ECE 129 Final Report

ECE 129C: Capstone

Scott Oslund
Robert Box
Vela Rajesh

June 16, 2023

Contents

Abstract

In this paper, we present 'Cruize Software', a groundbreaking, full-stack, self-driving car software framework specifically crafted for research on the Nvidia Jetson platform. Cruize Software offers a lightweight, intuitive, and scalable framework, designed to be interoperable on a variety of compute boards. This adaptability allows researchers to develop and test novel algorithms for specific aspects of the self-driving car software pipeline, eliminating the necessity of rewriting the entire pipeline from the ground up. While comparable software stacks do exist in simulated environments, as of now there's is no open-source software framework that operate on tangible, real-time self-driving cars. Cruize Software fills that gap, providing this essential functionality without compromising the intuitiveness and simplicity associated with simulation software. Through rigorous testing in both simulation and on a small-scale testing vehicle, along with customer feedback, we have ensured the quality and reliability of our software stack. Our innovative system will revolutionize how researchers and engineers conduct future explorations in the domain of self-driving car technology.

1 Introduction

Road conditions across the globe are dangerous due to human drivers, whose limitations often lead to unsafe roads, hazardous conditions, and congested traffic. Unlike computers, human capabilities are not constant nor computationally powerful. People inevitably get tired, lose attentiveness, and find their focus wavering. According to estimates by the National Highway Traffic Safety Administration, approximately 40,000 fatalities occur each year due to car crashes, with almost half involving drunk drivers, and over-speeding [7]. In light of these issues, self-driving cars, the holy grail of AI research since the early 1970s, present an attractive solution. Offering promises of safer, more efficient, and relaxing travel, autonomous vehicles offer a compelling case for the future of transportation. Unlike their human counterparts, self-driving cars would never lose focus, would consistently act rationally, and would always adhere to traffic laws. While it is true that computers can still be prone to glitches and face challenges in accurately perceiving their environment, they are expected to eventually outperform human drivers as technology continues to advance. In the long run, this progression will lead to a reduction in driving-related deaths, save significant amounts of time, and result in less road congestion. This would be facilitated by the superior ability of computers to communicate across vehicles and coordinate actions more efficiently than humans.

The promise of self-driving cars has been very influential, prompting researchers to dedicate substantial effort towards the development of robust autonomous vehicle technology over the past century [31]. This momentum has surged in recent years, spurred by significant advances in machine learning and deep neural networks. The field is continually evolving, with new methodologies and algorithms being developed constantly. However, the testing and validation of these algorithms prove challenging, subsequently limiting progress. A principal challenge in autonomous vehicle development pertains to the complexity of the

overall system [17]. Self-driving cars necessitate real-time environmental perception, accurate predictions of future surroundings, robust handling of potential edge cases such as road work or accidents, and continual updating of dynamic environmental maps [26]. Each of these components forms a cog in the broader machine, making it a prerequisite for every sub-module to be in place and capable of integrating with newly developed algorithms. This interdependence complicates the process of testing individual modules or algorithms. In addition, access to viable research platforms for testing self-driving software is another hurdle [31]. Simulation testing presents an alternative to on-road trials with real vehicles. However, these simulated environments do not capture the real world's complex dynamics, timing constraints, and visual complexity [31]. As a result of these obstacles, researchers often face significant challenges in validating their work in the field of self-driving cars[17]. Thus, it's crucial to develop robust, accessible, and accurate testing methodologies to ensure steady advancement in the realm of autonomous vehicles.

The Hybrid Systems Laboratory at the University of California, Santa Cruz (UCSC) is under the leadership of Professor Ricardo Sanfelice, and specializes in the formulation of innovative control and planning algorithms for self-driving cars. To test and validate these new algorithms, the lab has initiated the development of a small, self-driving electric vehicle, slightly larger than a golf cart, which will serve as a tangible testing platform. Equipped with a drive-by-wire system, this prototype allows for remote operation without direct human intervention. However, more research and development is needed to create essential features necessary for autonomous operation. The vehicle is equipped with sensors required for perceiving its environment, and has a compute board for processing and decision-making. Currently, the software necessary to manage and control the car has not yet been developed. While a dedicated team within the lab is already addressing the issues of sensor integration with the compute board, a significant gap still exists in terms of the software solution [26]. This lack of a comprehensive software platform hinders the car's potential as a viable testing environment for new control algorithms. Hence, our focus was drawn towards this challenge, prompting the development of a clearly defined problem statement.

> In the Hybrid Systems Lab at UCSC, a significant challenge faced by graduate students and professors is the absence of an effective software test platform autonomous vehicles. Their extensive research mandates the availability of a scalable and modular software stack that can operate efficiently on their full-scale vehicle. Addressing this issue, we suggest a decoupled framework consisting of perception, path-planning, and motion modules that can be used on any platform. To validate a baseline solution, our plan is to integrate this framework into a small-scale RC car. By doing so, we aim to demonstrate that our software has the capacity to control and guide the car around a small scale track.

To solve this problem our team developed the software platform, Cruize AI. Cruize AI alleviates current issues in self-driving car software research by creating a software framework with all necessary requirements to serve as a full self-driving car software stack. The stack consists of three modules: perception, planning, and motion. Each module has submodules that provide different functions. For example, perception is a module with a stereo camera depth perception submodule, a lane detection submodule, and an object detection

submodule2. Thanks to the use of ROS and the publisher/subscriber paradigm, these modules and submodules are decoupled and can easily be modified or rewritten by researchers wishing to test a new algorithm for a different section of the pipeline [24]. This leads to an interoperable software stack facilitating research. These properties allow Cruize AI to serve as a full software stack for the UCSC hybrid systems lab self-driving car while allowing for easy testing of future algorithms.

In the rest of this paper, we will review our work and the creation of Cruize AI. We will begin by providing a survey of the history of self-driving car research, beginning with its inception in the 1920s and continuing into the present. This will provide the context of where our project fits in the history of self-driving cars. Following that, we will review the technical details of our project and software stack while providing the context of how the system comes together as a whole. This will be done by dividing the software into three major parts: perception, planning, and motion. We will also go over a small-scale car created for validating this software stack and the extensive testing done to ensure each part of our software stack is viable for use in a self-driving car. In closing, we will provide a conclusion to the project to summarize our work and provide advice for future researchers. Finally, we will also summarize the challenges we faced in completing this project and everything our team learned along the way.

1.1 Prerequisite Knowledge

To get the most out of this report and to fully understand the contents of this project it is important for the reader to have the following prerequisite knowledge.

1. Familiarity with ROS (Robot Operating System)
2. Basic knowledge of deep neural networks and commonly used metrics
3. Knowledge of Feedback control and modeling dynamics
4. A strong general understanding of Python, algorithms, and analysis of algorithms
5. An understanding of the pinhole camera model in computer vision

It is infeasible for this report to introduce the reader to all of this technical knowledge for the first time. Each of these topics could, and do, occupy full textbooks. Our report assumes familiarity with these topics. If the reader is lacking knowledge in one of these areas, we recommend reviewing courses on these topics and reviewing our cited sources.

