76
16.5 Neural Network Fine-Tuning 76
16.5.1 Learning Rate Schedule 76
16.6 Checkpoints 77

17 RC Test Vehicle **78**

18 Hardware Integration **82**

19 Design Execution **83**
19.1 Build 83
19.1.1 Simulation 83
19.1.2 RC Test Vehicle 84
19.1.3 Autonomy System 84
19.2 Testing 85
19.2.1 Integration with Real Buggies 85
19.3 Evaluation 86

20 Changes **87**
20.1 Hardware 87
20.1.1 Switch from TX2 to Nano 87
20.1.2 Dropping Lidar 87
20.2 Test Vehicle 88

21 Problems **89**
21.1 Simulation 89

21.2 Budget Constraints 89
21.3 Coordination with Other Teams 90
21.4 Purchasing Delays 90
21.5 Ambiguous instructions 91
21.6 Implementation Conflict 91

22 Project Conclusion **93**

1 Executive Summary

All across the world, millions of cars crowd the roads and spew tons of greenhouse gases into the air. This can be deadly for drivers and their passengers; In the United States, 37,133 deaths occurred as a result of car accidents in 2017 alone. Even safe driving is harmful to the environment, so how can these problems be answered? Duke Energy and UCF are seeking this answer via our project, the interdisciplinary Solar Powered Beach Buggy challenge. We have worked with three teams of mechanical engineers who have designed and built buggies that will be driven by our module, a system of ultrasonic sensors, a depth camera, and a processor. Our project explores two possible solutions to reduce fuel consumption and injury: the use of solar power to reduce emissions and the use of autonomous navigation to prevent the need for a human driver. We would consider it a success if it can in any way encourage or further the development of sustainably powered autonomous vehicles.

The current designs for autonomous vehicle systems vary widely, though the kinds of systems that can be realized on our budget are less numerous. We used these models and sensor designs as a basis for our research into how we wanted to build our system, and improved the system from that point. Our system was required to drive off road and in unmarked streets and parking lots, and so does not have the luxury of lines to follow. For this reason, a depth camera was more effective in determining the locations of obstacles, and ultrasonic sensors provided a last line of defense for detecting obstacles while the buggy navigates.

Our approach incorporated a Jetson Nano to process all the sensor data into instructions. Our sensor system will feed information to the Jetson which it will in turn feed into algorithms that recognize obstacles and determine the buggy's surroundings. The Jetson will also help in determining instructions, as the location and orientation of the system will directly affect how the buggy needs to steer. Once the necessary instructions are determined, these will be outputted to the mechanical

engineers' motor controller which will turn them into pulses that will instruct the speed at which the motors will turn and the angle that the buggy will steer.

More than a few design constraints are associated with our project, particularly regarding its scope, our budget, and our capabilities. Being limited to 15 percent of the 2000 dollar budget, there are not many sensors we can afford to implement in our system. We considered using Lidar initially, though it proved to be too expensive. The bulk of that money would be better spent on a Mynt depth camera and a Jetson Nano, leaving just enough for ultrasonic sensors and other necessary parts to put it all together. Another constraint we face is the need to interface with the mechanical engineers' designs, as our module is going to have to be able to drive all three of the buggies.

2 Interests and Motivations

2.1 D

My name is D, and I am interested in this project. I have been interested in conservation and the environment since I was young, and participating in a project that can further the progress of solar-powered vehicles seems like an opportunity for me to make a real impact on the prevalence of renewable energy. Renewable energy is an incredibly important resource for us to investigate, as environmental issues originating from the use of fossil fuels get worse and more numerous every year. This project has the potential to encourage the use of renewable energy when developing the vehicles of the future. This means it could have a ripple effect, reducing carbon emissions that would otherwise come from a large number of vehicles. In addition to the possible environmental impacts, I'm also curious about machine learning and how it might apply to designing an autonomous vehicle like the solar buggy. Gaining knowledge in these fields while working on this project has been immensely rewarding.

2.2 V

When I first came to the University of Central Florida, I knew I wanted to do Computer Science because I wanted to program software. I didn't know what I wanted to specialize in or really even what type of software I wanted to program. I had a thought that it was something I would be good at and make me lots of money. As I went through my classes, I started to realize it was not as easy as it seems and there was more to software development than just writing blocks of code. So many professors and textbooks stress the fact that we are not just attempting to answer the question but also do so in a way that is efficient, concise, and intuitive. I've found that my interests lie in Computer Vision and Artificial Intelligence and these topics require an in-depth analysis of the problem using multidisciplinary knowledge to solve real-world topics. When I saw that this project was on the list of ideas I knew it was the one for me. I just love the idea of a future where everything is automated, and all the tedious work is done by robots. I hate the drive to UCF and the time it takes to find parking but if I had a car that could take me there and park as it drops me off at the building I need to go to I would have so much extra time to cram for that test. I hope that with the knowledge I gain from this project that I could one day be a part of building that future where we could leave less damage on the environment while improving everyone's productivity.

2.3 A

It seems as though not only I wanted myself to work on this project, but also the universe that wanted me to invest my time in it as well. Personally, artificial intelligence has always been something that has fascinated me as it has always seemed like some sort of magic. I spend about ten hours each week just traveling by car. Autonomous vehicles are of immense interest to me as they could allow me to take my focus off of operating a vehicle while traveling and let me put focus on things that I feel are more important and productive. With experience creating a self-driving buggy from the ground up I feel it will make me a better candidate in the eyes of employers as artificial intelligence is in high demand and is definitely a space I feel myself gravitating towards after graduation. The effects that self-driving buggies have on safety are what really shine the brightest to me, since more than 90% of motor vehicle accidents are caused due to human error. Building a robust system that could generalize to many different types of environments and situations and save lives is a dream of an opportunity. I really look forward to learning how to wave the magic wand when it comes to autonomous vehicles.

3 Goals and Objectives

The goal of this challenge was to design and create a system to be used on three beach buggies built by the Black, Gold, and Blue mechanical engineering teams. This buggy had to be capable of driving itself in different environments while avoiding obstacles. Originally the buggy was going to be tested on a beach, but early on in our meetings with the MAE advisors it was determined that we would be testing on paved and unpaved locations near UCF instead.

The first large objective we had was to decide upon and plan a system to both gather information from the buggy's surrounding environment as well as process that data. There are many different options for every component of this system, and once we had decided upon them we began producing it. We planned on doing this by coming up with a large number of ideas (as we were instructed by Bob Hoekstra at the senior design boot camp) and then choosing the best ideas based on function/cost. After choosing, we began planning the steps needed to create these components and assigning them to the team. This also involved drafting and finalizing the final Design Document, which set our plans in stone for how we were to create the autonomous driving system as a module to be used by each of the mechanical engineering teams' buggies.

Our next large objective was to create the module. This was broken down into many smaller objectives based upon how we decided to build the module, and these were fleshed out after deciding on a design. They all would have the same outcome, however; once we completed the objectives we had a module that can attach to all three of the mechanical engineering teams' buggies and operate them with no input from a human. This involved purchasing parts and developing programs to run on those parts. One constraint of building this module was that it must fit onto the buggy, and it had to function properly even if all three buggies are slightly different in the ways they were designed. This meant that after the module was completed,

we needed to test it and make sure it can function properly when affixed to any of the three different buggies produced by the mechanical engineering teams.

As we understood, these two objectives comprised most of what was required of us for Senior Design 1 and 2. To complete both we spent two semesters and one summer working on a series of small objectives that led to the completion of our ultimate objective, having a functional buggy driving module.

4 Function of the Project

The function of our final finished project This buggy is able to drive at a brisk pace of three miles per hour. The mechanical engineering students also needed to ensure that their buggy could keep this pace while it drove for at least twenty miles and be powered completely by solar energy. While it drives along, it is able to avoid obstacles and create a new path to move around these obstacles. This autonomy is a result of our module, which is able to attach to each buggy separately. To facilitate the process of integration, we created an Interface Control Document early on, per Dr. Kurt Stresau's advice. The mechanical engineering team buggies had varying degrees of success communicating with our module, but our simulation and test vehicle discussed later in the paper are able to communicate and take controls consistently.

5 Broader Impact

The autonomous Beach Buggy we helped to build is not just our senior design project; it represents all that we have learned so far and what we will continue to learn as we progress in our field. It will be the last project we create as UCF students and will certainly be the most impactful. We hope that the autonomous Beach Buggy accomplishes a myriad of objectives and creates lasting impacts on societies throughout the world.

If it is successful, the technology and knowledge created could reduce carbon footprints by making solar-powered vehicles more desirable. In 2016, over two billion tons of carbon dioxide was released by transportation in the United States, far surpassing the amount released from the burning of fossil fuels for power generation. If even a small amount of vehicles started using renewable energy such as solar we could make a difference in saving our environment.

With self-driving cars, we would improve the productivity of almost every working adult in America. According to the U.S. Census Bureau, the average, one-way commute time is twenty-six minutes. Over the course of a year, a full time five-days-a-week job, the round trip would add up to over 200 hours. With 128 million full time employed people in the US that could be over twenty-five billion hours of extra time used for self-improvement, working on tasks, even just sleeping and getting more rest.

An even more important aspect of an intelligent automobile is the number of accidents it could reduce. Computers are faster, calmer, and more focused when in stressful situations than humans. A car will not panic if it is about to hit another car and will hit the brakes quicker, angle the car to reduce impact, and in general, make the best decision without fear. A smart car always will obey the law and drive defensively in the process, minimizing the number of situations that will even lead to an accident. Removing people from the equation will make driving, which is an

unavoidable consequence of civilization, a safer activity that is no longer clouded by human error. With the creation of our solar-powered autonomous beach buggy, we hope to pave the future for these ideals. We also hope that the success or even the failure of our project will contribute to the overall outcome of a society with vehicles that do not damage the environment or living things.

6 Legal, Ethical, and Privacy Issues

For our project, we did not run into any legal, ethical, or privacy issues. We tested the vehicles on willing participants using original code supported by publicly available libraries. If Duke Energy were to scale up our solution and made these beach buggies open to the public, there could be a slew of problems. An autonomous vehicle creates a question of responsibility. There will be some situations where the car gets into an accident, and we must ask, is it the driver's, the manufacturer, or the programmer's fault? When it gets into the situation where it must run over a group of elderly on the right, a single child on the left, or harm its driver, how must it make that decision? The most natural solution would be to make the driver the primary control and override for each autonomous vehicle. Even if the buggy could be genuinely autonomous a human must be aware and in control of the vehicle ready to make a hard decision just like in a real car. Until there are laws that clearly define the legal and ethical responsibility of autonomous vehicles, we can not have vehicles make their own decisions truly. They will always have to be backed up by a human reference. A privacy issue we foresee is if we were to collect location data from each vehicle to give other autonomous vehicles greater awareness for their decision making. Another problem is if we were to gather all data related to the movement and control of the vehicle in the case of accidents to further prevent situations leading to them. In either case, we can easily allow users to opt-out of the program, just like how software is structured in current times.

7 Specifications and Requirements

Through communication with our mechanical engineering teammates, sponsors and advisors, we have compiled a set of requirements that our solution must meet. Our Solar-Powered Beach Buggy must:

- Be completely safe, in that it does not harm any person in or outside of the Buggy
- Be fully autonomous and avoid obstacles
- Be designed in such a way that a module of predetermined size can be mounted on the buggy to process sensor data and decide how the buggy should drive
- Avoid driving off the beach or into the water
- Be able to drive itself on a beach for more than 20 miles and drive for periods of time longer than 10 hours
- Drive at a speed of 3 miles per hour or less
- Be fully solar-powered
- Be able to carry a passenger weighing up to 120 pounds
- Cost $2000 or less to build
- Be comfortable enough for an adult to ride in for an extended amount of time

8 Ideas and Research

To tackle this project, we went through the process of brainstorming to create a number of different options that we could choose from to use and expand upon for our final buggy design. Detailed here are a few holistic ideas for how the buggy can operate, as well as more specific ideas for components that will make up these ideas. We did not end up using all of these future-tense ideas in our final design, but they are kept here to show the process we went through when constructing our project.

8.1 Seeing With Cameras

There are many possibilities for how we might design a system that controls the buggy computationally. One possible solution to making the buggy autonomous is to use a camera to capture images of what is in front of the buggy and use some sort of processor to run object recognition machine learning algorithms that would determine how the buggy controls itself. This would allow the buggy to avoid obstacles and stop when something is in the way. We could also incorporate sensors on the side of the buggy, which could detect when something was in the buggy's path and trigger it to stop. This concept can be broken down into even smaller ideas, as each individual component will have to be chosen or possibly made in a certain way.

8.2 Solutions to Object Recognition

One part of the buggy that will need to be chosen is how it will handle the object recognition computationally, especially with the restrictions on how much power this computation can use and how much it can cost. One option is to use a combination of a Raspberry Pi and the Movidius Neural Compute stick to recognize objects in the footage captured by a camera facing towards the buggy. This can be accomplished by creating an object recognition model in the Caffe or Tensorflow frameworks and running it through the Movidius on the Raspberry Pi. When the module recognizes on object in front of the Buggy, it will send a signal to the buggy to make it stop and redirect itself. This is one way to prevent the buggy from running into objects, and

it also opens up options for us to design a way for the buggy to programmatically decide which path to take next.

8.3 Human Training Data

A different approach to controlling the buggy's steering and acceleration autonomously would be to take a different supervised learning approach and actually drive the buggy using an expert human driver and record the driver's decisions (steering angle and acceleration). These decisions will then be used as the labels to the data from the sensors.

8.4 Powerful Embedded Boards

Another technology we were thinking of using was the NVIDIA Jetson Nano Development Kit. It is expensive, but it would definitely be able to perform big computations on high resolution and high frame rate images coming in which means faster and more accurate decisions for the buggy to take in avoiding and calculating paths for itself. Currently we do not know the amount of power we will be working with which might make a Jetson too energy-costly when it comes to real-world applications. Over the course of 8 hours it will be using about twelve times more power than a Raspberry Pi.

8.5 Heat Vision for Better Identification

The camera could even use infrared modules to create a thermal image of what it is seeing which might give us more insight on the nature of the obstacle. This may be an easier way of detecting the ocean since the gradient of temperature of the water is different from the sand. It might create clearer borders which in turn make it clearer for models to determine different objects which might make our program more efficient.

8.6 LiDAR

LiDAR is an acronym for light detection and ranging. As shown in the name these sensors shoot off a wave of light. Light c is the maximum speed we can observe in our known universe. While it uses the concept of emission and reception like ultrasound the implementation is vastly different. Generally LiDAR uses two sensors: one laser transmitter and one laser receiver on the near infrared wavelength. Infrared waves have different strengths that make it a more reliable source of vision than ultrasonic. Being part of the electromagnetic spectrum, its non-physical properties allow it to go farther and return much quicker than mechanical waves. The distance of objects using LiDAR is given by this equation:

$$L = \frac{Tc}{2}$$

where L is the Distance, T is the time between emission and receptions(time is halved because it accounts for going there and back), and c is the speed of light.

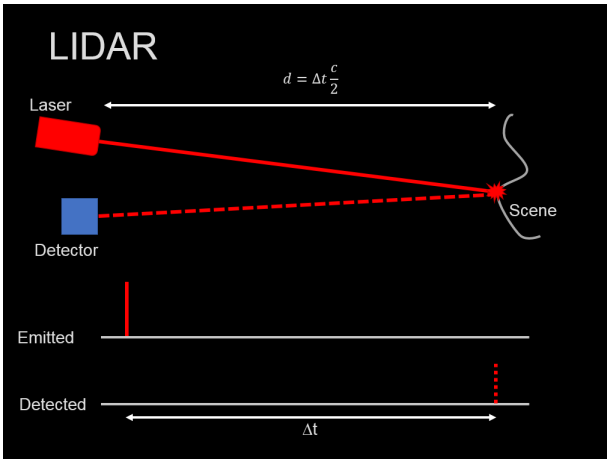


Figure 1: Sketch of the functionality of an LiDAR sensor, from [1]

Pros

- Data Update Speed (light moves at the fastest speed known currently which allows data points to be generated quickly)
- High Sample Density (It creates many data points creating samples that have a much higher surface density which is useful for doing calculations on)

- Light-Independent (can be used at night or in low brightness situations)
- Penetrative (Can reach through thin materials)
- Accuracy (Can be focused to pinpoint more specifically than mechanical waves)
- No Geometrical Distortion (Not affected by any geometrical distortions such as angular landscapes unlike other forms of data collection.)

Cons

- Expensive (one of the more expensive sensors especially when getting into more than one dimension of data collection)
- Affected by weather (Heavy rain or whiteout conditions will create inaccurate or wrong data points)
- Degraded at high sun angles and reflections (The principle of reflection makes is inaccurate when there are large reflections or high sun angles)
- Water Unreliability (Water or surfaces that are non uniform will affect the reflection)
- Large Data set (A large amount of point cloud data requires more time to compute and analyze)

8.7 Camera Vision

Camera vision mimics the human method of detecting obstacles. Light from the object you are observing goes into the camera lens. This incoming "picture" hits the image sensor chip, which converts it into millions of pixels. The sensor measures the color and brightness of each pixel and stores it as a number. The data we receive is long string of numbers describing the exact details of each pixel it contains. Once it is turned into numbers we can work with the data to classify the pixels into objects which will allow our computer to perceive them in a manner that is conducive to our goals.

8.8 Ultrasonic as a Last Line of Defense

Another option for detecting objects in the vicinity of the buggy is to use ultrasonic sensors. These sensors are inexpensive, and they can detect when an object is up to 400 centimeters (about 13 feet) in front of them. When an object crosses the path of one of the sensors, it can send a signal through the trigger pin that can be processed by the module, which can then send a signal to stop the buggy. The buggy will need some method of finding a way around the object, but these sensors should be enough to prevent the buggy from driving into an object.

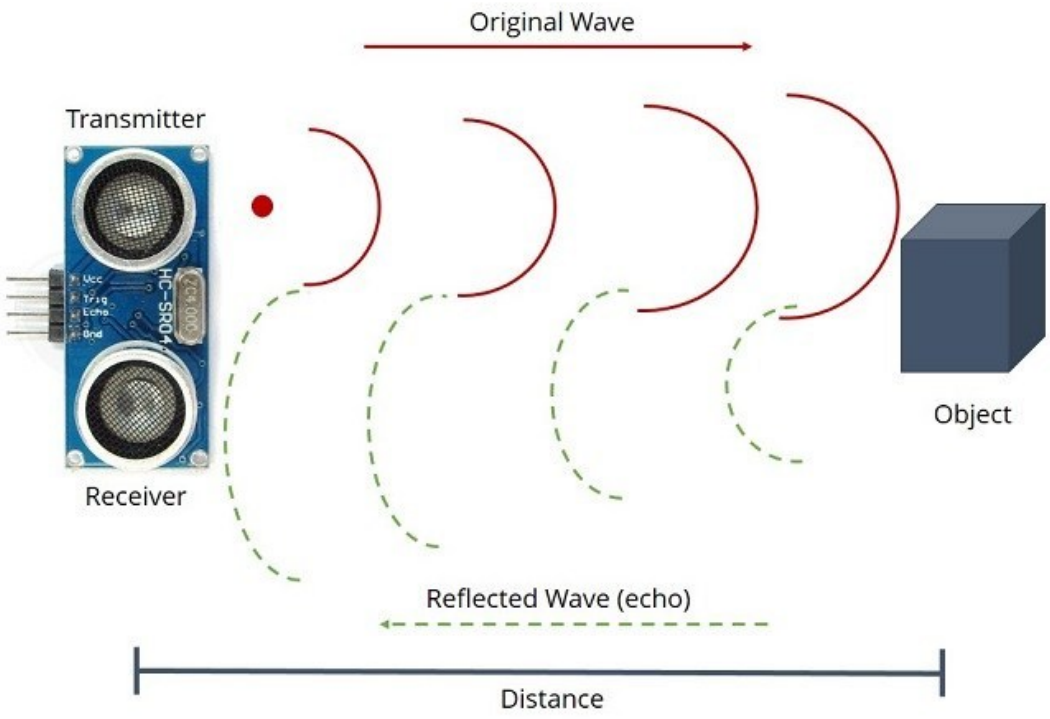


Figure 2: How an ultrasonic sensor functions.