2 Background

In the early days, self-driving cars were a far-fetched concept, however they quickly captured the publics imagination leading to early attempts to create them. As early as the 1920s the first attempts at building a self-driving car began. In 1925 Houdina Radio Control created the first drive by wire car while marketing it as a driverless car. While this car did not have any intelligence of its own and it relied on a human operator, it still marked significant

milestone in the creation of a self-driving car [15]. Work continued on how to achieve a car that could pilot itself. A popular approach to this problem between the 1930s and 1970s was the use of guiding cables embedded into the road which the car could sense and follow. These would provide the car with a path to follow, but they did not provide the flexibility of the modern car instead forcing the car to follow set paths.

Between the 1980s and 2000, new approaches to self-driving cars based on artificial intelligence began to emerge. Early computer vision research and the creation of LIDAR around this time all contributed to the viability of a car that senses its environment and dynamically responds to it. One example of such a car was Carnegie Mellon University's Navlab project. Navlab was able to traverse 2850 across the US while driving autonomously for 98% of the time [29]. Also in this decade, the use of neural networks for self driving cars emerged in the form of ALVINN (Autonomous Land Vehicle in a Neural Network). This car used a two layer neural network to map a camera feed directly to motor outputs. While ALVINN was extremely limited in capabilities only being able to drive at less than 20 miles per hour, its novel use of neural networks was an important milestone in the creation of self driving cars [22].

Advances in self-driving technology continued into the 21st century at increasing rates thanks to greater computational power and the advances in artificial intelligence and the use of deep neural networks. The development of new sensors such as IMUs and GPS allowed for self-driving cars to localize themselves and their position. The increased viability of self-driving cars lead to a series of challenges such as the Defense Advanced Research Projects Agency (DARPA) Grand Challenges [26]. In these challenges teams competed autonomously complete a driving course over 100 miles long [1]. Another challenge was VisLab Intercontinental Autonomous Challenge (VIAC), in which a self driving car was able to drove about 10,000 miles from Italy to China [10].

At present, self-driving car research is steadily advancing, and commercially viable, fully autonomous vehicles appear within reach [26]. While today's self-driving cars demonstrate considerable capability in optimal conditions, but further research is necessary before self-driving cars become ubiquitous. Leading companies such as Waymo, Tesla, and Cruise are making investments towards the realization of this technology. However, these initiatives grapple with several challenges including handling complex real-world scenarios, dealing with regulatory constraints, technological limitations, and addressing safety concerns. The need for more research platforms and rigorous testing procedures is becoming increasingly evident.

3 System Requirements and Methodolgy

We will begin by examining the methodology of our project at the system level before investigating each subsystem in subsequent chapters. First, we formulated the key requirements our system must achieve in the form of a success criteria matrix. That matrix can be seen in figure 1. The success criteria matrix is organized into sub-matrices, each to be validated on a different platform. For example, our scalable success criteria is listed under the full-scale car sub-matrix since we test this criterion on the full-scale car.

	A	B	C	D	E	F	G	H
1	**Simulation Success Criteria**							
2	Criteria	Indicator	Type	Units	Baseline	Result / Objective	Measurement Strategy	Results
3	(Item/need to be addres	(How will you know?)	(Qualitative/Quantitative)	()	(current situation	(target outcome)	(How will you track the change?)	Actual Outcome of the Project
4	Algorithmic complexity	The simulated car will be able to plan a path with a low enough run time to work in real time	Quantitative	Big O analysis	Popular path planning algorithms like A* run in O(n^b) time where b is the average number of branches	We want a more efficient algorithm than A* with O(n^2) or less algorithmic complexity	We will perform algorithmic analysis on our path planning code. We will verify our results by consulting with the CSE 102 Professor, Professor Fremont	We chose RRT as our path planning algorithm in O(nlog(n)) time
5	Simulate Path-Planning	Ability to move from point A to point B in a simulated environment	Quantitative	% success rate for navigating to target location	Basic framework implemented in Gazebo for TurtleBot models but nothing specific to our car	Adapting Gazebo framework to simulate our car and achieve a 90% success rate in navigating in our pre-mapped environment	Run our path-planning algorithm wi feedback control for motion in Gaze and track how many times it successfully reaches the destinatio	Car is able to reach desired position 100% of the time.
6	Efficient Routing	Distance of the car way from planned path. We will measure this using the Gazebo error measurement tool	Quantitative	feet	Currently there is not path following algorithm implemented	Path following algorithm keeps the turtlebot within 1 foot of the planned path 90% of the time	Use the Gazebo error measurement tool to determine the distance of the car from the path throughout the run	We found the path to followed within 1 foot of the path 96% of the time across 50 trial runs
7								
8	**Small Scale Car Success Criteria**							
9	Criteria	Indicator	Type	Units	Baseline	Result / Objective	Measurement Strategy	Results
10	(Item/need to be addres	(How will you know?)	(Qualitative/Quantitative)	()	(current situation	(target outcome)	(How will you track the change?)	Actual Outcome of the Project
11	Reliability	Number of laps around the track without more than one wheel crossing over the lanes at one time	Quantitative	Number of Laps	Current self driving cars frequently fail to find lanes with errors approximatley once every minute	Run software stack demonstrating the small scale car can average 10 laps around the track without more than one wheel crossing the lanes	Run 25 trials of the car travelling around the track until failure. Record reason of failure and number of laps made prior to failure	The car averages 11.2 laps before failure. The most common point of failure is a wheel falling off of the small scale car
12	Object Detection	Ability to identify obstacles, road signs / indicators, and hazards	Quantitative	% obstacles and street signs identified in test environment	Effectiveness of deep neural networks vary depending on what training data is used.	Should be able to identify and correctly classify stop signs, pedestrians, and other vehicles with a 90%+ accuracy.	Train deep neural networks with different data sets and see how well objects on the small scale track are detected.	Stop signs and other vehicles were detected with a 93% accuracy found from 30 amount of runs.
13								
14	**Full Scale Car Success Criteria (Stretch Goal)**							
15	Criteria	Indicator	Type	Units	Baseline	Result / Objective	Measurement Strategy	Results
16	(Item/need to be addres	(How will you know?)	(Qualitative/Quantitative)	()	(current situation	(target outcome)	(How will you track the change?)	Actual Outcome of the Project
17	Scalable	Software running on the small scale car can be ported directly to the full scale car without modification	Quantitative	Bool	No scalable small scale to full scale solution	Prove software functioning on the small scale car is easily integratable onto the full scale car without platform specific issues (sensors may need to be recalibrated)	Demonstrate same software stack running on the Jetson Nano and Jetson Orin	Code written on Jetson Nano integrated on Jetson Orin. Exact same software stack from small scale car ran on the full scale car with no errors or issues
18	Integration	The perception system running on the small scale car accutley determining depth, lanes, and identifying pedestrians	Qualitative	Bool	The hybrid systems lab car has no software integrated on it	Show the perception system functions on the full scale car after calibrating the sensors	Video of a small track and the lane detection, depth perception, and object detection working simultaneously on the hardware in real time	Sucessfully ran the perception stack on the full scale car without issue (no crashes)

Figure 1: Team Success Criteria for organizing system level requirements

All of the listed criteria were created in collaboration with and based on the need of our client, Ricardo Sanfeliche.

Once the system's requirements were defined, we began to design our software solution from the top down. A high-level view of our software stack can be viewed in the block diagram of figure 2. As seen in the colored blocks of the diagram, a successful test platform needs these 5 systems: Sensors, Perception, Navigation, Motion, and Framework. Our sensors component simply represents the inputs from the vehicle's onboard sensors. On the full-scale car, this block would be provided by our sister team, the Self-Driving Car Sensor team.

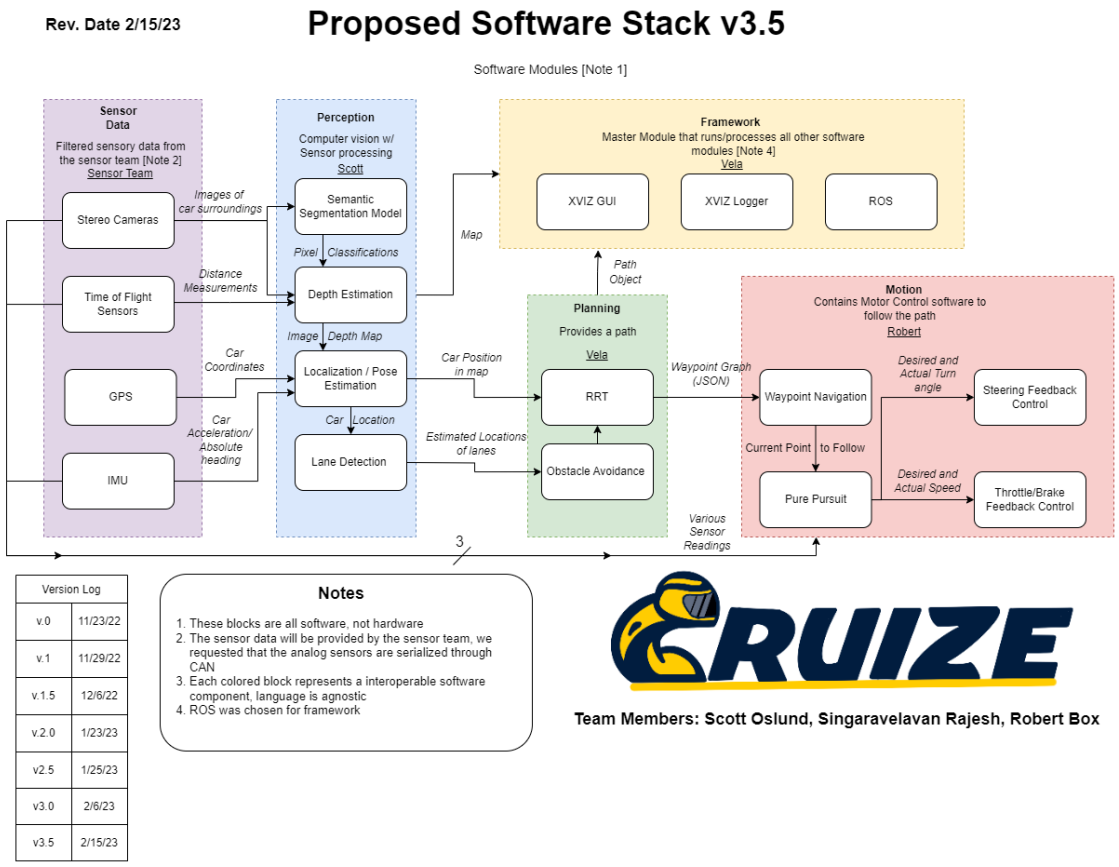

Figure 2: Self-driving software stack divided into 5 independent modules

A description of each subsystem shown in the block diagram is shown below.

- Sensor Data (purple) - This module simply represents the sensor data. This section of the diagram is not created or designed by us, it is provided by the Self Driving Car Sensor team.
- Perception (blue) - The perception module intakes the sensor data. It filters it, makes inferences using deep neural networks (DNNs), and outputs information about the world around the car.
- Path Planning (green) - This module uses the perception data and the knowledge of the world around the car to plan out a path for the car to take through the known environment. This is done using a path-planning algorithm that accounts for obstacles in the road and street laws. The output is a series of waypoints for the car to follow.
- Framework (yellow) - The Framework module provides the method for interconnecting each module and visualizing the car's thinking.
- Motion (red) - The motion module executes the desired path of the car. This is done using a path-following algorithm using the waypoints provided by path planning. In addition, PID control is used to ensure the car correctly follows the desired path.

We successfully incorporated all five modules in simulation and on a small-scale car, providing us with the opportunity to test our software stack in real-time. The software demonstrated proficiency in reacting to various road conditions and objects captured by the car's camera. The embedded decision-making process enabled the car to follow road lanes, halt at stop signs, or avoid obstacles as needed. The inter-module communication was facilitated seamlessly via the ROS framework, underlining its pivotal role in our system. Significantly, we were able to transfer the entire software stack into a different environment on the full-scale car, thereby demonstrating the software stack's robust interoperability.

In the following sections, we will examine how each subsystem is decomposed and the technical details of how these systems achieve their goals. We will then examine how the software was integrated together in simulation, on the small-scale car, and on the full-scale car. Throughout, we will also include our validation data and testing results showing us to have met our success criteria.

3.1 Perception

The perception module is responsible for intaking sensor data and converting it to a map of the world around the car. This system receives the raw data the sensor module outputs. It then filters and interprets this data to output key features that will be used by the Navigation and Motion systems. To accomplish this, this module heavily makes use of computer vision and machine learning.

Our perception module was separated into a few major subsystems allowing them to be flexible and interoperable. This was done by using ROS to write a publisher module for each sensor input. This design choice was made to allow the perception module to be easily alterable depending on the sensors available to the user. By default, our perception module features object detection, lane finding, and depth perception via Time of Flight sensors and Stereo cameras. The design of each of these subsystems will be discussed in the following sections. More subsystems can be created and added or removed as necessary depending on sensor availability.

3.1.1 Object Detection

A mandatory part of perception is the ability to locate and classify objects around the car using computer vision. This subsystem relies on the input camera feed. Each individual frame of the feed is sent to this subsystem. The subsystem then uses the frame and an object detection machine learning module to find and classify objects in the frame. The subsystem outputs a python dictionary of the found objects classification and the bounding boxes of where the objects lie in the image.

Our object detection algorithm is centered around the YOLO-SM architecture [27]. The YOLO object detection algorithm is a popular and efficient approach for real-time object detection in computer vision tasks. The YOLO-SM architecture, is a variant of YOLO that incorporates spatial pyramid pooling to capture multi-scale information [11, 13, 23]. At its core, the YOLO-SM algorithm divides the input image into a grid of cells. Each cell is responsible for predicting bounding boxes and class probabilities for the objects present within its boundaries. Unlike other object detection algorithms that perform region proposals

and subsequent classification, YOLO-SM performs these tasks simultaneously, resulting in faster inference times [27].