8.9 Mapping Objects using Point Clouds

A more expensive option for detecting objects is to use Lidar. This is similar to an ultrasonic sensor, but instead of measuring the rates at which sound waves reach the sensor it measures the rates at which pulses of light bounce off objects and reach the

sensor. These sensors are what many self-driving car manufacturers choose to use as their vehicles' primary source of vision, and have a much higher range than ultrasonic sensors. This is because Lidar is a fast and efficient way of mapping an entire space, and a properly trained machine learning model can use that mapping to decide which way the car should turn and how fast it should accelerate to get to its destination. Many advanced models can use Lidar data to determine what an object is (such as a person or a sandcastle) and how fast it is going in a certain direction. This can be useful information when trying to navigate around the object, as the size, shape, distance, speed, and direction all factor into what direction the buggy will need to go to avoid the obstacle.

8.10 GPS

Earth is a pretty big place. If one were to be given the task of getting to a particular destination far away without knowledge of what direction to go in, they might not ever get there or know they truly reached their destination. For this reason we plan on using GPS. GPS stands for Global Positioning System.

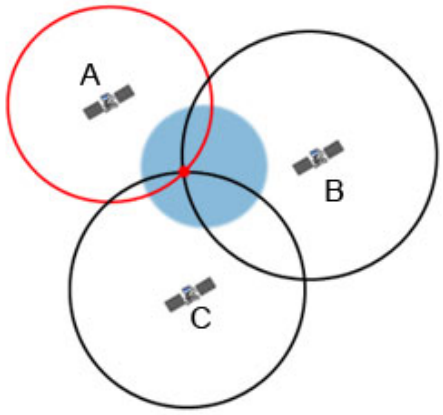


Figure 3: Two dimensional diagram of satellites A, B, and C being used for trilateration to find the position of the receiver (the red point).

8.11 Using Depth to Plan Paths

During our research, we came across cameras with two lenses that can recognize how close or far away any surface in the image is from the camera, or the depth of that surface. This can be invaluable in building autonomous robots, as this depth information can immediately be used to determine if an object is too close and needs to be avoided. Depending on the distance of an object, the path that our buggy chooses to follow may be completely different. For example, our buggy may choose to not move around a near obstacle on a certain side of that obstacle if it can determine that there are more obstacles a certain distance along that side. The depth camera would give us the ability to determine when these sorts of objects are in the way.

8.12 Traversing an Unknown Environment

It is possible that for this project, the buggies may need to drive in an environment where they have not been and for which training data has not been collected. This might be the case if the beach where we are going to be driving is not determined until later, and it has even been mentioned that we may not end up driving on a beach at all. For this reason, the location where we end up testing needs to be determined as quickly as possible, and we need to be able to collect training data from an environment that is as similar to the testing area as possible. This means we would like to have an understanding of what objects we are trying to avoid, as well as the type of weather and lighting we might have while testing.

8.13 Approaching the Multi-Team Problem

For this Solar-Powered Beach Buggy Challenge, we have been assigned to be the single Computer Science team to work with all three Mechanical Engineering teams (Gold, Blue, and Black teams). Each mechanical engineering team is tasked with the same end goal of creating an autonomously driving solar-powered beach buggy, but our best guess is that each of those teams will be working on designing and building the body of their buggies while we work to give them a mind of their own. This

means that our team would have to work to serve all buggies with the ability to drive autonomously. This involves lots of communication and collaboration on everyone's part, and we propose a few different approaches.

8.13.1 Creating Multiple Different Modules

This approach has been suggested to us as a means of accommodating the engineering teams in the event they want to create different buggies. If each individual buggy is designed to have a different shape, size, drivetrain, etc. then our module that controls the buggy will have to be different for each one. In this case, it may be easiest to design three different modules that are shaped and function differently to fit each of the buggies. This is not likely to be the route we decide to go, as it could end up requiring three times as much effort as designing one module, and can get expensive quickly.

8.13.2 Creating Multiple Identical Modules

This approach employs the creation of three separate, identical modules. This plan would also allow the mechanical engineering teams to test their buggies without having to worry about the other teams occupying a single module. Though this approach may prove to be more difficult in the long run, it has been suggested that we actually create three individual modules to allow each of the mechanical engineering teams to demonstrate their buggy driving during the showcase simultaneously. This would not be terribly difficult as the same model we use to perform object recognition as well as determining a path forward could be loaded into each module. One potential issue with this approach is cost, as purchasing the parts needed to construct three different modules may be outside our allotted budget.

8.13.3 Creating a Single Module

Our final proposed approach involves just creating a single module to be shared by all three mechanical engineering teams. This would have us create a single specification for all mechanical engineering teams to follow. We would much rather agree with

the engineers on a set of measurements and design requirements so that our module might plug into and work with all three buggies. These specifications can be laid out on an ICD, or Interface Control Document. This document will spell out the measurements of the buggy as well as our module, and it will describe exactly how those two components will communicate with each other so that all teams are on the same page and know what to expect. In this way, all teams can continue their development without much input from the other teams, which can make it much easier for us to make progress. When it comes time to test the module on the buggies, it could be a bit trickier as we would have to come up with some sort of scheduling system so that we can could rotate the module to each buggy.

8.14 End-to-End Deep Learning Approach

End-to-end is a method involves using a single neural network to cut out the need of traditional intermediate tasks to get the same end result. In the world of speech recognition, a more classic approach would involve taking the audio input, hand-crafting features from that, using features to detect the phonemes, using the detected phonemes to predicted the words, and using that to finally construct a predicted transcript. Using end-to-end learning would just involve taking the raw input audio data and passing it through the neural network to get the transcript. Companies like NVIDIA have applied this approach to autonomous vehicles with great success when it comes to collision avoidance. [2] The issue with this approach would be that we would need a sufficiently large neural network to achieve this. Because this problem is so complex with so many edge cases to be considered, we imagine the neural network would have to be infeasibly large.

8.15 Multitask Learning Approach

Expanding upon our end-to-end learning approach for collision avoidance, we could use a neural network that uses multitask learning to control both the steering and acceleration of the buggy. Multitask learning basically, in this case, means that our

model is tasked with performing regression for two values: one for the steering angle θ and one for acceleration α. We believe this is would be a better approach than coming up with separate models for steering and acceleration. When using a single end-to-end model that uses multitask learning, the model can also learn the relationships between steering angle and acceleration. For example, with this approach the model is more likely to learn that its better to lay off of the acceleration while taking sharper turns.

8.16 Convolutional Neural Network

Convolutional neural networks are networks that are used frequently when trying to do image analysis and object recognition. They work by analyzing images as a matrix of pixels, with each image being spread into three layers, or channels, that each represent the RGB values of that pixel. For example, an image that is 50 pixels by 50 pixels will be broken down into three 50 by 50 pixel channels. Each one of these channels will be populated with numbers representing the levels of red, green, or blue in that pixel. Using these numbers as the basic input data, the network can try and find patterns in the pixels that help it to recognize objects.

The processing of the image into useful data will begin by passing a filter over over the image. This filter is another type of matrix or channel that is much smaller than the original, and the dot product of this matrix and a section of the original channel is saved into another layer called the activation layer. Because the dot product returns one number, the layer built from these dot products is smaller and easier to store than the previous layer. To save even more space these maps can then be subsampled, which involves shrinking the data by dividing it into subsections, and then taking the largest value within the subsection and discarding the rest. This is effective, because each filter is designed to return high values when it recognizes certain patterns. It is possible for potentially valuable information to be lost in this step, but it is worth doing for the sake of saving memory and processing time. Soon each layer will have gone through subsampling (and there will be many layers, as each

of the original three channels creates a new layer for each filter passed over it) and then filters can again be passed over the subsampled layers. This shrinks the data again and again, and allows numerical patterns that represent recognizable objects to become more apparent to the network.

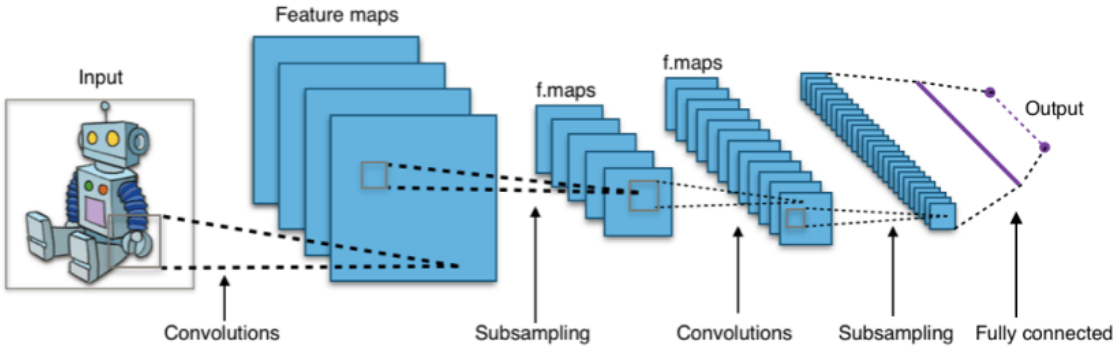


Figure 4: Convolutional process of subsampling and creating maps.

8.16.1 Unfreeze Convolutional Layers for Fine-Tuning

If using a ResNet convolutional base pretrained for extracting features from raw images, depth images and segmented images, after training a fully connected neural network, we could unfreeze the last convolutional layer of the ResNet convolutional base. This means that the weights of the last convolutional layer would be trainable. The ResNet convolutional base should be excelent at extrating useful features for the fully connected neural network but the original ResNet model was trained on the ImageNet dataset to perform image classification, not regression for steering angle and throttle intensity. Unfreezing this last layer should tweak the convolutional base slightly through more training of the entire neural network to produce more valuable features for the fully connected neural network, yielding a more accurate model.

8.17 Image Classification

When we see a person we identify them as a person and we understand we cannot walk through that person so we plan to walk around the person in accordance with our

knowledge base. To understand that you must avoid something you must understand that something exists. Our perception system must learn to identify objects in a manner that allows it to think like a human and avoid it. Maybe it does not have to be as accurate as labelling the person as "John" or even as a person but it must know that it is an obstacle that must be planned around. In our case it may be easier because on a beach there may not be too many things to classify.

Imagine the buggy is standing from this pictures orientation viewing upon the beach. In this picture there is mostly sand with people close to our field of view.

Figure 5: Picture of London Beach from @sophieneo, Instagram.

The computer may first try to recognize anything that is an object. It would then create partitions in the pixel values that classify everything in a box as a something.

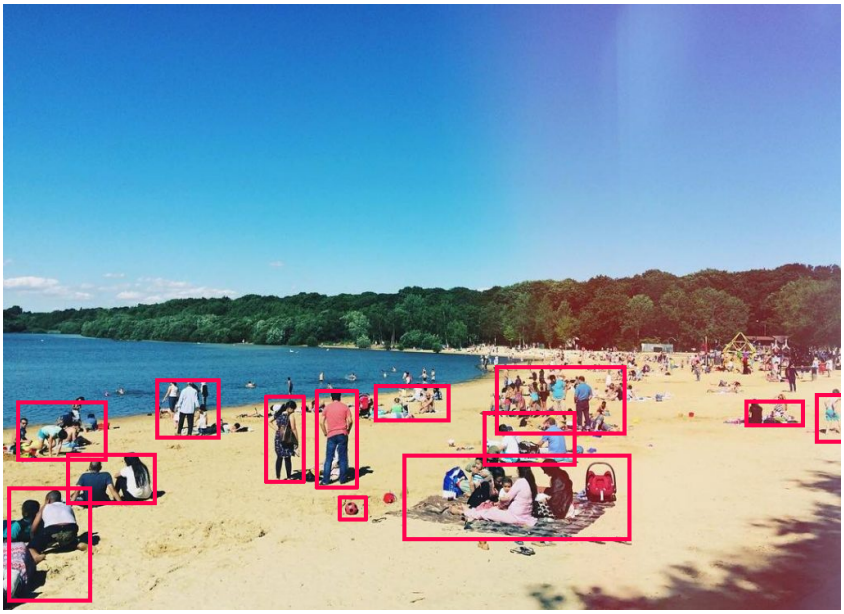


Figure 6: Objects on the beach.

It may also identify specific objects as some type of thing a "ball" or a "person". The most basic of planning it will do is that if there is something in the way then it must be avoided. If there is anything in a red box it must go around it or stop until the object passes.

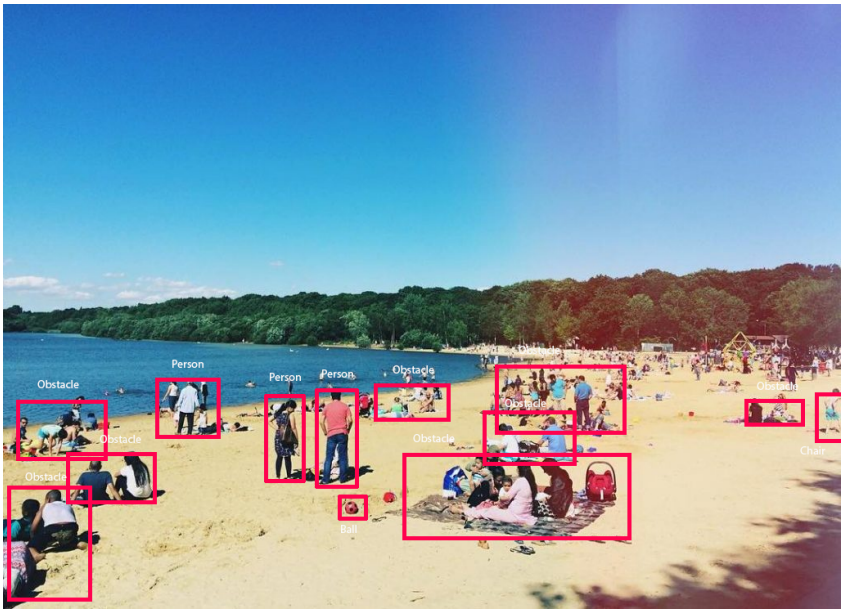


Figure 7: Objects on the beach identified.

It could also even simplify the image further by creating zones that it cannot enter and zones it can enter. It no longer needs to know anything besides the fact that it cannot make a path through red and only make a path through green.

Figure 8: Objects on the beach turned into regions.

8.18 Cloud Computing in Place of Board

Though it is hopefully unlikely, there is always the possibility that our embedded board is somehow damaged to the point of being unable to function during testing and handling. In addition, we may find that it is cheaper or more efficient not to use a board at all. In place of this board, we can use cloud computing to run our model and navigate the buggy through the testing environment. The process will involve two main steps: First, the sensor data will be sent to a cloud server from the buggy where the model can process the data and then develop a path for the buggy to follow. Secondly, the path data will be sent back to the buggy and can be delivered as instructions to the motors and steering components.

8.19 Waypoint Path Planning

A problem we foresee is allowing the Buggy to make decisions about which way it should go. It might be able to detect and avoid obstacles, but if it does so by going in circles, that is not really a success case. We want the Buggy to be able to reach a destination by possibly using GPS to frequently get closer to the point as it avoids

obstacles. The GPS could be divided up into intermediate waypoints that we place ahead of time or in the buggy's maps to keep it going in an efficient line so it does not try to get to the end point by going completely off the beach and trying to find another path.

By intelligently planning out waypoints for the buggy to follow on its way to the destination, we can ensure it does not drive into an area that is determined not safe for it to be. This is considered autonomous navigation, and it is comprised of being able to determine where the buggy is and how the buggy is going to get where it needs to go. Determining where the buggy is can be accomplished relatively accurately using GPS. GPS locations can occasionally be inaccurate, but in an open area like the beach or the sidewalks of campus it should work well enough to get the buggy where it needs to be. The GPS will also be able to help in the formation of waypoints and allowing the buggy to follow these waypoints.

Exactly how we go about forming these waypoints along the route from start to finish can be accomplished in a few ways. Regardless of the method we choose, it will most likely be beneficial for us to be able to estimate the distance between the location of the buggy and the location of the marker that got placed by the user. This can be calculated using the latitudes and longitudes of these points and using them in the Haversine Formula. This formula is designed to find the distance between two points of latitude and longitude. The points are treated like points on a 2D graph, however the formula is used in place of a regular distance formula to accommodate for the curvature of the Earth. It is essentially the equation:

$$d = r\theta \tag{1}$$

It is used by finding a central angle theta with this equation:

$$centralAngle = 2\arcsin\left(\sqrt{\sin^2\left(\delta L/2\right) + \cos L_1 \cdot \cos L_2 \cdot \sin\left(\delta Longitude/2\right)}\right) \tag{2}$$

This can then be plugged in for theta, and with the radius of the Earth as r we can find the distance between the two points along the surface of the Earth. Knowing the distance can be beneficial when deciding if the buggy has been driving for too long to get to a destination, such as when it is turning in one direction and making circles instead of progress towards the destination.

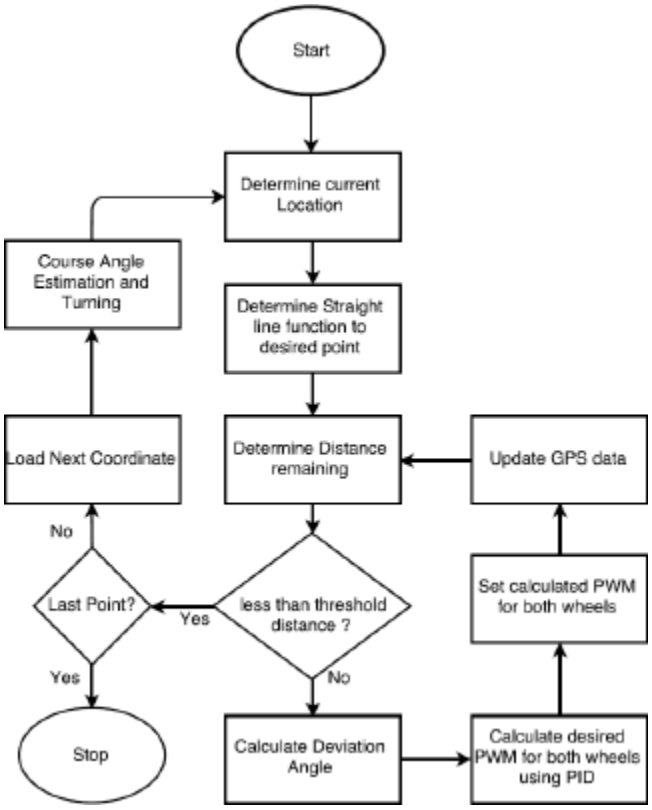


Figure 9: Flowchart that describes using GPS to navigate.