The YOLO-SM architecture consists of several key components. The initial part of the network is a deep convolutional neural network (CNN) that processes the input image and extracts high-level features [27, 11, 13, 23]. In our case, the CNN is pre-trained and we import the model from PyTorch [21]. The output of the CNN is then fed into the detection layer, which predicts bounding boxes and class probabilities. This layer consists of a set of convolutional and fully connected layers that generate a fixed number of bounding boxes for each grid cell. For each bounding box, the algorithm predicts the coordinates (x, y, width, height), confidence scores indicating the presence of an object, and class probabilities for different predefined object categories.

To handle objects at different scales, YOLO-SM incorporates spatial pyramid pooling. This pooling mechanism allows the algorithm to capture features at multiple scales by applying pooling operations on different subregions of the CNN's feature map [13]. This enables the network to effectively detect objects of varying sizes. During training, YOLO-SM uses a loss function that combines localization loss (based on bounding box predictions) and classification loss (based on class probabilities) [11]. This loss function guides the network to improve its object detection performance by adjusting the weights of the network through backpropagation [27, 21, 23]. A visualization of the YOLO-SM architecture is shown in figure 3.

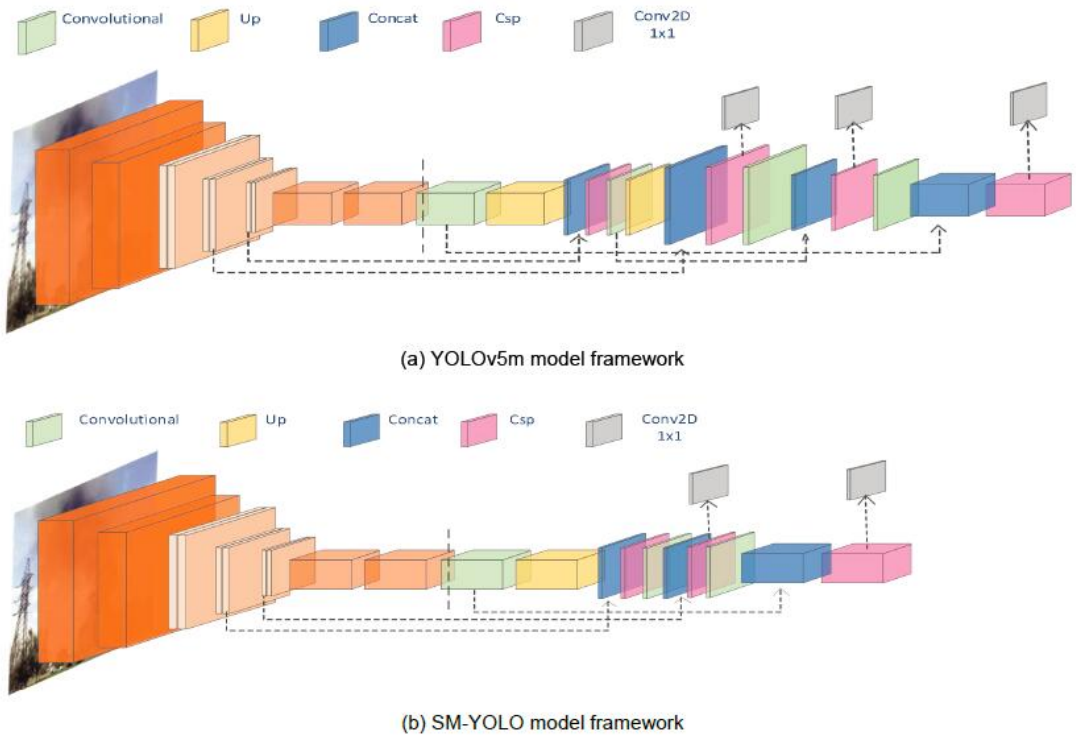

Figure 3: Comparison of the traditional YOLO architecture to YOLO-sm

Overall, we chose the YOLO-SM architecture because it offers a fast and accurate solution for object detection tasks, making it suitable for real-time applications such as autonomous

driving. Its ability to process input images in a single pass and produce bounding box predictions directly makes it highly efficient for use in perception subsystems like the one described, where objects around the car need to be located and classified in real time. An example of the model's output can be seen in figure 4.

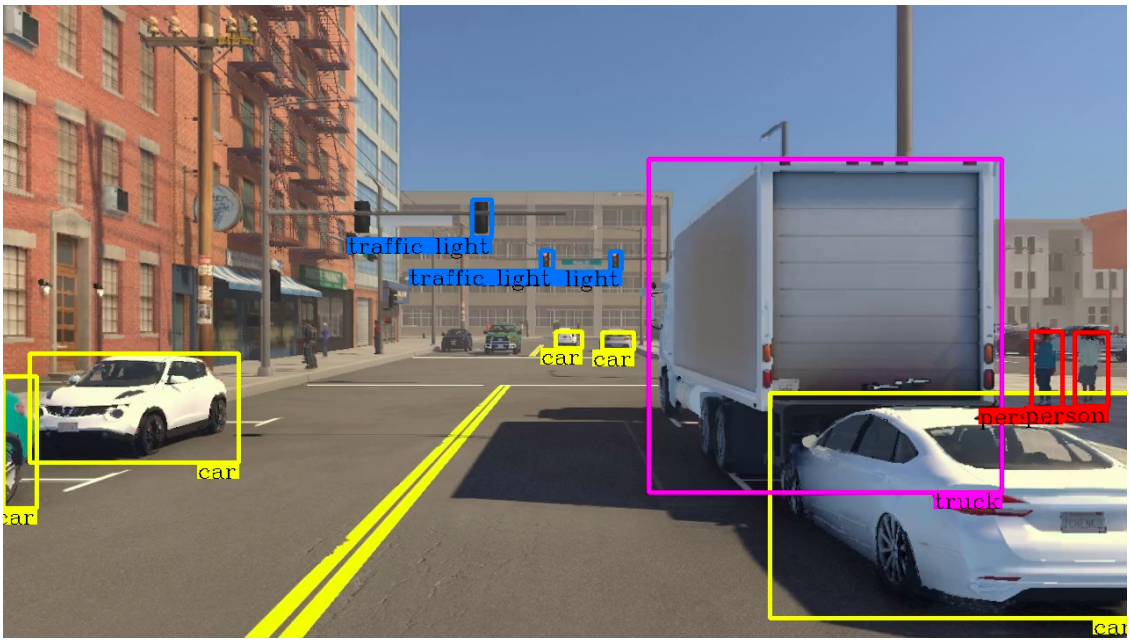

Figure 4: A visualization of the output of the YOLO-SM classifier used for this project

We implemented our YOLO-SM object detection using pytorch [21] and a pre-trained version of the model. This allowed us to get a version of the object detection model working correctly while having known statistics about the model. However, in addition to using the model we also had to apply filtering to the images. Prior to inputting images into the model, we used pytorch to apply a gaussian blur to the images which filters out high-frequency noise [13]. Using nearest-neighbor interpolation, we scaled down the image to the 240x240 image size required by the YOLO model. To validate our model ran it on the Pascal VOC dataset and calculated the IoU (Intersection over Union) score. The IoU score measures the overlap between the predicted bounding box or segmentation mask and the ground truth, by calculating the ratio of the intersection area to the union area between the two. The IoU score is a popular metric in object detection models since it captures the performance of the model while punishing both false negatives and false positives.

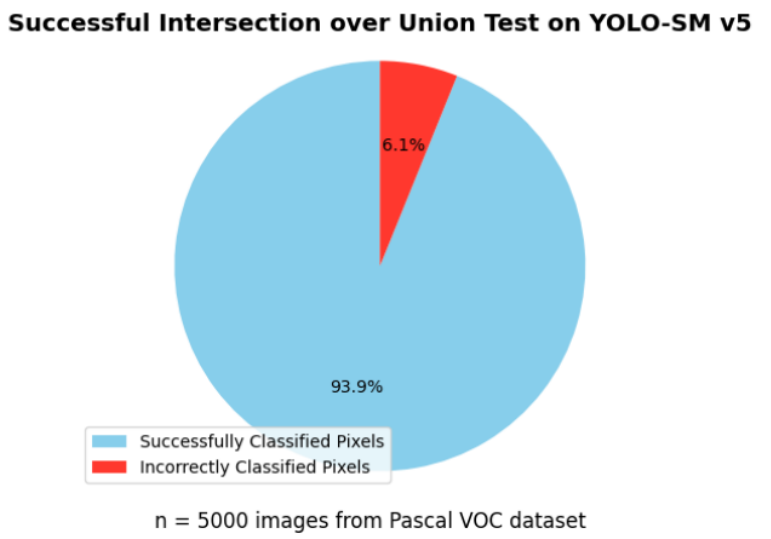

Figure 5: Successful Results of the IoU test on the Pascal VOC dataset

As seen in figure 5, our object detection has a high success rate and meets our success criteria matrix standards. Every different possible classification of the YOLO classifier was tested in these tests.

3.1.2 Lane Detection

Lane detection is a key component of autonomous driving. An autonomous vehicle must identify and follow the position of lanes on the road to ensure it remains within its designated lane. We were able to implement a lane detection system on a small scale car using a Logitech camera and computer vision code, powered by the Jetson Nano [6]. Our validation process involved driving the car on a small scale track and checking for successful lane detection and adherence. We also conducted preliminary tests on an actual car, demonstrating that our software stack is transferable from a Jetson Nano to a more advanced Jetson Orin platform [5]. Due to the limited space on the small scale car for additional sensors, we utilized lane detection for closed-loop steering control as well.

The small scale car was equipped with a Logitech C920 HD camera positioned on the top layer of the vehicle. This camera, which was connected to the Jetson Nano via USB, faced slightly downwards to capture the lanes on the road. The camera is capable of recording high-definition footage at 30 frames per second. However, in order to conserve processing resources for other software modules, we reduced the camera's frame rate to 10 frames per second. This adjustment was suitable for our tests, as our small scale car had a very low RPM motor and the requirement for rapid reaction time was minimal.

Figure 6: Raw image feed from Logitech camera

Upon receiving a frame from the camera, the lane detection algorithm applied an HSV (Hue, Saturation, Value) filter to the image [13, 20]. This filter allowed us to convert colors from the traditional RGB color space to HSV, which is more intuitive for color representation and isolation. The lanes on our small-scale track were marked with blue masking tape, so we set lower and upper bounds in the HSV color space specifically for various shades of blue [20]. Any pixel within the set blue range was masked to white, while all other pixels were masked to black. By carefully selecting these color boundaries, we achieved a binary image output where the lanes appeared in white against a predominantly black background. This enabled more efficient subsequent edge detection and line fitting stages in our lane detection algorithm.

Figure 7: Blue sections of the image are marked as white

Following the conversion of our image into a binary representation, the next step in the lane detection process is to identify the borders between white and black areas. These borders, delineated by thin white lines, represent the edges of the lanes that our car is programmed to follow. This edge detection is accomplished using the 'Canny()' function from the openCV library [20]. The Canny function initiates this process by applying a Gaussian blur to the image. This blurring technique is used to reduce image noise and detail, helping to smooth out any individual white pixels that might otherwise disrupt the edge detection process [32]. It ensures that only large sections of white pixels remain visible. Subsequently, the function calculates the gradient between adjacent pixels [20]. The gradient is a measure of the change in intensity between pixels, which helps us identify where significant changes occur in the image. The orientation of these gradients also provides valuable information about the direction of the edges. Strong gradient differences are then marked as lane edges and highlighted with white lines. This results in an image where the lane borders are clearly marked.

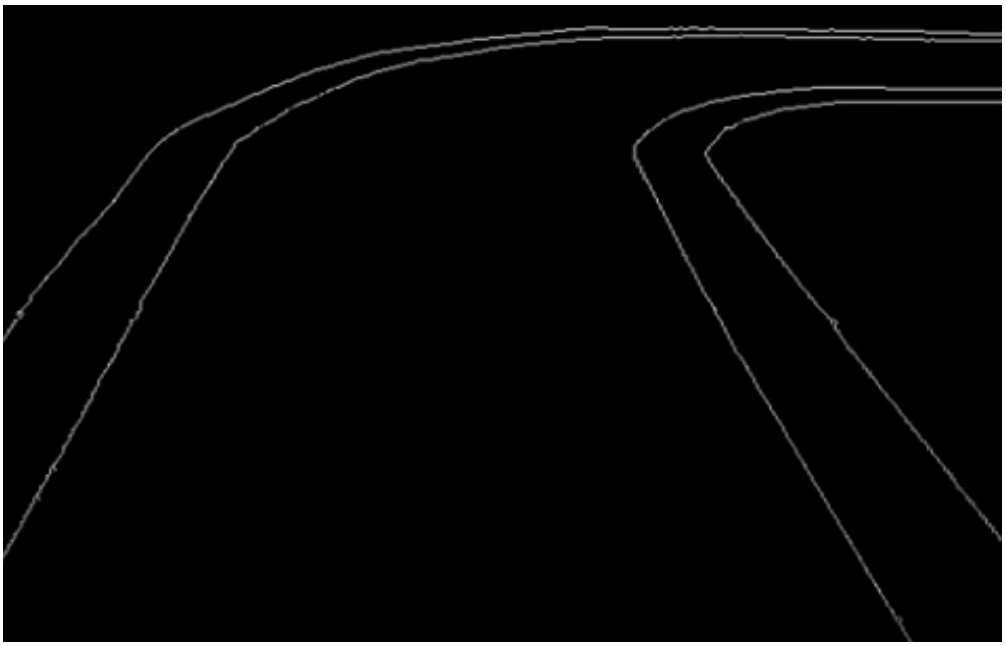

Figure 8: Edges between white and black areas are marked white

In the processed image, we can typically observe four "lane" markers. These are essentially the edges of the actual left and right lanes as seen by the camera. The program needs to determine which two of these four markers represent the path for the autonomous vehicle to follow. Sometimes, not all four lane markers are visible, necessitating a logical framework to correctly identify the relevant lane boundaries [20]. The image is bisected into two halves. Each half can contain a maximum of one lane. The identified edges from the image are then linearized to form straight lines [32]. This step is critical as it simplifies the path for the vehicle to follow by converting the potentially irregular edges of the lanes into regular straight lines [13]. Occasionally, multiple horizontal lines might be detected within one half of the image. In such cases, these multiple lines are averaged to form a single line that is considered to be the new lane to follow. This process reduces the complexity and

increases the robustness of the lane detection module. After this transformation, ideally, we end up with an image that displays two distinct straight lines. These lines are overlaid on the original image, indicating the paths for the autonomous vehicle to follow.