After determining the distance between two points, the buggy will actually need to come up with a method to generate waypoints to move between those two points. This may need to be done by hand ahead of time, or the challenge may dictate that this has to be done automatically. If it needs to be done automatically, there are multiple ways we can achieve this. One possibility is to use GPS data and information from the buggy to determine possible movements that the buggy can make. These

can then be estimated in different directions on the map, creating nodes that can serve as waypoints. This can get difficult to calculate at large distances however, and may be out of our scope. For this reason, it may be beneficial to plant waypoints ahead of time or to calculate waypoints in predetermined distances apart, and then update these waypoints based on information obtained from the sensor.

8.20 ROS

ROS stands for Robot Operating System, and it serves as a framework that helps save time and drive robotics development. It exists to simplify the process of developing and integrating the different parts of a robot, including sensors, decision-making, and motors. Normally these components would exist entirely in isolation until code and drivers were written to handle different hardware inputs and outputs, different protocols, and different means for controlling and reading information from the various parts of a robot. ROS handles this process automatically, so nodes can either request information from another node or publish information continuously to another node. This information could be instructions, images, sensor readings, or anything else relevant to the function of a robot.

ROS divides all of the tasks a robot must perform and its different parts into individual processes, or nodes, and coordinates and keeps track of these nodes and their activity. ROS also makes it much easier to allow these nodes to communicate, which can improve performance of a robot and make its actions more efficient. This is accomplished by storing certain values, such as a temperature, in the ROS master process and allowing all other nodes to access these values. Another way ROS allows nodes to communicate is through direct message passing. This level of cooperation is not necessarily just between sensors and one processor, as ROS permits a fully distributed format whereby multiple computers and sensor systems work to control the robot as a whole. Sharing computation, storing important values and allowing for message passing makes ROS an invaluable tool when designing robots, as orchestrating the cooperation between components would otherwise be a difficult process.

ROS is a particularly effective tool for simulating a self driving car for the reasons above, in addition it offers a variety of packages that help with different aspects of the simulation and processing input data. Many of these aspects of developing with ROS translate well or effortlessly to our actual buggy, meaning that over the course of the simulation development we will be creating our system and not only simulating it. Listed below are some of the packages we intend to use to accomplish this.

8.20.1 rviz

One package that may help us display important data when simulating is Rviz. Rviz is used to visualize what the robot's sensors are telling it, how the robot is processing that data, and what actions the robot undertakes as a result of this processing. We will use Rviz as a means of seeing through our buggy's camera, as trying to understand what the buggy sees based on numerical values of pixels or groups would be difficult. By being able to see what our buggy sees, we can quickly identify issues with the camera or the processing of the camera data. Rviz can also help us visualize depth information from the camera as well as the path the buggy intends to take through the environment.

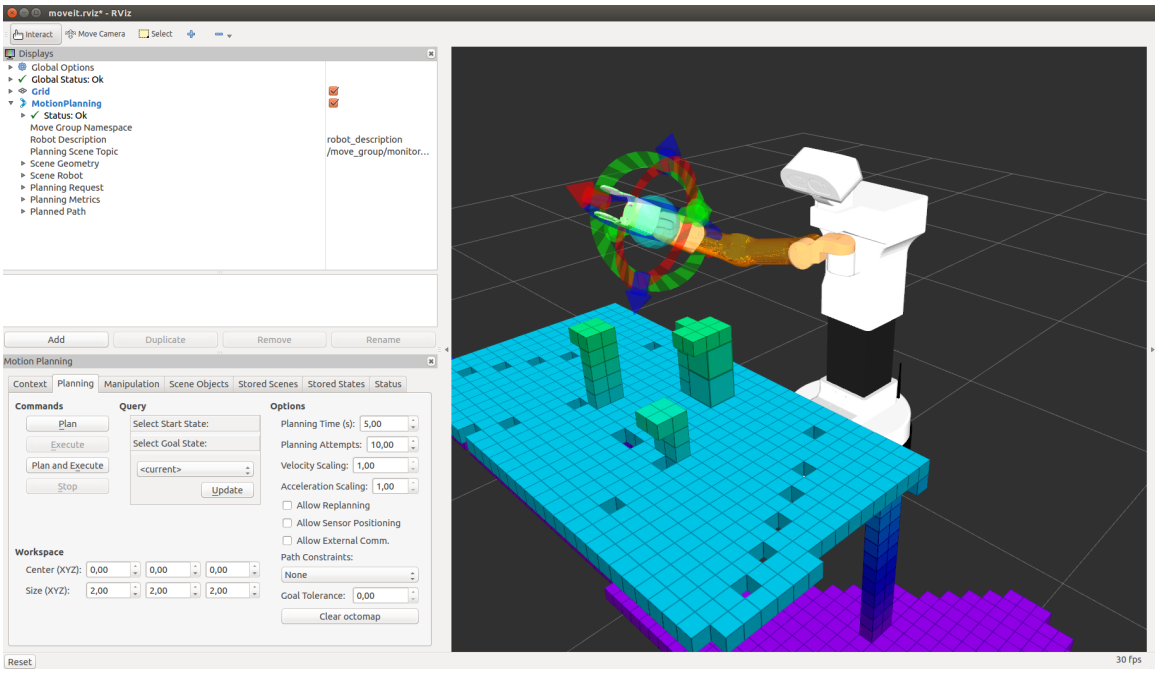


Figure 10: An example UI with rviz. On the left is the list of displays and on the right is the 3D scene.

8.20.2 rosbag

Rosbag is another package that will be important for handling data in our project. It helps collect the input data from sensors and store that information in files, or bags. These files can be accessed and visualized later, and this can be faster to run during the simulation than capturing and visualizing the data in real time. Rosbag can store recordings of all kinds of topics and messages that were sent between systems of the robot. Storing bags can also help if we would like to try visualizing data in different ways, such as creating plots or graphs from images taken from a camera. Trying to track different metrics and generating new kinds of graphs would not be possible during live data collection, but by keeping records of that data we can go back and experiment later. This might also be beneficial in training a model, as information collected during a manual drive could be stored and used as training data. This data can be easily modified as well, allowing for easy data augmentation.

8.20.3 rqt

For creating certain GUIs that will aid in running our simulator, we will use a package called rqt. This package can run any type of gui, and it can be used to visualize important aspects of a robot's function. These are stored in rqt_robot_plugins, and one in particular that we will be using is called rqt_robot_steering. This plugin creates a GUI for steering a robot, and can be used to manually steer the buggy while we are setting up the simulation by connecting to a topic that takes in velocity commands. Rqt has other robot plugins that can be used in conjunction with one another to create a full visualization of all aspects of a simulation. This can include sensor data, message data, steering angles and throttle, and other important information.

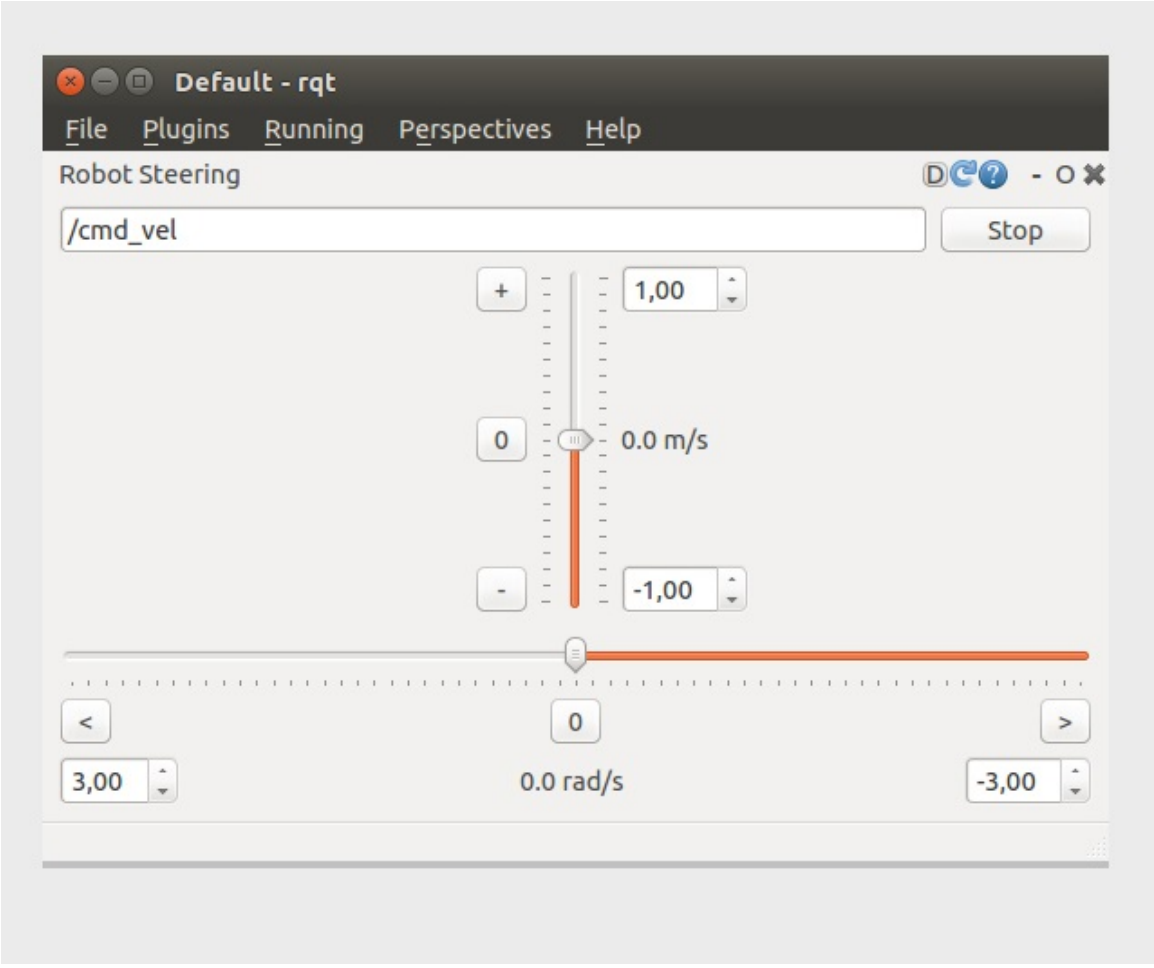


Figure 11: A screenshot of rqt_robot_steering.

8.20.4 rosserial arduino

One important function our robot has is its ability to communicate with a motor controller that is handled by the mechanical engineering teams. The teams all used an Arduino, and ROS has packages that allow our Nano to speak with an Arduino over its Universal Asynchronous Receiver/Transmitter, or UART. It treats the Arduino as if it were a ROS node, and this means it can publish/subscribe to messages from other systems in the buggy. The Arduino can then use those messages to send certain PWM pulses to control the steering and throttle intensity.

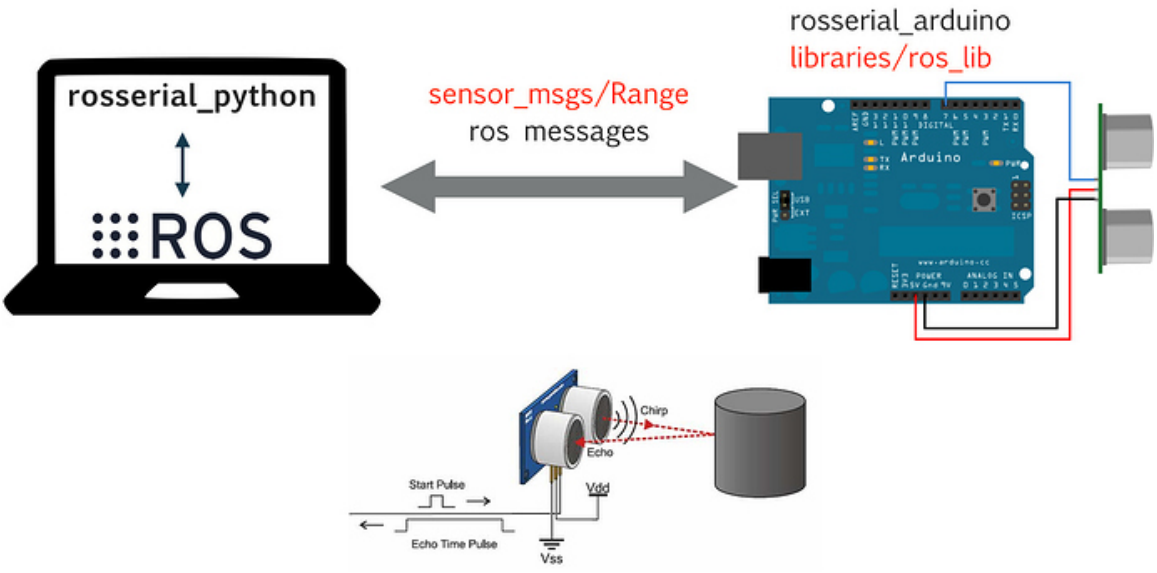


Figure 12: Diagram of ultrasonic sensor connected to ROS through Arduino.

8.21 Perception Subsystem

Eyes alone cannot make decisions! We now need something to make sense of all this data being collected by the sensors from the Vision Subsystem. The Perception Subsystem will be in charge of taking in data from the Vision Subsystem and using this data to classify objects, create a map of the environment and localize the buggy.

8.21.1 Semantic Segmentation

We plan on using semantic segmentation to classify the regions in our raw images from the MYNT EYE. Semantic segmentation involves classifying different aspects of an image into categories, and it is something that our brains do almost instantly. For example, consider a picture of a street. The brain can discern immediately between which parts of the picture include a car, which include the street, and which include the sky. This is an easy task for humans, but not so for a computer. Semantic segmentation is a key factor in a computer system understanding an image, but it is difficult to accomplish cleanly when an image is merely a set of values to a computer. Every pixel will need to be labeled as part of a group: car, street, sky, etc. For this to be done, we will need a powerful model.

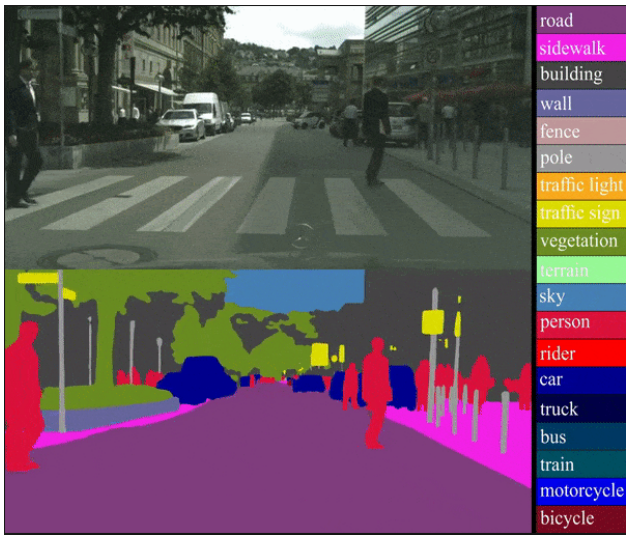


Figure 13: Example of a segmented image made using an autoencoder.

To perform semantic segmentation on our raw images, we plan on using a fully convolutional autoencoder. Convolutional autoencoders work by simplifying input, and having its output match the corresponding segmented image. They can recognize patterns in image data and then simplify them into signals that are less heavy than a group of pixels. They do so in a hidden layer or multiple hidden layers between the input and output, and this helps reduce the size of the data to boil the input down to its essence and increase computation time. This is helpful for us, as it will allow

the Jetson Nano to more quickly figure out what it is looking at and determine a way to move forward. After running through the neural network, the output will be an image that has every pixel classified into a certain category. We think it would be appropriate to have the following categories: derivable space, building, pole, person, vegetation, fence, sky, car, bike, bus and water.

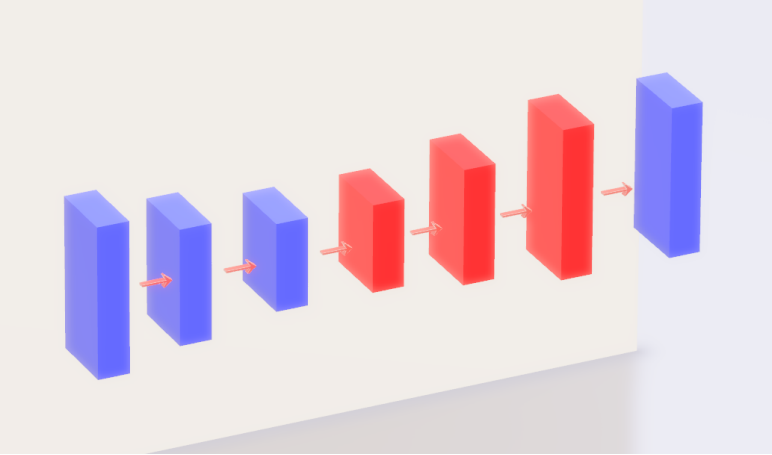


Figure 14: General overview of a fully convolutional autoencoder architecture.

8.21.2 Simultaneous Localization and Mapping

Since we are using a depth camera, the depth encoded images from the depth camera can be represented as a 3-dimensional point cloud. The point cloud can be used to create a map of our environment and localize the buggy within that map. We can use that map to then create a 2-dimensional occupancy grid, detailing the area where the buggy can and cannot go. With a map and occupancy grid of our environment we are able to search for paths that we plan for our buggy to take, but that is a job for the Planning Subsystem.

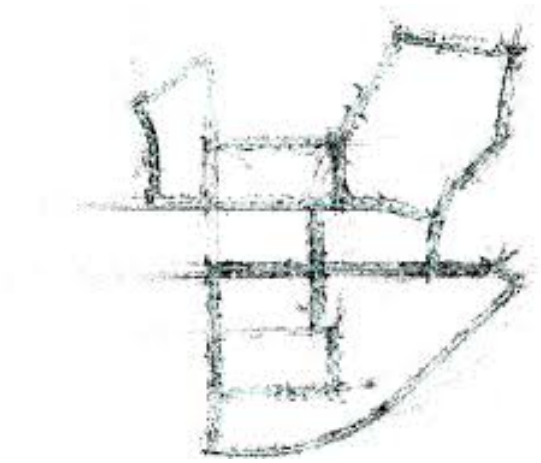


Figure 15: Image of a point cloud created using the ORB_SLAM2 library

8.22 Planning Subsystem

Now that we have created a map of our environment, our next step would be to come up with a plan to traverse the environment. The Planning Subsystem focuses on creating an efficient path for traversing the environment and heading the the general direction of the destination GPS coordinate.

8.22.1 Global Positioning System

While the vehicle will be able to detect and avoid objects it still needs a method of determining its location in space and redirecting it towards a goal that is set by the operator. We will be using GPS to give it a heuristic technique of understanding how close the vehicle is to its end point. The GPS will not be used to plan the path of the vehicle otherwise it would not work in novel environments that are not mapped yet but allows the vehicle to slowly shorten the distance from itself to the point using GPS coordinates.

We could divide up our map, which we can obtain using Google maps or some other GPS API, into cells which will limit the amount of points that our heuristic function will be forced to path through using whatever implementation we decide on. The smaller the grid cells we create the more accurate the vehicle should be when plan-

ning its path but that also means more points for calculation and that may overwork our system. Even if we did not have a map of features like the one from Google then we could easily draw a box with one corner being goal and the opposite corner being the buggy's location and use math to turn the created rectangle into a grid.

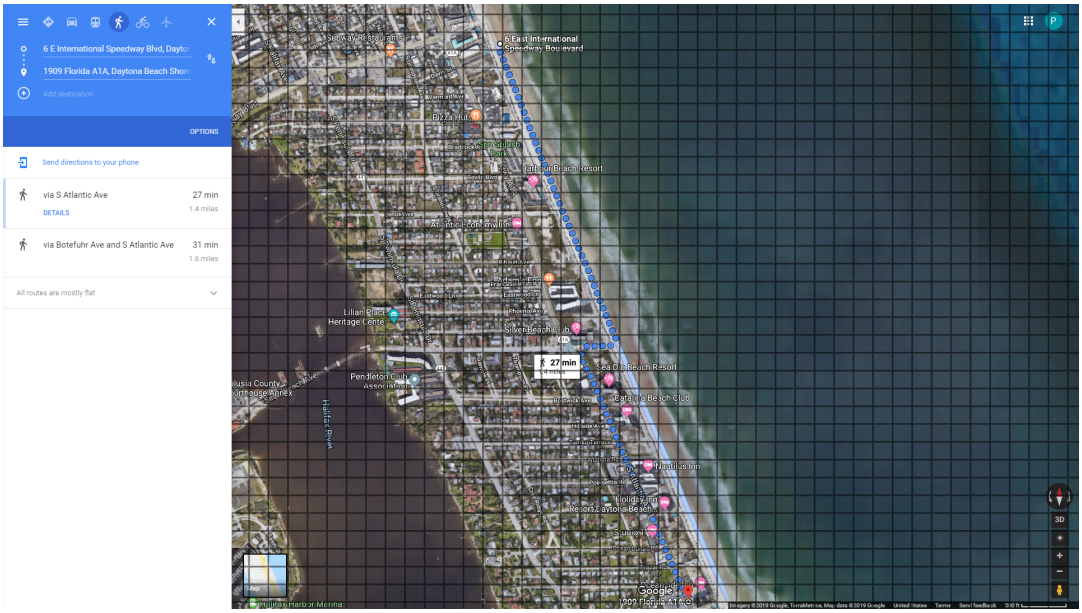


Figure 16: Grid Overlay of Google Maps Daytona Beach

Using a heuristic based on Manhattan distance the buggy would try to take paths in 90 turns to shorten the distance to the goal. This would be better if it were in the city because there are more often turns that follow the Manhattan distance.

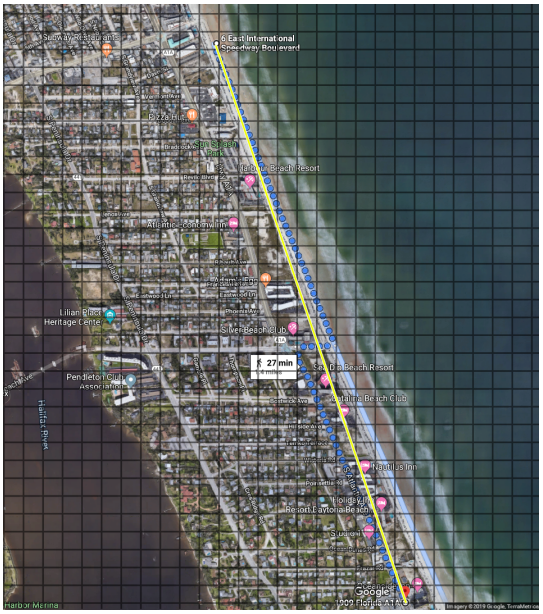


Figure 17: Euclidean path on Google Maps Daytona Beach

Using a heuristic based on Euclidean distance the buggy would try to take whatever path it can to shorten the distance to the final goal. This should work better on the beach because the beach has open paths generally from one end to the other.

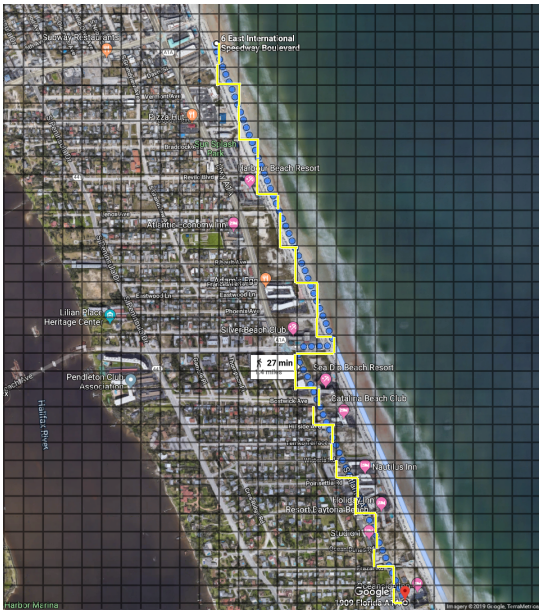


Figure 18: Manhattan path of Google Maps Daytona Beach

Using a map with features from Google Maps for our GPS path could give us a

better path. The default Google algorithm simulates how a human would traverse from one point to another taking the shortest distance that is still accessible using streets and roads. It essentially implements both Manhattan and Euclidean as it takes the Manhattan paths while in the city portion of the map but then switches to a straight Euclidean path when it gets onto the actual beach.

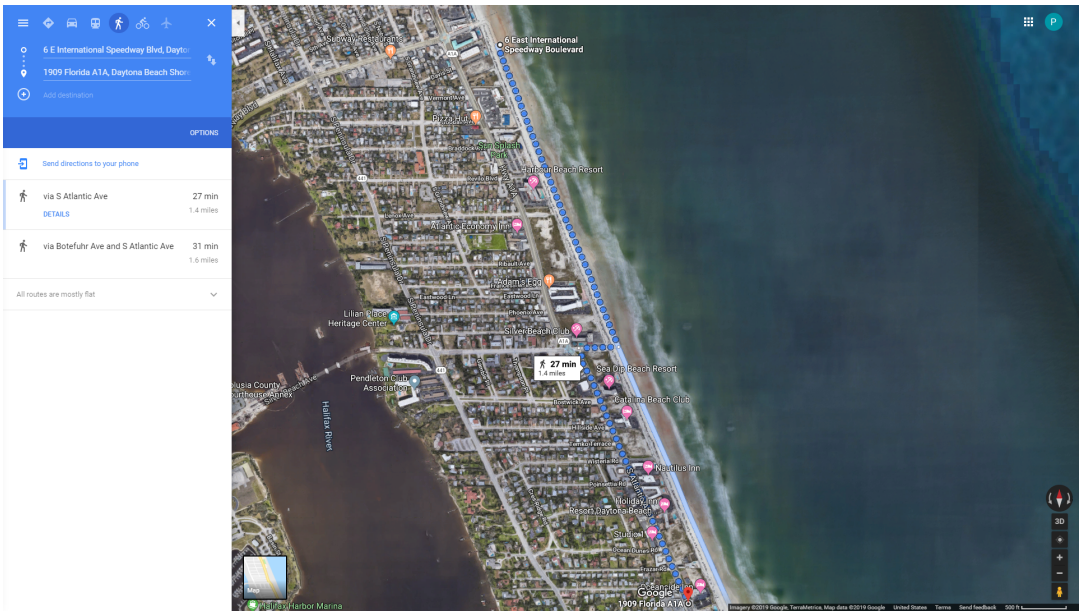


Figure 19: Satellite view of Google Maps Daytona Beach

The heuristic function we choose will have a large impact on the efficiency and actual ability of the buggy's ability to get to its goal. We will most likely have to combine multiple heuristic functions if we do not use an API and make sure that they are admissible and represent close to the actual cost.

8.22.2 Hybrid A*

An A* algorithm would be able to plan and find an efficient path to a goal, avoiding obstacles by dividing the field into edges that have costs and taking a low costs route that will still attain the goal, but our buggy cannot rotate in a circle to choose the edges around it instantly. A hybrid A* will avoid searching paths that the buggy cannot actually take and only consider paths that the buggy can actually take [3]. Since the environment is ever-changing, the Planning Subsystem will perform an

efficient hybrid A* search at each iteration. Once the search is complete and the path is has been chosen, a path vector will be passed down to the Control Subsystem for consideration.

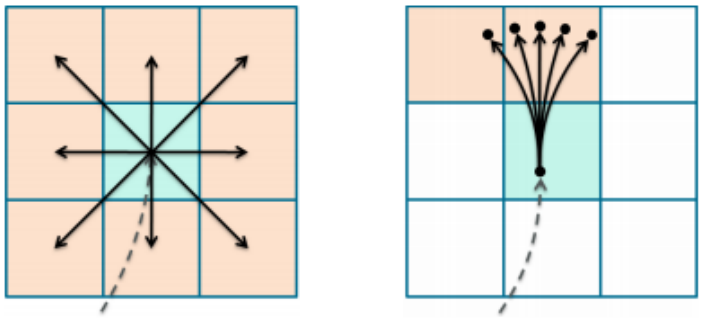


Figure 20: Demonstrating the paths considered by (a) A* vs (b) Hybrid A* [3]

8.23 Control Subsystem

Finally, we need to find out the right steering angle and throttle intensity so that our buggy can stay on its path. The Control Subsystem will be in charge of generating the instruction for the buggy to follow. This instruction will consist of two values: a steering value θ and a throttle intensity α. The Control Subsystem will generate this instruction by use of a convolutiononal neural network that performs regression. Once this instruction is generated, it will be published to the buggy's microcontroller which controls the buggy's motors and steering differentials.

8.23.1 Control Neural Network

The Control Neural Network will take in as input ultrasonic data, the path vector given by the Planning Subsystem, raw image, depth image, and the segmented image. The output of the neural network will be the steering angle θ and the throttle intensity α.

First, the raw image, depth image and segmented image will be fed into a ResNet convolutional base that has been pretrained on the ImageNet dataset. Once we have finished passing the input tensor through the base, we will then flatten the extracted features into a vector and concatenate the ultrasonic data and path vector. This will be the input vector to the fully connected neural network that will perform regression for the θ and α values.

We believe it makes sense to have our buggy instructions created by this neural network because it imitates how humans drive vehicles today. We use information that we gather from our sight about the environment around us and an efficient path that we get from some navigation system to safely traverse the environment to get to our destination. Using this approach, we will also be able to learn each buggy's imperfections and work our way around them.

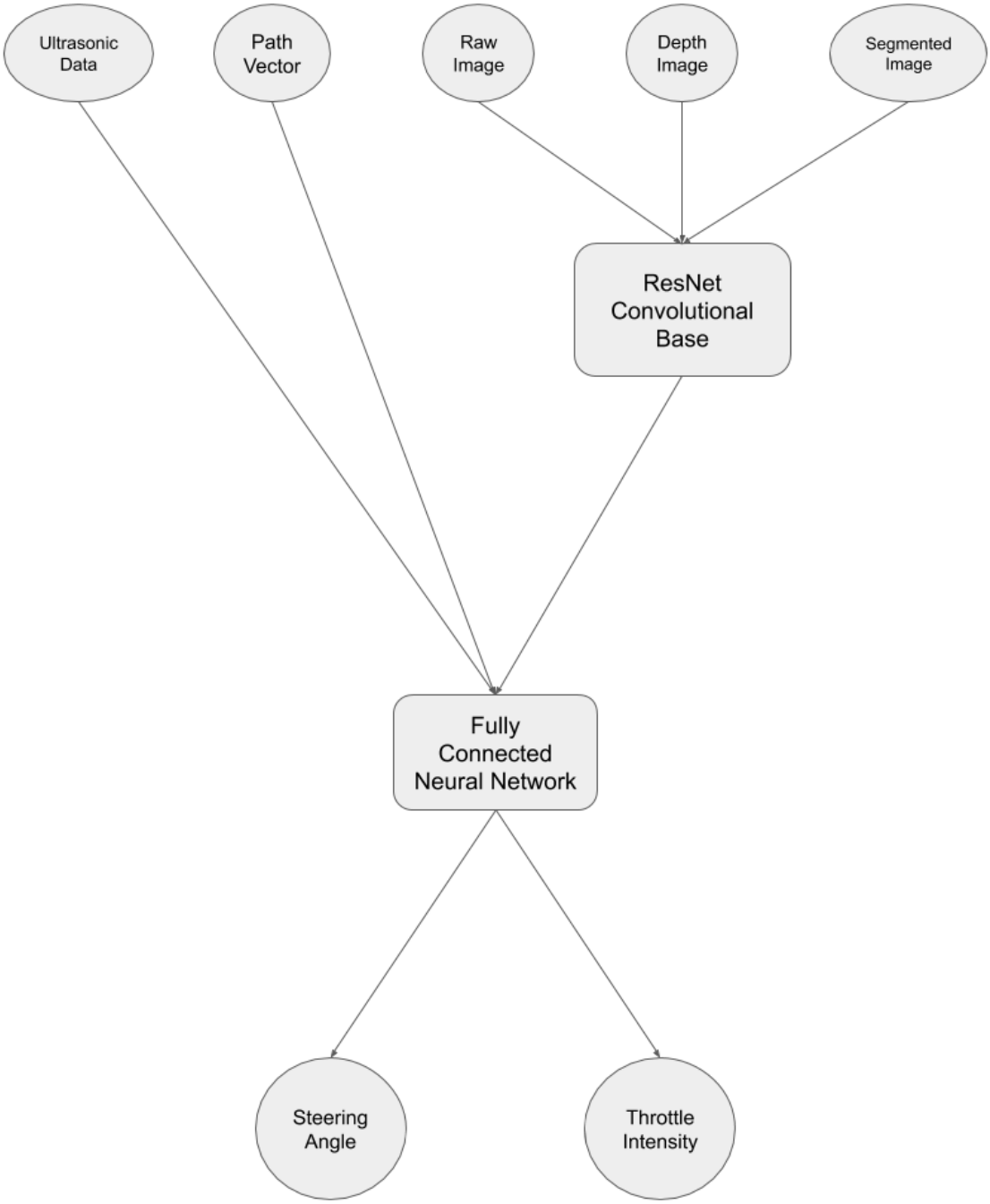


Figure 21: Overview of the Control Neural Network

9 Budget and Financing

Duke Energy sponsored every mechanical engineering team with 2000 dollars, 15 percent of which went to our computer science team. This meant that we had a total of 900 dollars for purchasing any parts and services that were needed to make our autonomous system a reality. Since we had to provide a brain to three beach buggies, we created a single solution that was used for each buggy and was purchased using our portion of the budget. This way, we only needed to spend money on one set of parts rather than spend it on one for each buggy. We also were able to get a MYNT EYE S that the purchasing department already had, which gave us a lot of breathing room in our budget. We ended up using only $476.47 of our $900 dollar budget, and we spent it on these items:

- Hardware for doing computationally intensive tasks (Computer Vision & Machine Learning):
  - NVIDIA Jetson Nano
    To be used for running object recognition and path planning
- Hardware to be used for the collection of the input data:
  - Elegoo HC-SR04
    Ultrasonic sensors to be used as a failsafe for stopping the buggy
  - MYNT EYE S
    To be used for determining depth of obstacles as well as their location relative to the buggy
- Hardware to be used for testing, connecting sensors, buggy, and Jetson:
  - Exceed RC 1/16 2.4Ghz Magnet EP Electric RTR Off Road Truck
    RC car used to test our autonomy system before the mechanical engineering buggies are finished
  - HiLetgo L293D DC Motor Drive Shield Stepper Motor

- XT60 Plug Male Female Connector with Sheath
- SunFounder PCA9685 16 Channel 12 Bit PWM Servo Driver
- ELEGOO UNO R3 Board

  Arduino board used for communicating between Jetson and motors
- L298N Motor Controller
- DROK Voltage Regulator
- Ovonic Lipo Battery
- HTRC Lipo Charger

10 Project Milestones

- 02/27/2019 - Beach Buggy Beach Simulator Planned
- 02/27/2019 - Simulated Sensors Planned
- 03/15/2019 - Dataset Labels Designed
- 03/27/2019 - Blocked and Directional Models Designed
- 04/22/2019 - Final Design Document
- 09/25/2019 - Beach Buggy Simulator Ready for Use
- 09/27/2019 - Simulated Sensors Ready for Use
- 10/15/2019 - Simulation Running Autonomy models
- 11/04/2019 - Software Integration with Buggy Teams Started
- 11/13/2019 - Buggy Testing Started
- 12/01/2019 - Project Completion

11 Design Summary

This Design Summary gives a brief insight on each of the larger components of our project's design. This includes the test vehicle, simulations, autonomy and interface control document.

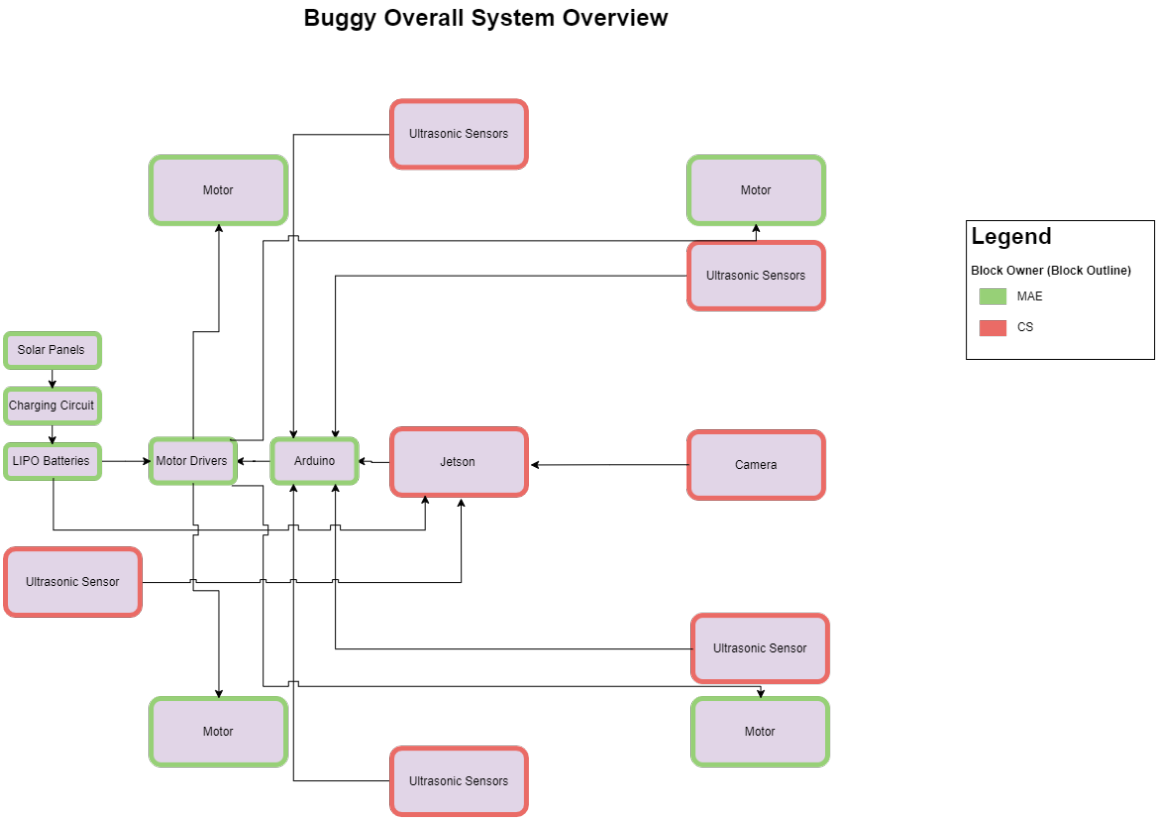


Figure 22: Overall System Design

11.1 RC Test Vehicle

A test vehicle was decided for usage in helping us test our algorithms and circuit diagrams on a smaller scale. It was decided that an RC car would be the most time and cost efficient way to fulfill our goal. We would not have to build a vehicle from scratch like the MEs but we would be able to learn how to use and adjust the Arduino while setting up a vehicle that functionally behaved the same as a beach buggy using similar modules and components.