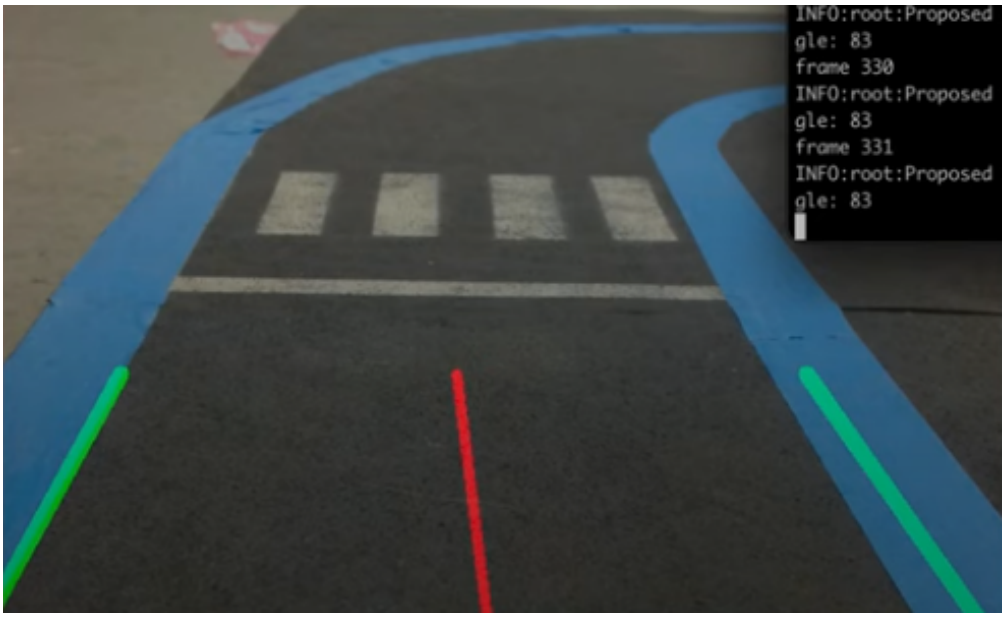

Figure 9: Lanes (Green) are used to calculate the steering angle (red)

The position and direction of the two lanes in the image has been identified, providing the necessary input to generate a steering angle that will keep the vehicle centered between these lanes. If only one lane is detected in the image, the system will create a steering angle designed to keep the vehicle parallel to that single lane, maintaining a predefined offset. This strategy ensures that the vehicle will continue to navigate effectively until the second lane becomes visible to the system. When both lanes are visible, the algorithm calculates an offset value that represents the vehicle's deviation from the midpoint between the two lanes. It then generates a steering angle intended to correct this offset and brings the vehicle back towards the center of the lane. This steering angle, is transmitted to the Arduino board. The Arduino then controls the servo motors that adjust the front wheel angle in accordance with the computed steering command.

This strategy ensures that the vehicle remains well-aligned with the center of its lane, continuously adjusting to deviations. In this way, the autonomous vehicle can effectively navigate the environment.

3.1.3 Depth Perception

In addition to detecting objects in the environment, the car also required a method for estimating their position relative to itself. This is a key aspect of localization and creating a model of the world surrounding the car [30]. For this reason, we implemented two methods of depth perception in our software framework.

3.1.3.1 Time of Flight

one method of depth perception we incorporated into our design is using the use of time of flight sensors. In the case of our small scale car, we used the Vl61080x. The Vl61080x time of flight (ToF) sensor is an advanced optical sensing device that accurately measures distances using the time of flight principle. It incorporates an emitter and a receiver module to send out and capture the time it takes for light to reflect off of a target object. The Vl61080x sensor uses near-infrared (NIR) light. This light is used because it is less affected by ambient light sources and atmospheric conditions [9]. The emitter component of the Vl61080x sensor consists of a near-infrared LED. It emits short pulses of NIR light toward the target object.

Figure 10: Sample image of chosen time of flight sensor

On the receiving end, the Vl61080x sensor uses a simple receiver module containing a photodiode array [9]. This array captures the reflected light from the target object. By detecting the intensity and phase shift of the received light, the sensor can accurately determine the distance between itself and the target. To decrease the impact of ambient light sources and improve the signal-to-noise ratio, the Vl61080x sensor uses modulation techniques. The emitter emits modulated light signals, in the form of a square wave, at a known frequency. By synchronizing with this modulation frequency, the receiver can differentiate the emitted light from ambient light sources, ensuring precise distance measurements [9].

These calculations allow the Vl61080x sensor to provide precise distance information [9]. Further discussion of the interface low-level software interface for extracting data from this sensor is left until the small-scale car discussion.

While the full-scale car uses a different time of flight sensor, the VL5380x sensor, it functions in the same way as the Vl61080x sensor on the small-scale car. The only difference between these components is the VL5380x sensor is optimized for longer-range sensing, more appropriate for the full-scale car. Our software used for interfacing with the Vl61080x sensor

works with the VL5380x sensor too.

3.1.3.2 Stereo Vision

While the time of flight sensor is accurate and easy to implement, it can only function in one predetermined area which led to the requirement for a second depth perception method with a broader range. This came in the form of stereo-vision cameras. Stereo vision works by using two different cameras in different, known locations, to triangulate the distance of objects away from the camera. Since this method uses cameras, it is capable of determining the depth of any object within the field of view of both cameras [30].

The first required step in implementing stereo vision is obtaining two cameras with known camera parameters and known positions. For this, we used the HD 1.3mp 960P Double Lens USB Webcamera Module from Amazon chosen and sourced by the self-driving car sensor team. Using the given camera parameters on the datasheet, we were able to implement depth perception. However, our method could be used on arbitrary stereo cameras.

With the cameras set up, and the camera feed running, the first step in our stereo-vision algorithm was computing the disparity between pixels of the same objects from one frame to another to create a disparity map. This task is not trivial, it requires stereo-matching [13], the process of mapping one pixel to the corresponding pixel in the second image. Let $D(x, y)$ represent the disparity map, which assigns a disparity value to each pixel coordinate (x, y). We formulate the disparity estimation process as an optimization problem, where our goal is to minimize some cost function C(x,y,d) [30]. With this representation, our disparity map is constructed using the following formulation.

$$D(x, y) = \arg\min_{d} C(x, y, d) \tag{1}$$

We find the disparity d that minimized our cost function and assign that to our disparity map.