11.2 Simulation

The entirety of our finished and running project needed to be simulated so that we might pass the class in the unlikely event that we did not end up with a physical buggy to test on. The simulations of the buggy, our module, and the environment that needed to be navigated was designed with efficiency in mind. This meant we used the robotics simulation software Gazebo with ROS that has a lower overhead than some other simulators and engines. Efficiency of resource use is important in our case because we will most likely need to make a lot of changes to the system in between running simulations.

11.3 Autonomy System

The Autonomy System is the soul of our project. It is the driver of each buggy. In this section, we talk about the sensors we use and how we use data from those sensors to generate instructions for the buggies to follow.

12 Interface Control Document

The Interface Control Document lays out how the mechanical engineering teams were able to interface with our system. It details where the ultrasonic sensors should be fixed to the buggy, where our system should be placed and how they should expect to receive control instructions.

The purpose of this autonomy system is to provide instructions for the buggies to follow to get to their destination without manual human control of the vehicle. to better accomplish this we, the computer science team, detail in an Interface Control Document how the buggies interfaced with our system. Our autonomy system is comprised of a Jetson Nano as our AI computing device, a MYNT EYE S as our depth camera, and an array of ultrasonic sensors for reliable close-range sensing. The first two components are stored in/on a safe and convenient box that protects our parts and makes for easy transfer. Values for steering and throttle are sent from this box to each buggy's Arduino via a "drive command" which consists of a float value for steering $\theta \in [-1, 1]$ and a float value for throttle $\alpha \in [-1, 1]$. θ values of -1.0, 0, and 1.0 mean max steer left, straight ahead, and max steer right respectively. α values of -1.0, 0, and 1.0 mean max throttle backwards, no throttle, and max throttle forwards, respectively.

12.1 The Box

To house our Jetson Nano and Mynt depth camera we created a box with dimensions 165.1 millimeters wide, 203.2 millimeters high and 152.4 millimeters deep. The box is fixed to the top-front of the vehicle such that the box's view ahead of the buggy is not obstructed by the buggy itself. This ensures our sensors on our box can collect the necessary data from the environment in front of the buggy.

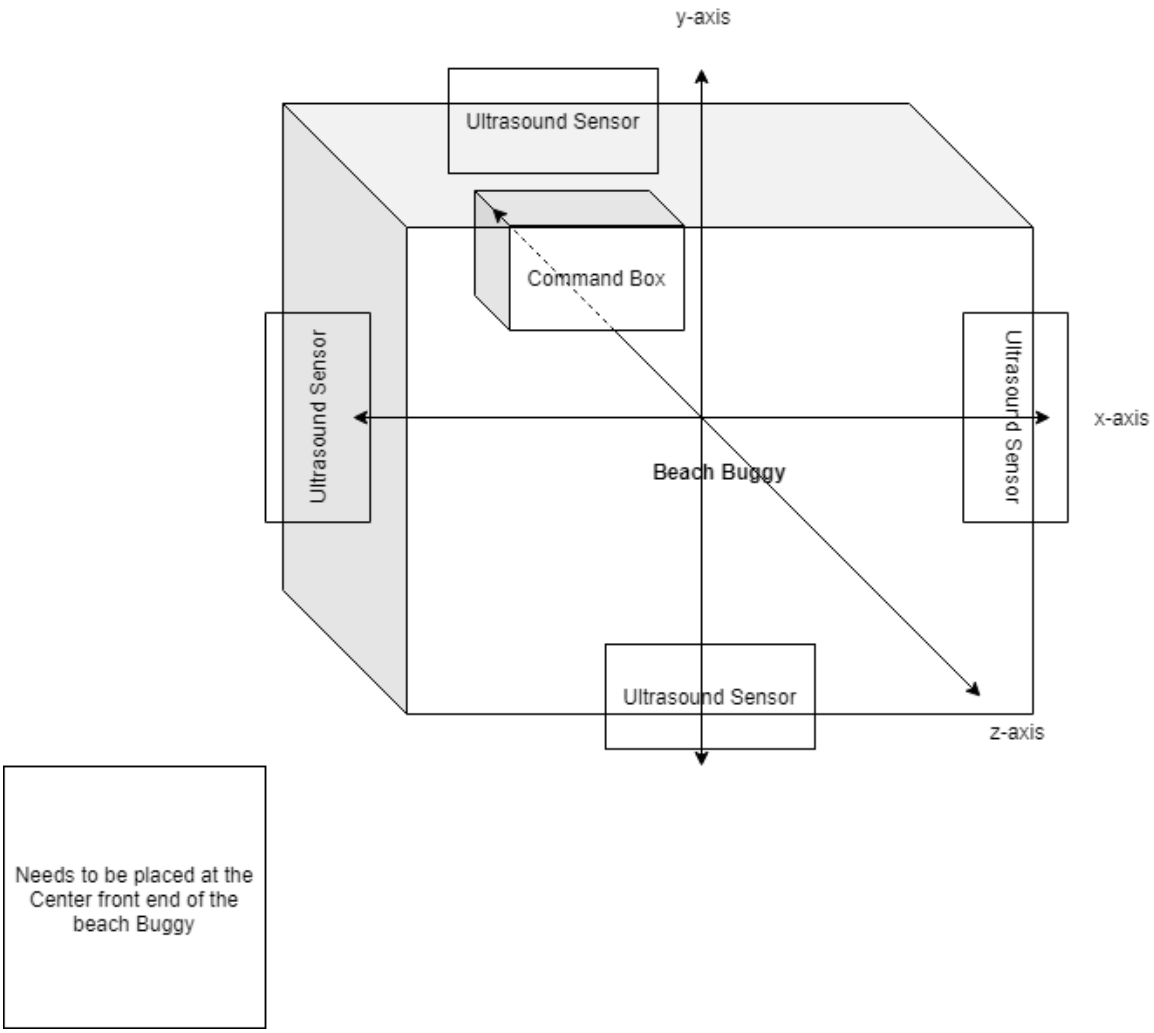
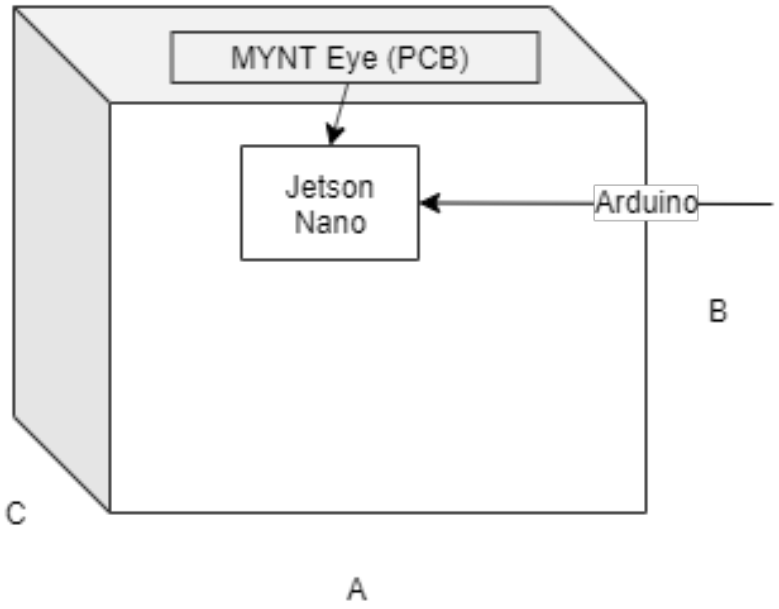


Figure 23: Placement of The Box (Command Box) on vehicle.



Sizes - Width x Height x Depth (mm)
Box - 250 (A) x 200 (B) x 50 (C)
MYNT Eye - 145 x 20
Jetson TX2 - 70 x 45

Weights - Grams
Box - Depends on Material
MYNT Eye - 80
Jetson TX2 - 50

Figure 24: Size of component and housing Box.

12.2 Ultrasonic Sensors

We required that the mechanical engineering teams have five ultrasonic senors fixed to the buggy as shown in the figure below. One ultrasonic sensor on the front of the vehicle will point straight ahead. There are also two sensors on each front corner and back corner. These corner sensors should be pointing outward at a 45°angle. Sensors should be fixed and wired for easy plugging into the box.

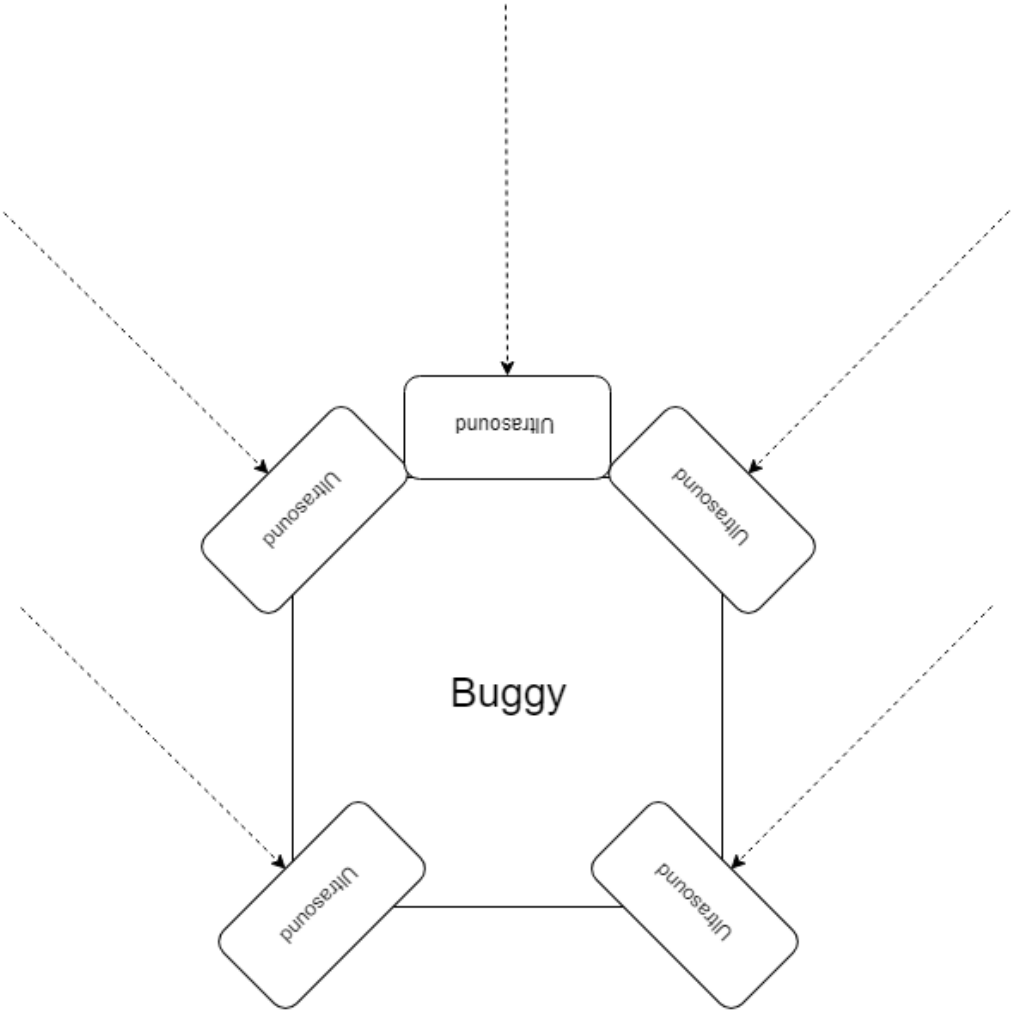


Figure 25: Forward Sweep with Point.

12.3 Power

Our Autonomy System demands power! The Jetson Nano, MYNT EYE S and ultrasonic sensors all need power to function. We only required buggies to be able to supply a max of 6 watts to the location of The Box for our Jetson Nano. We powered our sensors through the Jetson.

12.4 Control Instructions

We planned on having our autonomy system constantly publishing instructions from our Jetson Nano to a microcontroller that will be in charge of operating the motors and differentials of the vehicle. The microcontroller subscribes to these published instructions and will follow these instructions until it receives a new instruction to follow. An instruction will contain two 32-bit float values: a 32-bit float value for steering $\theta \in [-1, 1]$ and a 32-bit float value for throttle intensity $\alpha \in [-1, 1]$. θ values of -1.0, 0.0, and 1.0 would mean max steer left, straight ahead, and max steer right, respectively. α values of -1.0, 0.0, and 1.0 would mean max throttle backwards, no throttle and max throttle forwards, respectively. Although we helped them quite a bit, how these instructions are handled by the microcontroller to control the motors of the vehicle was up to the mechanical engineering teams to better suit the needs of their configurations.

Since we use ROS to send controls, we used a nice little package called rosserial to help us. to subscribe to these instructions from ROS, one can refer to this page in the rosserial documentation http://wiki.ros.org/rosserial_arduino/Tutorials/Blink.

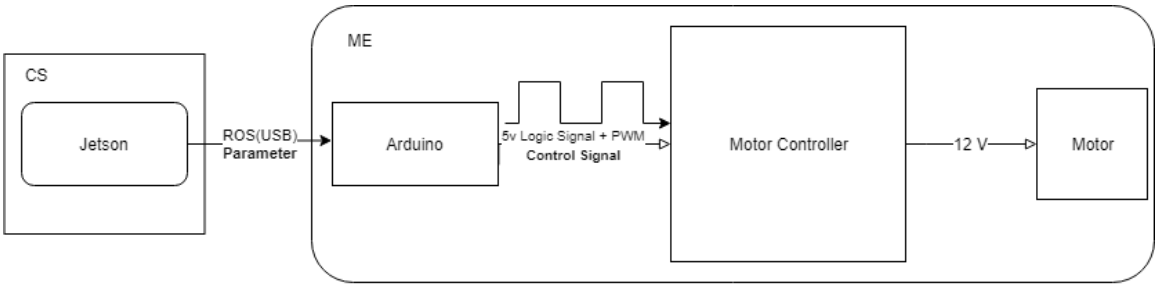


Figure 26: Control Structure

13 Simulations

One important factor of our project is the separation into teams based on discipline. This means that the engineers who build and design the buggy were not a part of building the module that controls the buggy, and our team was not involved in creating the buggy itself. For this reason, we were not be able to test the module's ability to process input information from the sensors on the buggy until shortly before the entire module was due. For this reason, to test our module's ability to process the sensor data, we needed to create simulated sensors that "collect" data from a simulated environment. By accomplishing this, we were able to create a model that was able to effectively traverse a simulated environment similar to the one the real buggy had to navigate through. When the real buggy was built we were hoping the module would be able to control it with little to no adjustments. Although the simulation ended up being different from the real world, it did help us to visualize our depth and ultrasonic sensor data.

13.1 Simulation Software

We chose to use Gazebo and ROS to create our simulation. These softwares were highly recommended, and the communication system of nodes, publishers, and subscribers was nearly identical to what we used on the actual buggy system.

13.1.1 Gazebo

Gazebo is considered an excellent environment for simulating robots that implement object avoidance and computer vision, and it is the software that we chose to use. It works in conjunction with ROS. We use ROS for communication between components, such as depth camera to Jetson and Jetson to Arduino. Gazebo can also use ROS services when simulating robots, and this was a accurate way of modeling how our buggy functions, even using the same signals and message system. This comes at no cost also, as both Gazebo and ROS are open source and free to use. In addition, Gazebo can be set up to be as lightweight as possible, and many of its plug-ins are

lightweight as well. Despite their advantages these technologies are hard to figure out and learning how to use them required a large time investment. There were difficulties getting the software set up and working properly, but with some tweaking and some equations learned in physics we were able to get it working. Outlined below are the components that needed to be simulated individually.

13.2 Simulated Ultrasonic Sensor

To create a simulated ultrasonic sensor, we needed to have some way of simulating sound waves that could be made in Gazebo. This was thankfully easy to implement, as Gazebo's built in library contains an ultrasonic sensor that we were able to use. The buggy was also programmed to stop when the sensor registers a short time period between when a sound is fired from the sensor and when it returns.

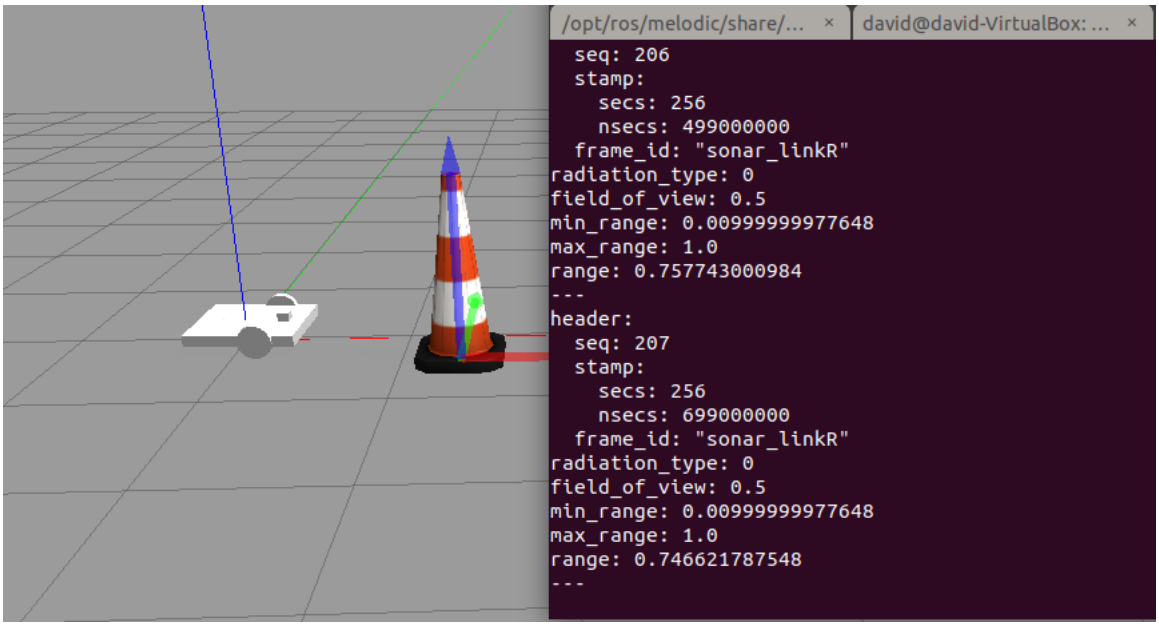


Figure 27: Ultrasonic sensor data being simulated in Gazebo.

The ultrasonic sensors were an effective tool in avoiding collisions with objects, especially in conjunction with the other sensors on the buggy. The sensors act as a last line of defense, being able to detect potential obstacles even when the other

sensors may not be able to. The buggy can then be stopped, and a new way around the obstacle can be determined based on input from each of the sensors. This way, even if our blocked detection algorithm or other method of seeing obstacles fails, the buggy is still able to avoid taking damage from a collision.

13.3 Simulated Camera Vision

The camera of the car is another component that needed to be simulated. The camera is probably the most important sensor and the one that the buggy uses to "see". This can also be simulated in Gazebo. This built-in camera is affixed to the front of a simulated buggy in the same spot the physical camera is on the real buggy. After "mounting" the simulated camera, it captures the images it sees onto files at a rate based on the frames in the simulation. These files include information about the depth of objects away from the camera. This provides enough data for an algorithm to run object recognition, and then the buggy can drive accordingly.

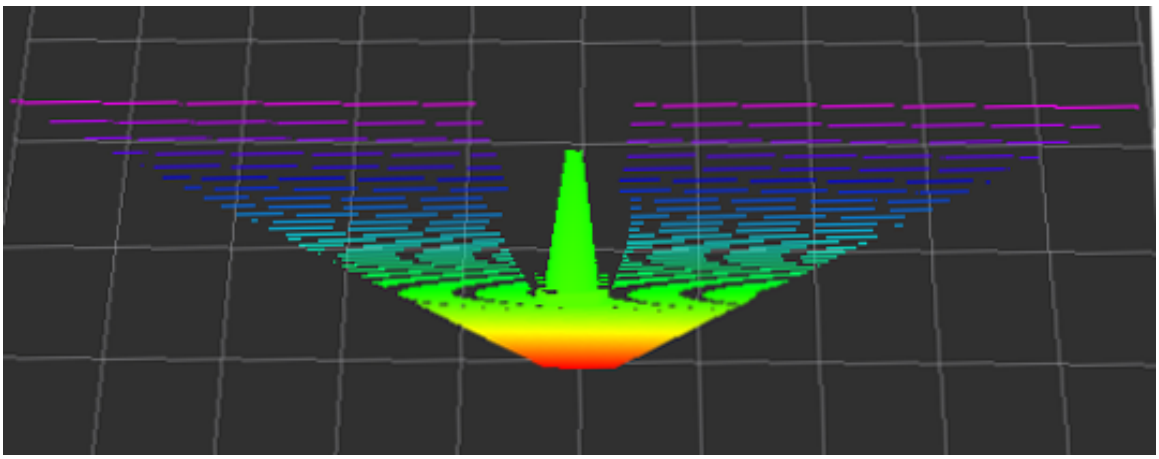


Figure 28: Depth camera point cloud in Gazebo.

13.4 Simulated Environment

If the sensors of the buggy are simulated, then an environment for the buggy to drive in and sense must be simulated as well. This was done quickly and effectively in Gazebo using their built-in physics system. Objects were placed into a "world" for

the buggy to sense and navigate within.

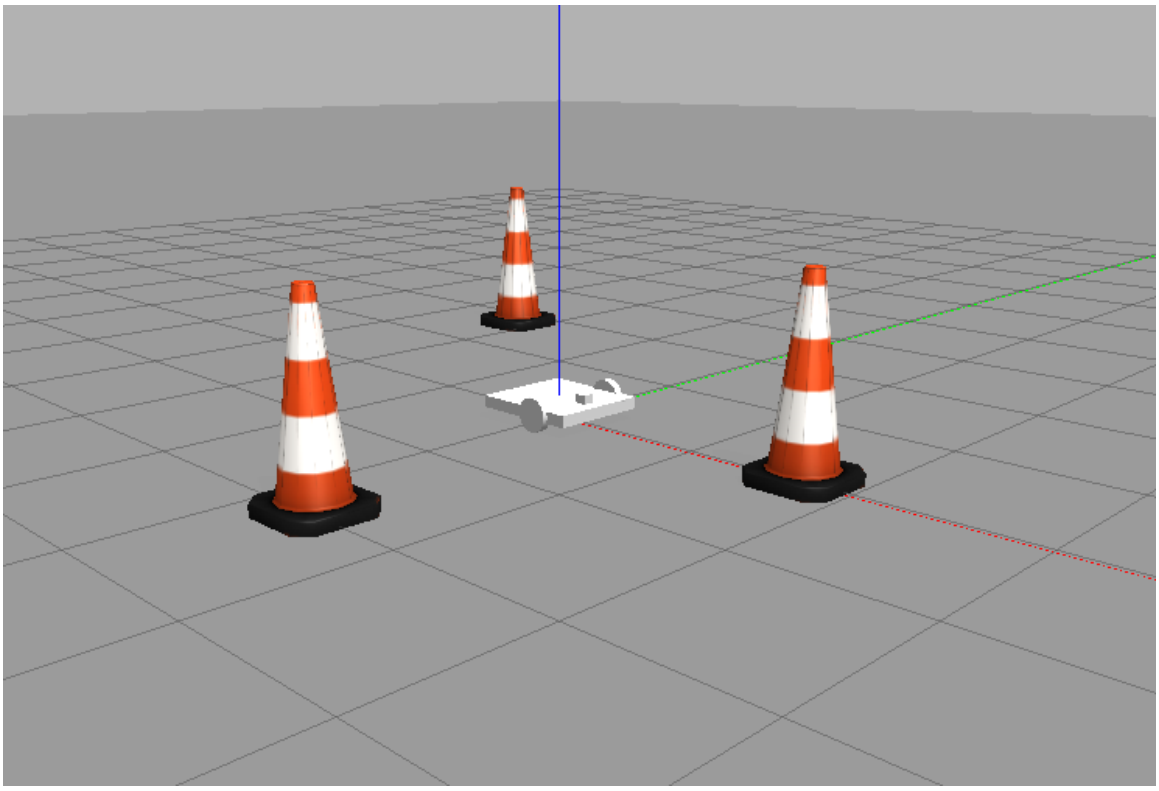


Figure 29: Gazebo simulated environment.

These objects are 3D models, and they have colliders that can be detected by our sensors. A depth image is created by our camera based on these objects, and so we can train our model based on these images before actually integrating with the engineers.

13.5 Beach Buggy Simulator

More important than the simulated sensors and even the simulated environment is the simulation of the buggy itself. All three designs had to be considered in the simulation. Time was of the essence however, and we wanted our simulation to be up and running early. Gold team was the only team whose dimensions we were able to get in September, and so theirs was the buggy that the simulation was based on.

The design was then simplified with a lower center of gravity and a rear caster wheel so to perform better and better represent a generalized buggy.

14 Sensor Suite

In order for the buggy to make good decisions autonomously, it must first be able to somehow get information about the environment. Without this information, our buggy would essentially be blind, deaf, anosmic, and tasteless. What nonsense! The Vision Subsystem is what the buggy will use to sense the world around it and "see!" The sensor suite will consist of an array of various different sensors, with these sensors being ultrasonic sensors and a depth camera. These sensors will be in charge of sensing the surrounding environment and collecting what is sensed as data to then be passed onto the Autonomy System.

14.1 Ultrasonic

These are sensors that use ultrasonic waves to measure distance. Sensors will emit a wave and then receive it as it reflects back from the target. The difference in time between emission and reception is what gives us a basis on how to find out the object's distance from the buggy. Sensors can use only one element for both the act of transmitting and receiving with an alternating pattern or they can have two elements for a more traditional approach. The distance of objects using ultrasound is given by this equation

$$L = \frac{TC}{2}$$

where L is the Distance, T is the time between emission and receptions (time is halved because it accounts for going there and back), and C is the speed of the ultrasonic wave.

Pros

- Inexpensive (compared to other modules it costs much less usually in the tens of dollars)
- Ignores Transparency (ultrasound will bounce off of transparent or liquid surfaces allowing it to detect windows or natural bodies of water)

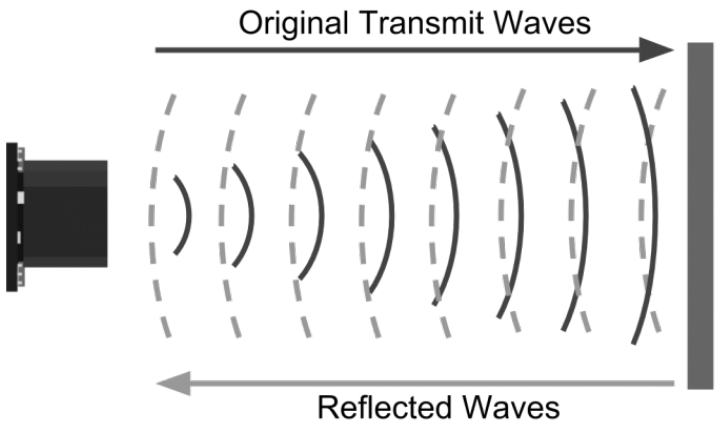


Figure 30: Sketch of the functionality of an ultrasound sensor, from [4]

- Precision (will detect complex shapes of all shapes sizes, colors, and reflective properties)
- Simple (easily integrates into most controllers while just requiring a single point and shoot)
- Light-Independent (can be used at night or in low brightness situations)

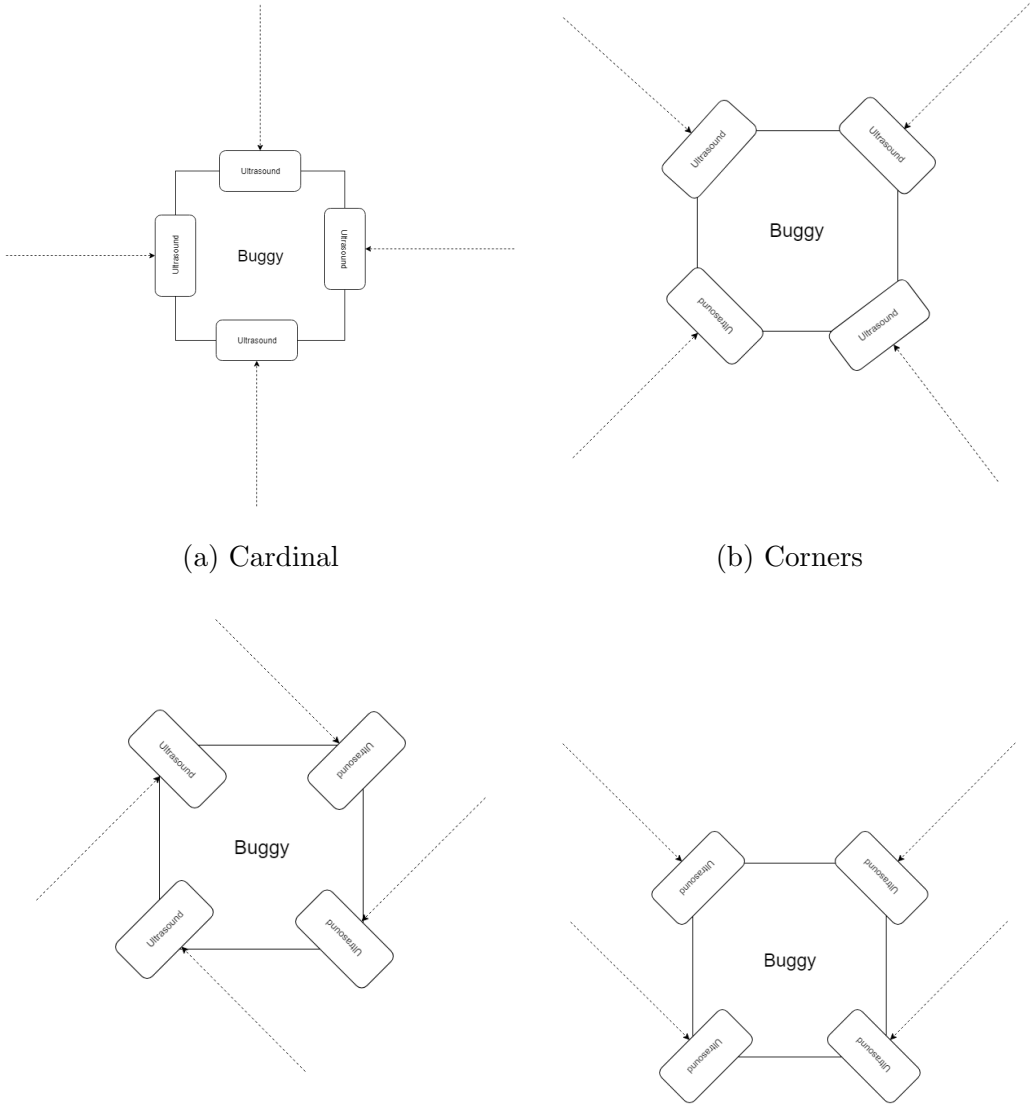
Cons

- Material-Dependant (soft materials will absorb waves making the object inaccurate)
- Limited Range (compared to other sensors ultrasound can not detect far off objects with a maximum of about 40 meters)
- Temperature (sound is affected by temperature with a change of 0.18% for every 1 °C a difference in temperature from the transducer to the target may affect accuracy)

14.1.1 Ultrasonic Sensor Placement

With ultrasonic sensors having a short distance and low measuring angle we have to think of different ways to face our sensors so that they cover the most ground while

also being practical. As we increase the amount of sensors we have, we can cover more fields of view but the cost will increase along with the chances of the waves interfering with each other and creating bad calculations. The sensors can be set to alternate but covering the same ground is ineffective in being efficient.



(a) Cardinal

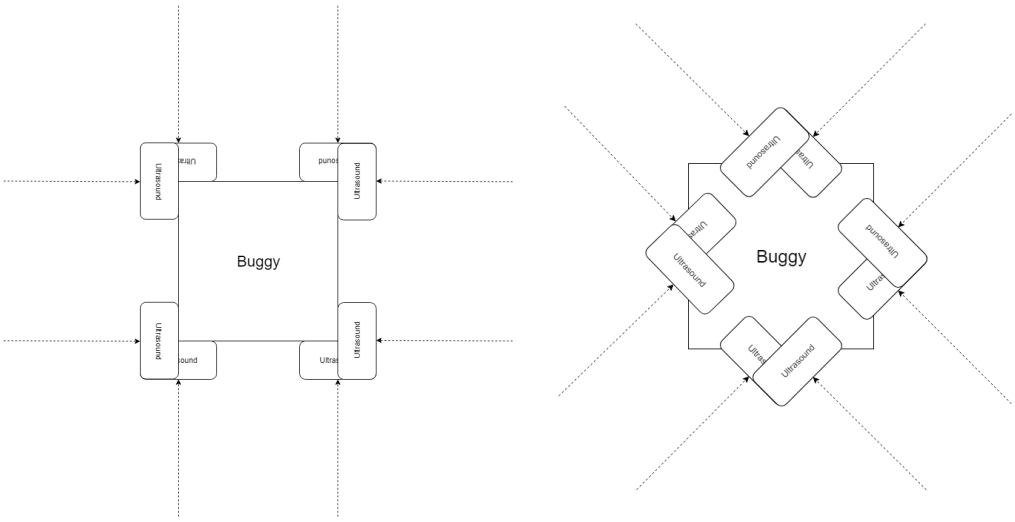
(b) Corners

(c) Angled

(d) Forward Sweep

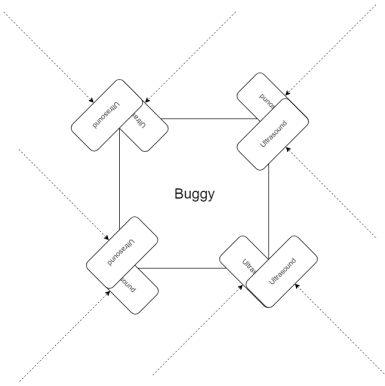
Figure 31: Four Ultrasounds Sensor Orientation

Here we have an additional four ultrasonic sensors which covers much more than having four sensors.



(a) Cardinal

(b) Cardinal Split



(c) Corners

Figure 32: Eight Ultrasonic Sensor Orientation

The one that is the most cost effective and safe is this five ultrasonic sensor positioning. It covers almost all view in a wide range to the sides and the blind spot beneath the camera. It cannot detect backwards but the assumption is that we will not need to backtrack. If we do plan to implement backwards movement then two or one sensors could be placed pointing backwards to verify there are no objects behind the buggy.

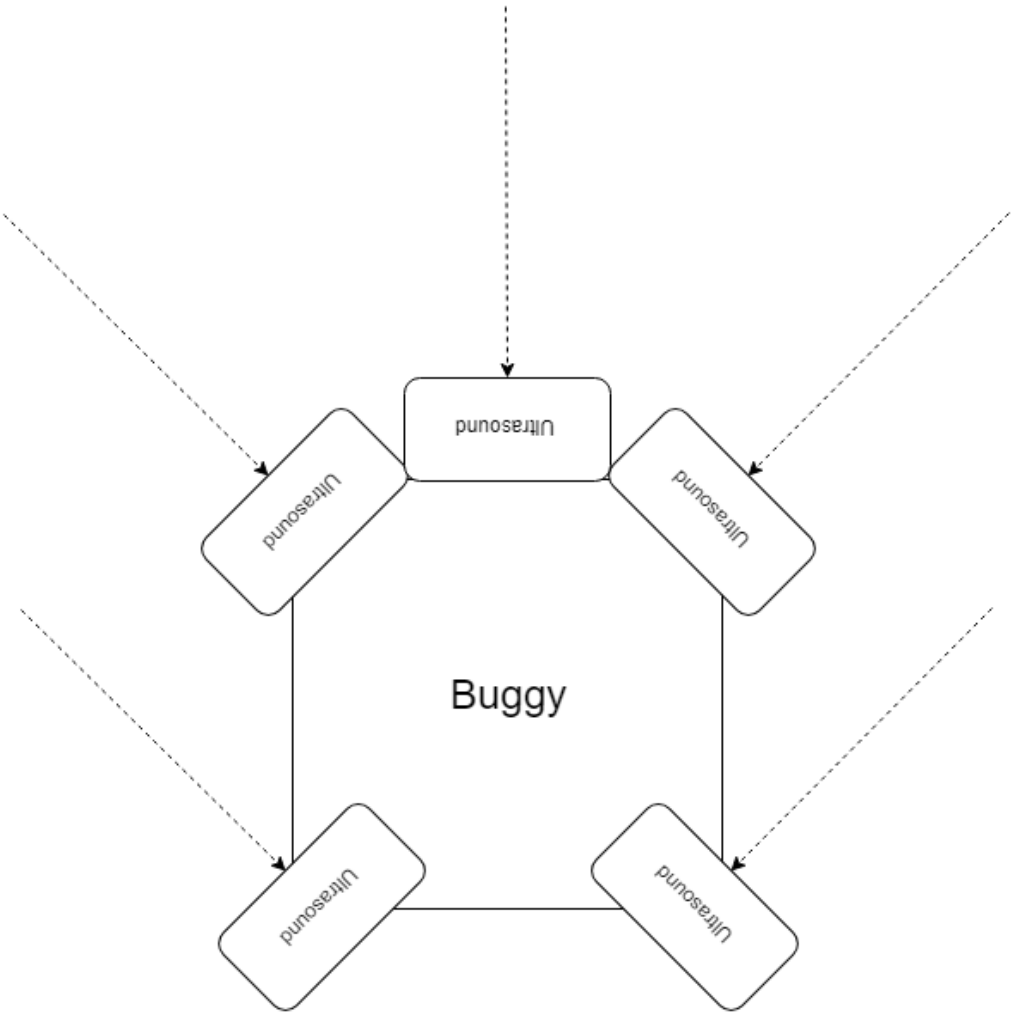


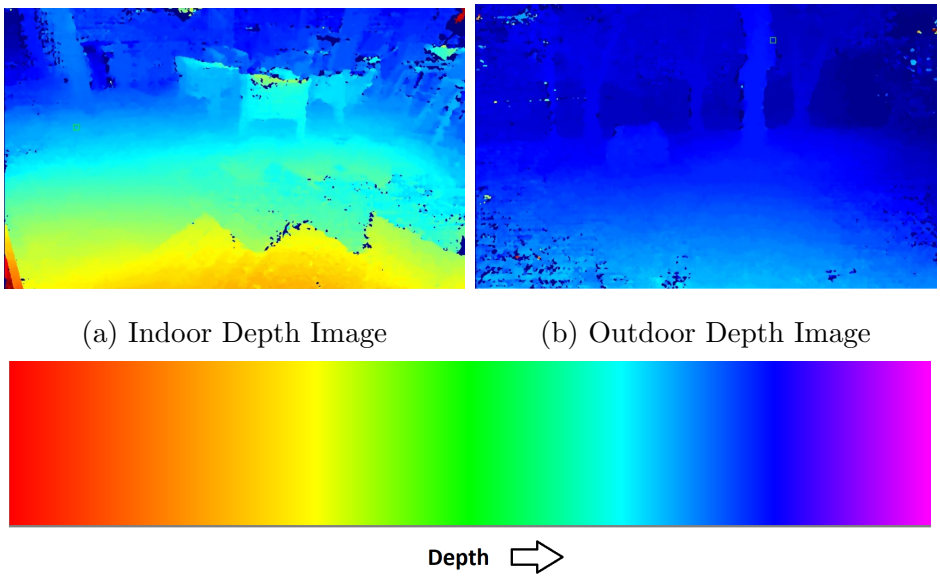
Figure 33: Forward Sweep with Point

14.1.2 Depth Camera

Using only a single camera can provide us with dense data about what it sees that we do not get with ultrasonic and lidar sensors. Depth cameras with multiple lenses allow for the calculation of depth of the environment. Depth data from depth cameras are typically only accurate to about 15 meters, which is great for detecting obstacles.