For this formulation to be successful, we need a cost function that can accurately determine the correspondence between pixels. If the pixel represented by coordinates (x, y) corresponds to the pixel $(x + 5, y)$ [1] in the second image, the cost function should output a low value when the disparity d is equal to 5 and a higher value as the disparity is further from 5 [23]. We chose to use the sum of squared differences algorithm for our cost function. This cost function measures the dissimilarity between the intensities of corresponding pixels in the left and right images for a given disparity.

The Sum of Squared Differences (SSD) algorithm is mathematically formulated as follows [13]. Let $I_L(x, y)$ and $I_R(x - d, y)$ represent the pixel intensities of the left and right images, respectively, at pixel coordinates (x, y). The disparity d represents the horizontal offset between corresponding points in the left and right images. The SSD cost function $C_{\text{SSD}}(x, y, d)$ can be calculated as the sum of squared differences between the intensities of corresponding pixels:

$$C_{\text{SSD}}(x, y, d) = \sum_{i} \sum_{j} \left(I_L(x + i, y + j) - I_R(x + d + i, y + j)\right)^2 \tag{2}$$

[1] Note that we only consider the disparity on the x-axis since our stereo cameras are horizontally aligned to be on the same plane. This implies that the disparity will be entirely along the x-axis

Where i and j represent the local window coordinates around the pixel (x, y) within a specific matching window. The window size and search range for the disparity is manually tuned and this depends on the chosen stereo camera hardware. Using this cost function, a disparity map can be successfully generated.

Once the disparity map $D(x, y)$ has been created, the final step is to compute the estimated depth and to convert this into a depth map. Let the depth map be represented as $Z(x, y)$, let B represent the separation between the two cameras, and f represent the focal length [23]. Then our depth map can be computed as

$$Z(x, y) = \frac{B \cdot f}{D(x, y)} \tag{3}$$

We see from this formula that intuitively, as the disparity increases, the depth decreases. A larger disparity indicates that the corresponding scene point is closer to the cameras. Conversely, a smaller disparity corresponds to a scene point that is farther away. By calculating the depth map using this formula, we can assign depth values to each pixel in the image pair, providing an estimate of the distance of objects from the cameras [13].

3.2 Planning

3.2.1 Path Planning

To implement path planning we chose the well-known RRT (Rapidly Exploring Random Trees) algorithm for continuous space environments [18, 28]. RRT* is a very popular path finding algorithm that is used to get the a path without collisions with objects in complex environments.

The basic concept is to construct a tree data structure by randomly exploring the environment space surrounding the car. The tree then grows iteratively through the addition of new nodes [18]. This random exploration allows us to quickly cover the unsorted space and quickly find a path towards the desired endpoint.

We implemented the RRT algorithm using an recursive approach. We use a random sampling strategy to generate nodes and points in the space we cover [28]. The randomness allows for us to explore a lot of the environment and find paths through narrow ways quicker. We also implemented a Cost Function to associate a cost value with an edge/node to represent various factors like distance, collisions, and straightness [18]. We minimize this variable to find the paths that will satisfy our criteria.

Algorithm 1 RRT Algorithm

Initialize tree Tq_{init}
for $i = 1$ to N **do**
  Generate random configuration q_{rand}
  Find nearest node q_{nearest} in T to q_{rand}
  Extend tree from q_{nearest} towards q_{rand} by a step size δ
  $q_{\text{new}} \leftarrow$ New configuration after extension
  if No collision with obstacles **then**
    Add q_{new} to T and connect it to q_{nearest}
  end if
end for
return Tree T

3.3 Motion

We developed a vehicle control system for the actual car that involves two critical aspects: velocity control and steering control. Each aspect is crucial to achieving a well-rounded autonomous vehicle system that operates efficiently and safely under a wide range of conditions.

In the 'Velocity Control' subsection, we delve into the mathematics and physics that govern the longitudinal velocity of our autonomous vehicle. We consider the forces acting on the car, including the thrust provided by the motor, wind resistance, and rolling resistance. By closely following the methods and principles outlined in this research paper [19], we will derive an equation for the acceleration of the vehicle. The calculation takes into account various factors such as air density, drag coefficient, and gear ratios. This model will ultimately enable us to design a PID controller that adjusts the car's speed in response to real-time conditions.

The 'Steering Control' subsection explores the derivation of motion equations for a four-wheeled vehicle, in order to create a transfer function for feedback control. We assume a two-wheeled bicycle model as an accurate representation for our four-wheeled vehicle. The dynamics of steering involve several forces in the lateral direction, such as the steering of the front wheels, slip angle of the tires, and inertia around the center of gravity (CG). We will cover the phenomenon of slip angles, their role in under-steered and over-steered scenarios, and how they relate to lateral forces on the tires. Finally, we will demonstrate how we can manage the variable transfer function for different velocities by employing an adaptive PID controller.

3.3.1 Longitude Control

To control the longitudinal velocity of the car, only forces acting in the longitudinal direction. Derivation of an acceleration formula for a four-wheeled vehicle was closely followed in this textbook[14].

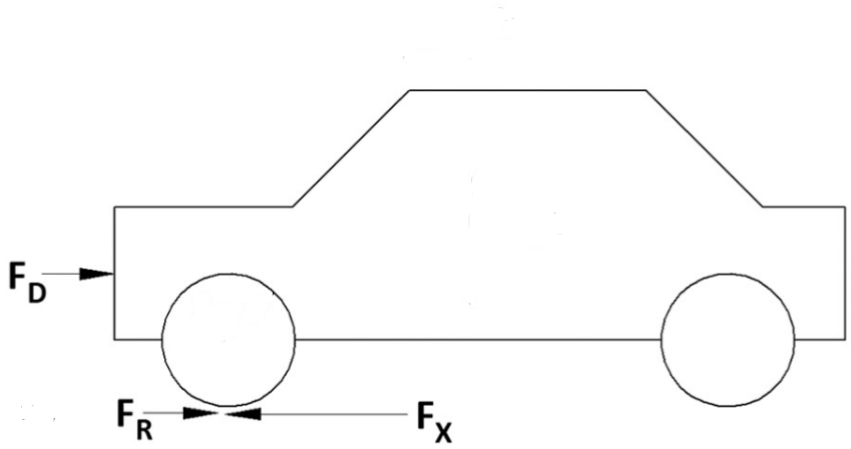

Figure 11: Free Body Diagram of longitudinal forces acting on vehicle

F_D = Force due to wind resistance
F_R = Force due to rolling resistance
F_X = Force due to motor spinning wheels

A simplified diagram can be made with only three forces acting on the car, thrust of the motor moving the car forward, and wind/rolling resistance slowing the car down. Using Newton's second law of motion we can create an equation containing all forces acting on the car and solve for acceleration:

$$ma_x = F_X - F_D - F_R \tag{4}$$

The equation for wind resistance is:

$$F_D = \frac{pV^2 C_d A}{2} \tag{5}$$

p = mass density of air
V = Longitudinal velocity of the car
C_d = Air drag coefficient
A = Area of the orthogonal projection of cars front side

The value for mass density of air was calculated by finding the average temperature, dew point, and air pressure of Santa Cruz, California. The average temperature in Santa Cruz is 13.2°C[8]. The average dew point is 3.3°C, and the average air pressure is 1015 hPa[2]. Using an online calculator, a good estimate for air density in Santa Cruz is $\rho = 1.2754 \frac{kg}{m^3}$. Area of the front side of the car was estimated by taking width and height measurements of the actual car. The orthogonal projection of the cars front side is estimated to be $2.1 m^2$. Air drag coefficient is determined by how aerodynamic the shape of the car is. The drag coefficient for the GEM E2 is not known, however, cars with similar shapes have a drag

coefficient of around $C_d = 0.3$. Resulting in our wind resistance equation only changing with vehicle longitudinal velocity.