Figure 34: Image of the MYNT EYE S Depth Camera.

For our project, we plan on using the MYNT EYE S as our depth camera. We have become set on this piece of depth-imaging hardware as it has stereo cameras, an inertia measurement unit and it computes quality depth-encoded RGB-D images in the camera itself, taking away from the work needed to be done by our Jetson Nano. This device also will allow for easy integration with the VINS-Fusion library (used for odometry). Even with the The MYNT EYE S is also surprisingly small coming in at just 190 grams and dimensions: 145 millimeters by 20 millimeters by 28.6 millimeters.



(a) Indoor Depth Image (b) Outdoor Depth Image

Figure 35: Depth images from the *MYNT EYE S* depth camera.

15 Autonomy System

The Autonomy System is what generates drive commands autonomously. It does this by taking in sensor data about the buggy's environment and making decisions based on that sensor data. The Autonomy System consists of the following ROS nodes: the Joystick Node, the Autonomous Driver Node, the Filter Node and the Arduino Node.

15.1 Joystick Node

The Joystick Node listens to all messages coming from our Xbox One controller and maps those messages to messages that are meaningful to other nodes in the Autonomy System. Fig. 37 shows what we are mapping each button to. The left joystick and right trigger are used for controlling steering and throttle, respectively. We also toggle the sign of the throttle value by using right bumper button. The Y and B buttons are used for setting the Autonomy System to User Drive Mode and Autonomous Drive Mode, respectively. We also mapped some buttons for aid in the collection of our data set. We first set the label for the image we are about to capture using the back button (Label "Unblocked"), start button (Label "Blocked"), X button (Label "Left"), A button (Label "Right") and right joystick press (Label "Either left or right"), then capture the image using the down button on the D-pad.

15.2 Autonomous Driver Node

The Autonomous Driver Node subscribes to messages published by the Joystick Node. This node also subscribes to messages coming from the MYNT EYE S depth camera and messages that come from the Arduino Node (ultrasonic data). The Autonomous Driver Node has two modes used for passing on drive commands: the User Drive Mode and the Autonomous Drive Mode.

15.2.1 User Drive Mode

When User Drive Mode is selected, the node only publishes the drive commands given by the user (drive commands coming from the Joystick Node).

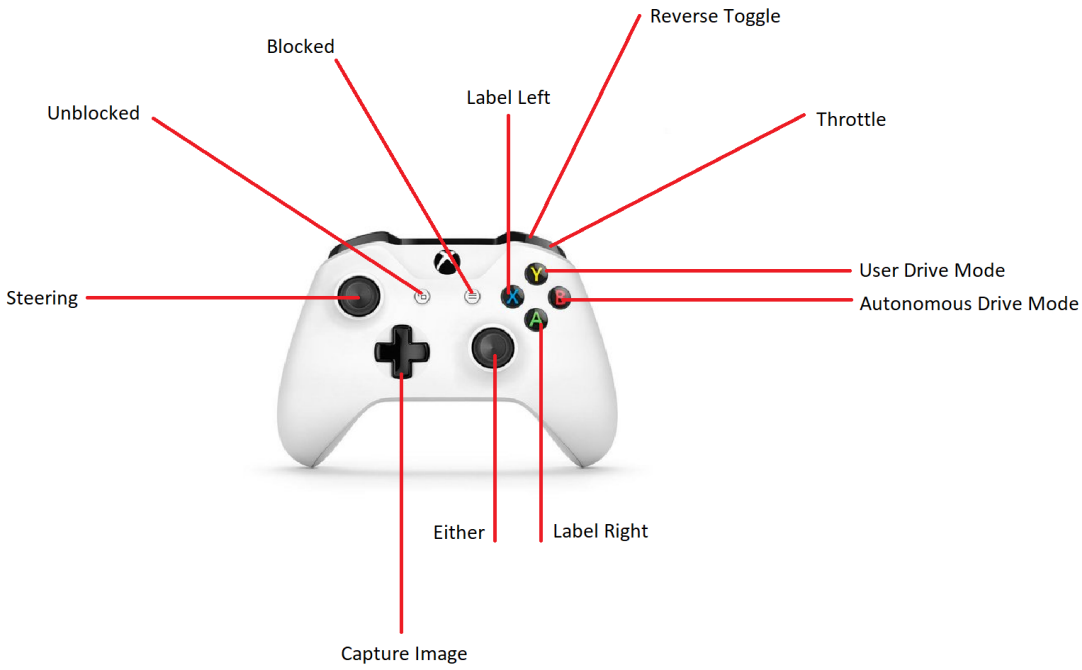


Figure 36: Xbox One contoller button mappings.

15.2.2 Autonomous Drive Mode

When Autonomous Drive Mode is selected, we make use of the Blocked Model, Direction Model and ultrasonic data that is given by the Arduino Node. If we ever receive an ultrasonic distance value of less than 150 (1.5 meters), we set the Autonomy System back to the User Drive Mode. Every time we receive a new depth image from our depth camera, we pre-process that image and have our Blocked Model decide whether the vehicle is in a blocked or unblocked state. If it decides the vehicle are unblocked, the node publishes a drive command with $\alpha = 1$ and $\theta = 0$. If it decides the vehicle is blocked, it passes the same image to the Direction Model to decide whether the vehicle must turn left or right. If it decides a left turn is needed, the node publishes a drive command with $\alpha = 1$ and $\theta = -1$. If it decides a right turn is needed, the node publishes a drive command with $\alpha = 1$ and $\theta = 1$.

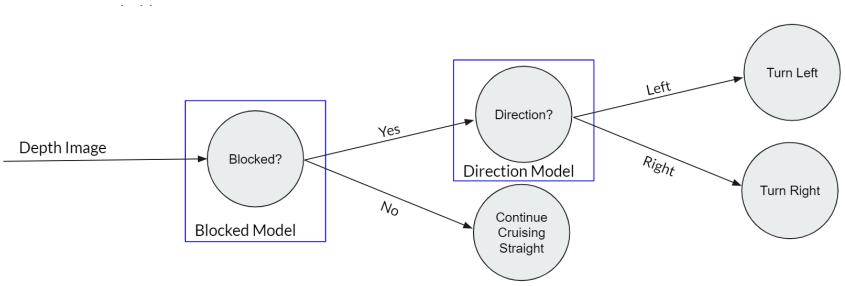


Figure 37: Autonomous Drive Mode flow.

15.3 Filter Node

To help ensure a smoother ride, we use this Filter Node to treat the drive commands as a signal and apply a low pass filter to that signal. This reduces the amount of jerk and puts less stress on the buggy's hardware. We implement this low pass filter by use of a weighted average of the previous drive command and the current drive command. The Filter Node proved to be helpful in simulation for smoothing the movement of the simulated buggy. The Filter Node was not as useful when we integrated with any one of the physical vehicles as friction seemed to be enough of a dampener.

15.4 Arduino Node

The Arduino Node subscribes and executes the drive commands being published from Filter Node. The Arduino Node also publishes an array of distance values (in centimeters) from each of the ultrasonic sensors that are fixed to the vehicle. This Node publishes its gathered ultrasonic data every 300 milliseconds to the Autonomous Driver Node. The logic required to successfully execute the given drive commands was to be implemented by each of the mechanical engineering teams.

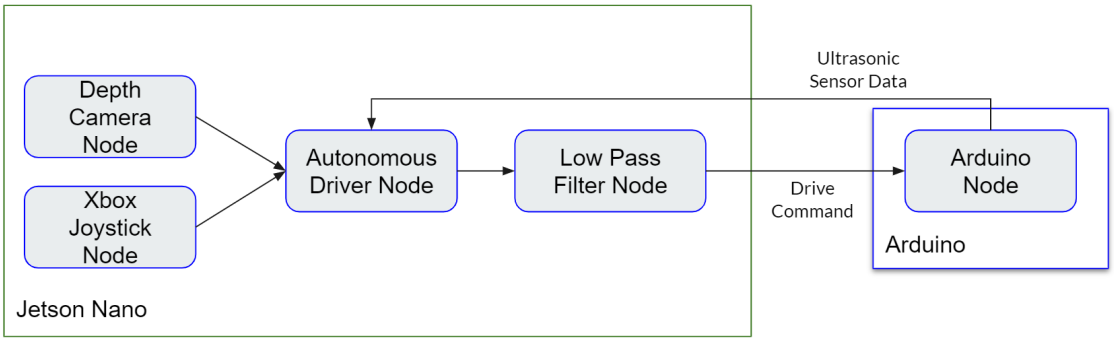


Figure 38: Overview of the Autonomy System.

16 Training

For training our machine learning models, we collected depth images from the MYNT EYE depth camera using an Xbox One remote to label them. The images could be labeled as blocked or unblocked, with other buttons labeling whether it would need to turn right or left when blocked.

16.1 Real World Data Collection

It is important that the depth camera image data used for training came from the actual depth camera sensor, as the simulated depth camera sensor cannot create depth images exactly like the MYNT EYE in the real world.

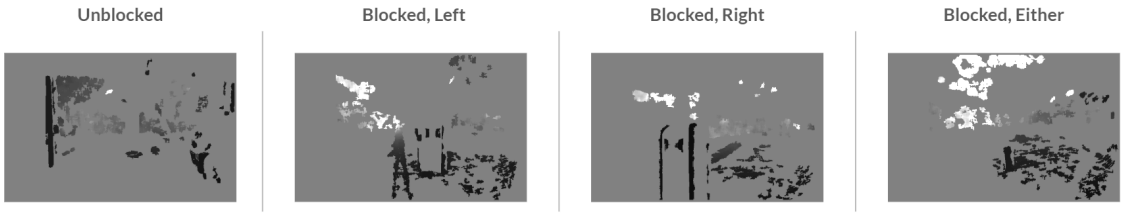


Figure 39: Different labels for our dataset with an example image for each label.

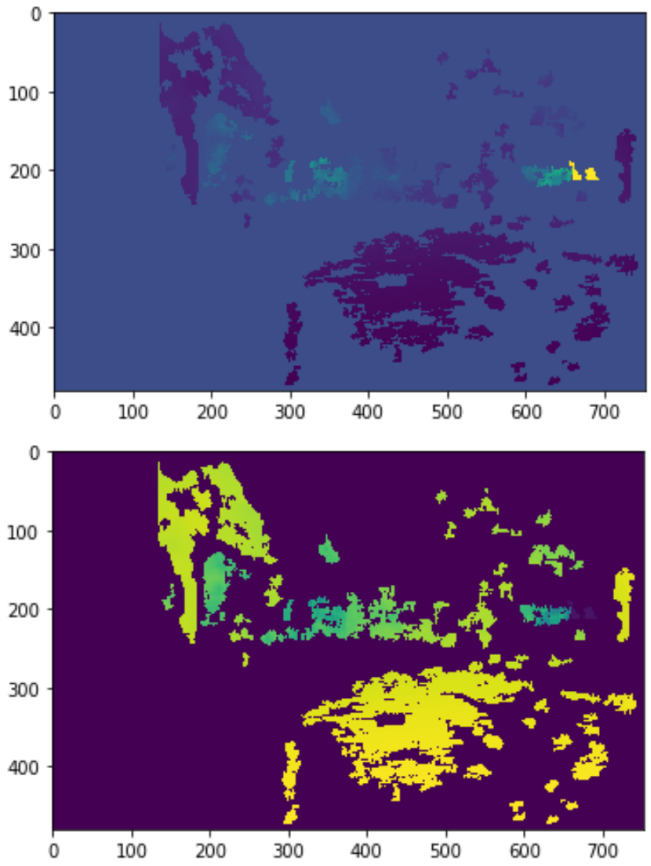


Figure 40: The top image is what comes straight from the MYNT EYE S depth camera. The bottom image is the same image after having gone through pre-processing.

16.2 Data Pre-processing

The depth image data that is produced by the depth camera on its own is not as effective for training of our machine learning models as it could be. This is because of how the depth is encoded in the image. Values in the collected depth images ranged from ∼250 (very close) to ∼35000 (very far). When the camera is not confident enough for the depth calculation for a pixel, it places a value of 10000 for that pixel in the image. This is a problem since this is within the range of values for valid depths and it is not certain that anything actually exists at the distance in front of the camera. To solve this issue, we create a new image D, and use the original depth image I to construct it by using the following equation:

$$D[i][j] = \begin{cases} I[i][j], & 10000 < I[i][j] > 10000 \\ 40000, & I[i][j] = 10000 \end{cases} \tag{3}$$

We place the same values in D that are in I only if they are not equal to 10000. Otherwise, we place a value of 40000 for that pixel in D. This essentially moves some of the uncertain data out of the way and has our machine learning models better understand the valid depths during training.

To help our machine learning models converge, we create a new image N by using the following equation:

$$N[i][j] = \frac{40000 - D[i][j]}{40000}. \tag{4}$$

Here, the max value expected to be in any image (40000) is subtracted by the value in $D[i][j]$. We then divide by 40000 to normalize the image by having all values be between the range [0,1]. The larger values in N represent the smaller depth values in the original depth image I. We do this because the smaller depth values from the original image are most useful for our neural networks.

16.3 Blocked and Direction Models

To generate our drive commands, we must know whether the vehicle is in a blocked or unblocked state. If blocked, we must also figure out whether we need to turn left or right. To make these calls autonomously we are using two neural networks: the Blocked Model and the Direction Model. Both models achieve ∼90% validation accuracy.

16.3.1 Blocked Model

The Blocked Model is a neural network that decides whether or not the buggy is in a blocked or unblocked state given the depth image from our MYNT EYE S depth camera. The input layer is a maxpool layer with dimensions 480 by 752 (the size of the depth image). The filter dimensions for this layer is 20 by 20. After this, the output from the maxpool layer is flattened and fed into a hidden layer with 16 neurons. This is followed by another hidden layer with 8 neurons that outputs to the output layer that consists of just one neuron with a sigmoid activation function that outputs a value in the range [0,1]. Any output value greater than or equal to 0.5 would be considered blocked. Any output value less than 0.5 would considered unblocked. This model is similar to that of a model provided by NVIDIA for their Jetbot [5], but we require an additional model for turning left or right to unblock the vehicle.

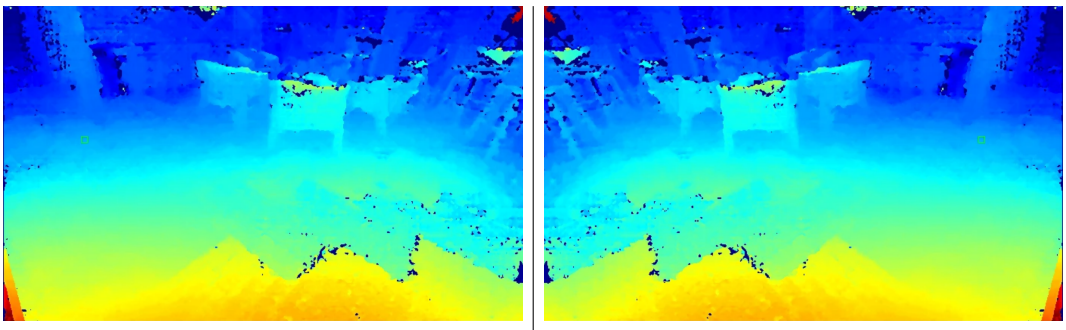
16.3.2 Direction Model

The Direction Model is a neural network that decides which direction the buggy needs to turn based on the input depth image. The Direction Model has the same structure as the Blocked Model. An output value greater than or equal to 0.5 is considered a need to turn right. An output value below 0.5 is considered a need to turn left.

16.4 Data Augmentation

Obtaining a set of weights such that the neural network is able to perform well on a wider range of input than given during training is difficult with a small data set. Unfortunately, we will not have the time to drive each buggy millions of miles to get a sufficiently large data set to train on. Fortunately, we can use data augmentation to create new data based upon the data that we have already collected. This will ultimately increase the size of our data set with "new" valid data. We will do this by performing transformations on our input data that would still be considered as valid input. Depending on the type of transformation we perform, we should still know what the input data's corresponding labels should be. We plan on using the following data augmentation only for training the Control Neural Netork.

16.4.1 Reflection



(a) Original Image (b) Reflected Image

Figure 41: Reflection across the vertical axis applied to a depth image from the *MYNT EYE S*.

We can perform reflection across the vertical axis to create input that would still be valid. This means that for every obstacle in the training data, the buggy will be trained to develop a path around it going either direction on that path. Although the input may still be valid, the input's corresponding label would not be correct as the steering would not be correct. We will account for this by manipulating the steering angle $\theta_{original}$ in the following manner to obtain the new steering angle $\theta_{reflect}$:

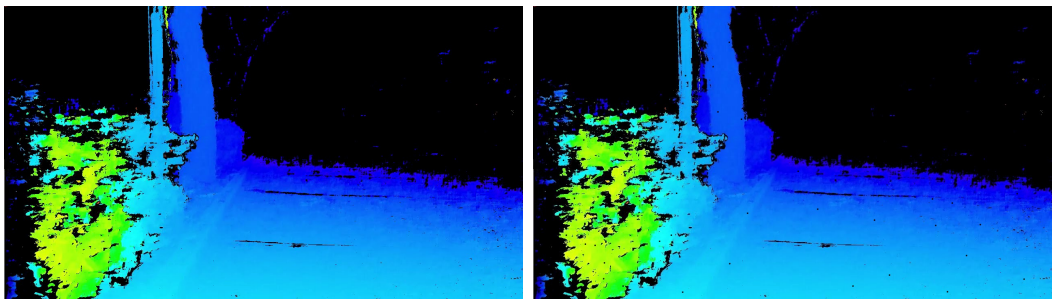
$$\theta_{reflect} = (-1) \cdot \theta_{original} \tag{5}$$

16.4.2 Rotation

While testing in the real we may encounter small bumps in our path in the literal sense. When going over these small bumps, our buggies may slightly tilt to either side, causing a small change in perspective that could completely throw off the performance of our models. To ensure our neural network models can handle these slight changes in perspective, we can augment our original data and apply a slight random rotation of $\phi \in [-0.5, 0.5]$ to mimic rotations that the buggy may face in the future in the real world.

16.4.3 Pepper Noise

Let us face it, the world is noisy. Not everything is perfect. Adding pepper noise so as to set the rgb value of any random pixel from our depth images with the value rgb(0, 0, 0). This technique may prevent the event where our model overfits by essentially throwing some wrenches into the training process by deleting some data from the input and having our models work their way around this obstacle. The corresponding labels to the new data should not change.



(a) Original Image (b) Peppered Image

Figure 42: Pepper noise applied to a depth image from the *MYNT EYE S*.

16.4.4 Gaussion Noise

For the raw images we receive from our MYNT EYE camera, we plan on adding a light gaussian noise during our augmentation process. This can help the model to learn how to function properly even in conditions where gaussian sensor noise might naturally appear in the input data, such as high temperatures or poor lighting.

16.4.5 Augmentation Process

Using these data augmentation techniques, we plan on creating a copy of the original dataset for each buggy and applying a single augmentation technique. This will make our training set much larger and cover a wider variety of conditions, creating a much more accurate model.

16.5 Neural Network Fine-Tuning

We need our model to be the best that it be, so we need to think about how we will squeeze every last bit of performance out of it. We used a learning rate schedule and checkpoints during the fine-tuning of our models.

16.5.1 Learning Rate Schedule

When training our neural network models, we plan on using a learning rate schedule. This means that for each epoch there will be a scheduled learning rate. As the number of epochs increases the learning rate will decrease in steps as shown in the figure below. We will do this to prevent any kind of overshooting while training and not get caught jumping around some loss minimum. We want to achieve as little loss as possible!

Figure 43: Learning rate schedule example (learning rate on y-axis and epochs on x-axis)

16.6 Checkpoints

We want the models that best perform on our validation dataset, meaning we want the model that performs best on data it has not been trained on. When training neutrals networks via data augmentation and stochastic gradient decent, we may find that after some epoch of training that the model does not perform as well on the validation dataset as it once did. Because of that we will use checkpoints. This means that if the model performs better than it ever has on the validation dataset after some epoch of training, we will save that model. After we finish training, we will just use the saved models that performed best.

17 RC Test Vehicle

To facilitate our integration with the MAE teams we created a small scale test vehicle using an RC car that utilizes the same command box we used for the actual beach buggies with a similar electrical set up. This allows us to physically test our code, wiring, and functionality. We used the electronic speed control and servo motor that came with our RC car, an Exceed Magnet, to drive the car. Those two were hooked up to a PCA9685 servo driver controller which was controlled using an Arduino Uno. Just like our actual beach buggies we pass the standardized throttle and steering values from our Jetson Nano to be interpreted by the Arduino.

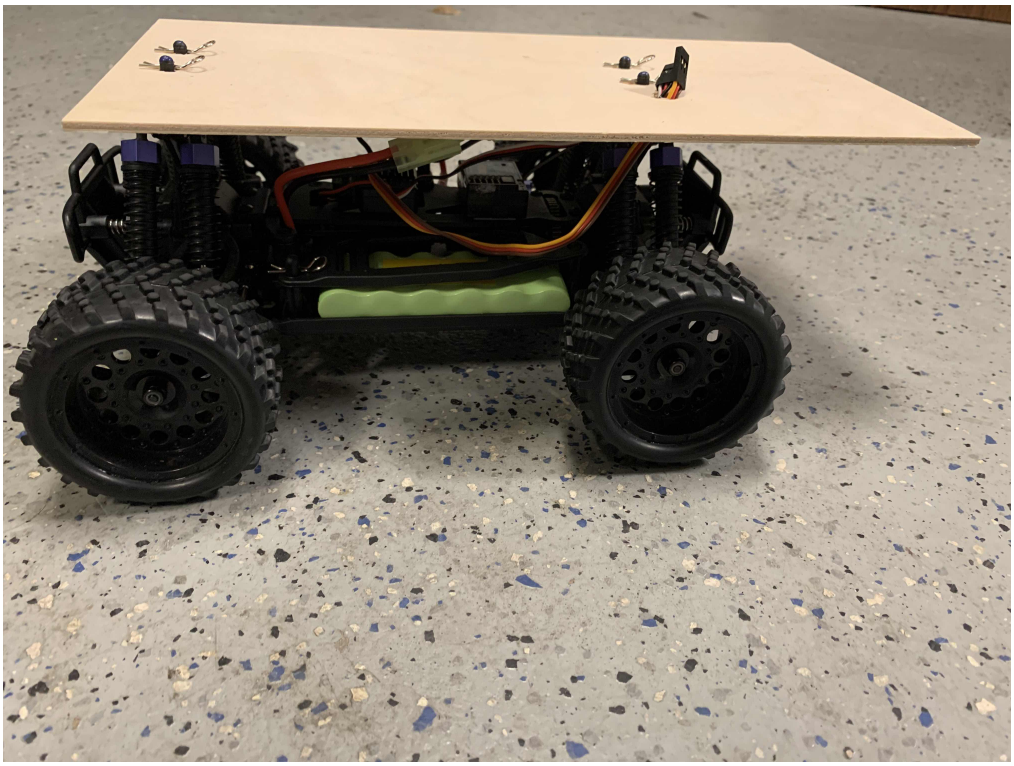


Figure 44: Image of the test vehicle without the Autonomy System mounted. (Side)

Figure 45: Image of the test vehicle without the Autonomy System mounted. (Above)

Figure 46: Image of the test vehicle integration station.

Figure 47: Image of the test vehicle with the Autonomy System mounted.

18 Hardware Integration

Our actual integration with each mechanical team was hectic. We had to match up schedules with a different team each week while making sure there was enough sunlight for testing battery usage and charging controller functionality. Each team before we met up would send us their Arduino code for general tweaking and installing of the proper functions and lines of code for ROS node functionality. They each had different configurations for steering and throttle so it made sense to defer the coding of their controls to each team individually. On the actual day of testing we would try to run each car but there was always bugs to grind out either in the code, the wiring, or even the physical components. Things would break and we had to wait while parts were reordered so we could test another day. The day before the showcase was spent visiting each car and making sure everything was working to maximum effect. In the end we got each vehicle to move and somewhat work autonomously with greater effect depending on the robustness of the vehicle architecture.

19 Design Execution

This section details the steps we took to fully realize our project and have an autonomy system that works in some capacity across three different buggies.

19.1 Build

All of our components were built with a plan and also some improvisation. Outlined below were the processes we followed when executing our designs. Also listed are a few bumps we experienced along the way, and how we dealt with them.

19.1.1 Simulation

Most of the building of the simulation consisted of creating one main package. Among some other metadata, this package contains a launch file and a URDF file that specifies all of the information about the buggy model. Thankfully Gazebo already has models and world files that exist to use for obstacles for our buggy. Although the launch file required tweaking, most of the work of building the simulation came down to creating the URDF file. This was split into different sections and some learning adaptation was required for every section.

The first part was creating the structure of the buggy itself. This all had to be done in URDF syntax, which is similar to XML. It was initially a hassle not having any sort of GUI to use for designing, but it was probably more effective to use URDF to ensure exact placement of components and joints. The final structure consisted of a buggy chassis, two wheels, and a caster wheel in the back. The module and ultrasonic sensors were added as physical blocks, and made into sensors later via plugins. At minimum a weight and moment of inertia was required for each component, and these needed to be calculated based on the actual buggy design. Torque was also needed for the wheels and their drive system outlined in the next paragraph.

The plugins were the next thing that needed to be figured out and incorporated into the build. There were plugins for the drive system, ultrasonic sensors, and depth camera sensor. Initially we were using the skid steer drive plugin, which assumed there

would be rear wheels on the simulated buggy. Because each team's drive system was different and skid steer functions in a relatively specific way, we switched over to a differential drive system with a caster wheel in back. This better generalized the real life buggies. The buggy drove erratically and with too much torque for awhile when trying to get either drive system set up, and the parameters required by the plugins needed to be changed until function was normal. The ultrasonic and depth sensors were comparatively easy to set up, it was just a matter of setting values for range, angle, and FOV.

19.1.2 RC Test Vehicle

Given that it was a relatively late addition to the plan, the test RC buggy's build required a lot of experimentation and improvisation. The vehicle was a mini project in itself, and required not only commands but also power and proper communication between components. Troubleshooting the issues that arose while bringing it to life was excellent practice for integrating with the mechanical engineering buggies, and shortened our integration time considerably.

The ability to send throttle commands was achieved early on, though controlling the steering was a problem. The buggy didn't seem to want to take commands, and we diagnosed a problem with the ESC. We discovered in research that the ESC needed to be armed by sending a PWM signal of a certain width and, given our tenacious attitudes towards solving the problem that night, we used a brute force solution to find the width of the needed signal. The ESC was armed, and we were able to get the components in working order so they could plug into the autonomy system.

19.1.3 Autonomy System

Planning for the autonomy system model architectures and function was probably the most effective of all the component planning. Setting up the networks involved Keras and Tensorflow, libraries that we had some experience working with. Collecting and using training data was done the same way throughout the process, using the Xbox controller mentioned previously

19.2 Testing

Testing our autonomy system meant testing it across all three mediums: the simulation, the test vehicle, and the actual buggies themselves. We achieved different levels of performance at all three levels, and having this many opportunities to test meant it came out performing quite well.

During testing the simulated buggy had some balancing issues, which were corrected before shipping off to be integrated with the autonomy system. When testing the autonomy mode on the simulation, performance was quite poor with the neural networks performing at about 50 percent. This could have been because of low resolution in the simulated sensors that was required for acceptable speed performance, or too little training data. Thankfully we started testing on the RC vehicle shortly after and performance was higher.

Then RC test vehicle performed very well throughout the process, and the only issues we had were with the ESC. Quite a lot of testing went into figuring out how to arm the ESC, something that needed to be done before the vehicle could be driven. Once this was done and the neural networks were trained on the real life data, the autonomy system was incorporated and performed quite well. The test vehicle dodged obstacles consistently, and even was able to dodge less solid obstacles like a stepladder and a window screen.

19.2.1 Integration with Real Buggies

We started meeting with one of our ME teams almost 3 weeks before the senior design showcase to try and smoothen the process of integration. Not all teams were ready that early, but this was just a product of purchasing delays and the learning curve of motor controller code for MEs. By our last meetings we had developed an effective system for testing wires and Arduino code. This made it easier to execute our integration plans and get more done quickly.

19.3 Evaluation

Shortly before the senior design showcase, we met with the three mechanical engineering teams to do final integration tests and collect some videos. Each team was able to get video of their buggy functioning in some capacity. With black team, we were able to get footage of the buggy navigating autonomously based on depth camera images and stopping when the ultrasonic sensors picked up something that was too close.

20 Changes

Projects this size almost never go exactly according to plan. We had a great design plan at the end of Senior Design 1, but a few changes needed to be made as we hit bumps in the road. Outlined below are the changes we experienced while creating this project.

20.1 Hardware

Changes in the hardware that makes up our system constituted the biggest changes in the project. Changes in plans, purchasing, and necessary functionality led to the swapping of parts on more than one occasion. Listed below are a couple of the changes that got made.

20.1.1 Switch from TX2 to Nano

We originally wanted to use a Nvidia Jetson TX2 for our module to run computations. We took great care to make sure that we would have space in the budget for this part, which was difficult as it is more expensive than our other components. We got lucky when we went to purchase the part, as the purchasing department already had one in stock that we could use. We took it home, but found out when trying to power it on that it was broken. We took it back, and did some more research because our budget was already getting close to the max we could spend while also buying a TX2. We found out that a Jetson Nano would probably be able to have just enough power to run our system, and so purchased that instead. It worked well, and not much had to change as far as our design to incorporate it.

20.1.2 Dropping Lidar

There was a time when we thought that Lidar would be the solution to all of our problems. It would be able to map an area like a depth camera, but in all directions. We looked at a lot of options for adding it to the buggy, but many were too expensive.

Ultimately, we decided to use a depth camera instead as it was still a powerful tool and worked well with our ultrasonic sensors.

20.2 Test Vehicle

Initially, we did not intend to have a test RC vehicle as a proof of concept for our project. We were simply going to use our simulation in the unlikely event that the mechanical engineers were not able to deliver. At the beginning of our Senior Design 2 class however, Dr. Heinrich advised that we incorporate an RC car as well as a simulation into our project. This was not overwhelmingly difficult, as the RC car's autonomy system functioned almost identically to the one attached to the buggies. It was definitely for the best, as it gave us good practice for when we integrated with the mechanical engineering buggies. Having two methods of proving that our autonomy system functioned properly helped us to sleep well at night, and through designing and bringing them to life we have learned a great deal about autonomous vehicle systems.

21 Problems

Over the course of creating this project we encountered a few different problems where we had to get clever to solve them. Designing and building a project of this size is difficult and the sheer number of options when it came to creating and troubleshooting its components caused problems in and of itself.

21.1 Simulation

The most difficult aspect in creating the simulation was the high learning curve. Our team had little to no experience with ROS, Gazebo, and Linux systems in general. Learning to use the necessary software components took nearly a whole summer. There were also bugs that naturally arise when trying to create a simulated physical system, and determining the cause and fix for these bugs took a lot of thinking, testing, and Googling. There were changes that had to be made along the way such as changing drive systems, the back wheel section of the simulated buggy, and the different weights and torques of components. These changes better reflected a generalized buggy, and like other parts of this project they demanded quick adaptation and learning.

21.2 Budget Constraints

Another issue with a project of this size is the cost of all of its components. Each mechanical engineering team designing a buggy was allotted only 2,000 dollars to use, and some of this money had to be used by our computer science team. We decided upon 15 percent, meaning that our system needed to be designed only of components that cost less than 900 dollars in total. This changed a huge portion of our design, as we were planning on using Lidar as one of our main sensors. Although this was a good idea, it would have been too expensive to implement and stay within our budget. To solve this problem, we changed our design to one that utilized a Mynt depth camera along with ultrasound sensors to feed input into our Jetson. These components fit just within our 900 dollar budget, and upon discovering that UCF already had the

Mynt camera we then had no issue with expenses. Although we got lucky with the depth camera, it was late enough that we could not go back and incorporate lidar into our design.

21.3 Coordination with Other Teams

We have been blessed with having three capable teams of mechanical engineers building buggies for us, but this means that we have had to maintain communication about our project designs. This communication was especially important regarding the aspects of our systems that interact, specifically how our module passes control commands to the buggies. This was established early on in our design process, and caused some confusion for our teams who forgot later on. Which team was supposed to buy and program the motor controller was at first ambiguous, and so we had to call a meeting with the mechanical engineers as well as discuss with our advisors which would be the best. We decided the mechanical engineers would handle the motor controller as they also handle the motors themselves. This way our team can save some of our budget and we will be able to send standardized commands in ROS straight out of the Jetson to the controller.

Although we collectively decided to have the job of programming the Arduino given to the mechanical engineering teams, we ended up doing an overhaul of almost every team's code. Learning how Arduino sketches work was a difficult task for the engineers, perhaps too difficult. In the future it may be a good idea to have this job be for a different team, or encourage the mechanical engineers to start the process earlier.

21.4 Purchasing Delays

A massive roadblock our team as well as the Gold engineering team faced while creating our project was the MAE purchasing department and process. Our orders and shipments were often delayed, Gold team was denied purchasing due to the department's confusion about the budget, and one of our shipments was lost after

being delivered. We were told our lost parts were lost in shipping, and we spent a great deal of time going back and forth with Amazon and UPS before finding out the shipment had been lost after being delivered to UPS. These kinds of delays would not be a huge issue if the project were not so time-sensitive in nature. If we were to do this project again we would take care to put in orders earlier and to try and get our advisors involved so as to be taken seriously.

21.5 Ambiguous instructions

Though creative freedom is to be appreciated in most any project, the laxness of the instructions we were given when starting the driving system was initially a problem for us. The instructions were to build a buggy that can avoid obstacles and drive along the beach for 20 miles. This soon changed, however, as during our first meeting with our sponsor and the mechanical engineers we learned we would actually be driving on or near campus. We were not given directives in what sensors we needed to use to help the buggy drive. Knowing these sensors would be used for avoiding obstacles, this change in testing environment caused a change in the types and appearances of the sorts of obstacles we might need to avoid as well as the sensors we wanted to use. We also had issues determining our budget, as certain parts we were told may be available for us to use that were purchased by previous teams. This uncertainty about which parts were available and which were not led us to design a system assuming no parts would be available. If we had known earlier, we might have been able to use some of the money we saved to build a more complex and potentially more effective system.

21.6 Implementation Conflict

One issue of having one team for three mechanical engineering teams is the fact that we cannot possibly do the equivalent of three CS teams of work. One of the advisors for this project wanted to create three brains, one for each vehicle. There was a fear of attempting to demonstrate a vehicle but the one module was on another vehicle

forcing the requester to wait. The most optimistic view was that we would split into three groups, one computer science team to each mechanical engineering team, and develop completely different systems to control the vehicle in a manner that allowed the vehicles to actually compete. Another solution recommended was to create three identical modules. The problem was that in all these cases it forces one student to shoulder all of the work essentially locking them down to working with a single team and becoming their single point of failure. It was decided that having the CS students work together and create one interchangeable module would be in spirit of the assignment. Once the showcase came around, we did not have the problem of trying to run all three buggies. Nobody was told to do a demonstration and the buggies were merely placed in front of the booths for display.

22 Project Conclusion

Through hours of hard work, we were finally able to achieve our goal for designing and creating a module that could autonomously deliver steering and throttle instructions to an Arduino. This concept was proven by reliably driving our test RC vehicle around obstacles as well as sending commands to avoid obstacles to the ME buggy. These commands were based on depth camera data as well as ultrasonic readings, just as we had planned last spring. We hope our example will show that autonomous and solar powered vehicles are viable projects, even on a tight budget.

References

[1] David Kohanbash. *Perception in Smoke, Dust or Fog*. [Online; accessed March 3, 2019]. 2016. URL: http://robotsforroboticists.com/perception-in-smoke-dust-or-fog/.

[2] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars." In: *CoRR* abs/1604.07316 (2016). URL: https://arxiv.org/pdf/1604.07316.pdf.

[3] J. Petereit et al. "Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments". In: *ROBOTIK 2012; 7th German Conference on Robotics*. May 2012, pp. 1–6.

[4] Roderick Burnett. *Understanding How Ultrasonic Sensors Work*. [Online; accessed March 3, 2019]. 2018. URL: https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm.

[5] Nvidia AI. *Jetbot Examples*. URL: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Examples.