The equation for rolling resistance can also be looked up quite easily.

$$F_R = f_r mg \tag{6}$$

f_r = rolling resistance coefficient

The rolling resistance coefficient of our specific tires is not know. A good estimate can be made based on the SAE J2452_201707 standard that measures the rolling resistance coefficient of every commercial tire[4]. The rolling resistance coefficient range under this standard is 0.007 to 0.014. The mass of the vehicle can be looked up in the GEM E2 user manual and it is 544 kg[3].

To calculate the force at the wheels caused by the engine we have to find gear ratios in the drive train that multiply torque at the wheels. Any spinning components of the vehicle add equivalent mass to the total mass of the vehicle due to their inertia. The car manual provides a picture of the transaxle. Due to being unable to take the transaxle apart to measure the size of every spinning component and the gear ratios. The schematic was used as an estimate for both.

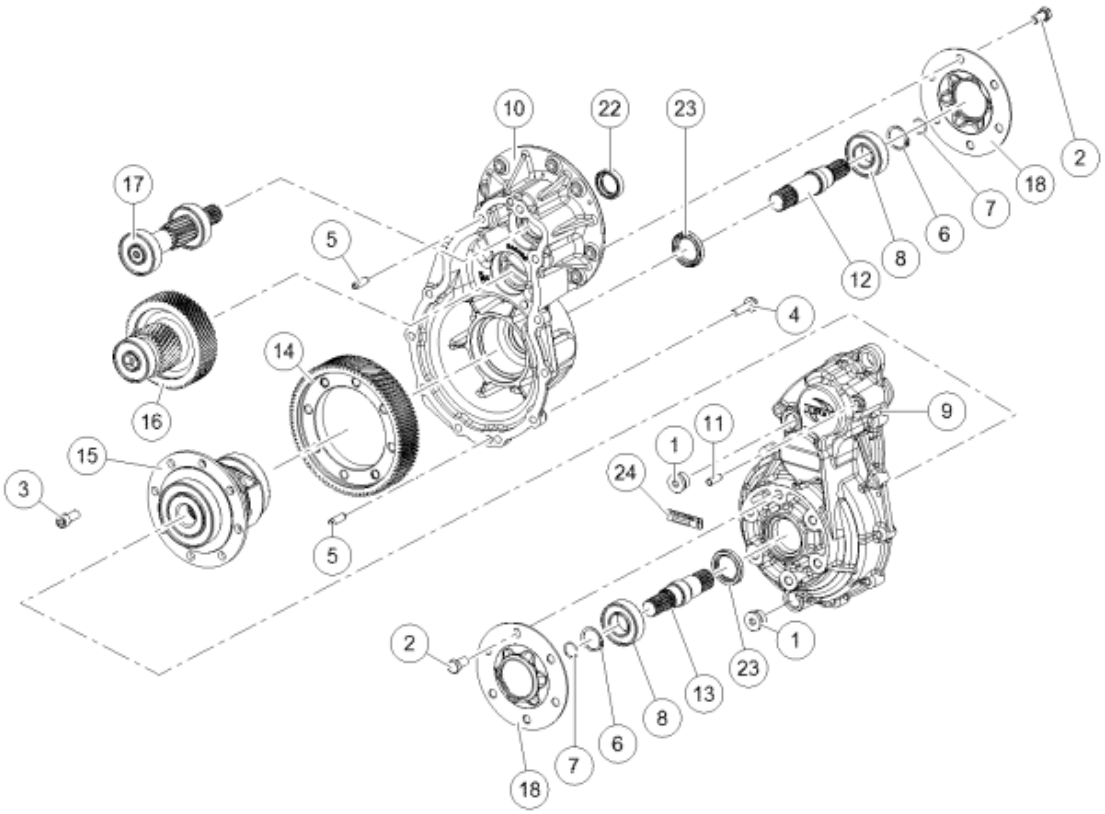

Figure 12: Transaxle assembly view of GEM E2. Rotating parts are: motor shaft (17), one gear transmission (16), driveshaft (14), differential (15), wheels and front axle shaft (not shown).

Equation for the tractive force from the engine as derived by Gillespie in his textbook:

$$F_X = \frac{T_e N_{tf} \eta_{tf}}{r} - \{(I_e + I_t)N_{tf}^2 + I_d N_f^2 + I_w\}\frac{a_x}{r^2}$$

T_e = Torque produced by engine
N_t = Transmission gear ratio
η = Drivetrain efficiency
r = Tire radius
I_e = Inertia of engine shaft
I_t = Inertia of transmission
N_f = gear ratio of final drive
I_d = Inertia of Driveshaft
I_w = Inertia of wheels

Maximum engine torque T_e can be found by multiplying horse power by 5252 and dividing by motor rpm. The GEM E2 motor has 6.7 horsepower and 4100 rpm(car manual reference), therefore, having a resulting maximum torque $T_e = 11.6Nm$. The total transaxle gear ratio

is documented in the user manual as 17.05:1. The transmission gear ratio N_t and differential gear ratio N_f have to combine to equal 17.05:1. By estimating from the transaxle assembly view, $N_t = 3.1:1$ and $N_f = 5.8:1$. Inertia of all the moving components was estimated by taking measurements of each components and simplifying the geometry of the objects into cylinders. A cylinder has a simple Inertia equation:

$$I = \frac{m * r^2}{2} \tag{7}$$

The mass of each part was found by calculating the volume and multiplying by the density of mild steel ($7.8kg/m^3$). The resulting inertias are $I_d = 0.06\frac{Kg}{m^2}$, $I_w = 0.45\frac{Kg}{m^2}$, $I_e = 0.002\frac{Kg}{m^2}$, $I_t = 0.01\frac{Kg}{m^2}$. The tire radius was measured to be 0.28m and the drive-train efficiency is estimated to be 85%.

Combining equations (1), (2), (3), and (4) and solving for a_x we get an equation for longitudinal acceleration.

$$a_x = \frac{-f_r mg - \frac{\rho V^2 C_D A}{2} + \frac{T_e N_t N_f \eta_{tf}}{r}}{m + \frac{(I_e+I_t)N_t^2 N_f^2 + I_d N_f^2 + I_w}{r^2}} \tag{8}$$

For a PID controller, we need a linear equation, unfortunately, there is a V^2 in the equation due to wind resistance that needs to be linearized. After combining some constants for cleaner linearization the equation looks like this:

$$a_x = \frac{-R}{G} - \frac{A}{G}V^2 + T_e\frac{J}{G} \tag{9}$$

Where the constants are defined as:

$$\begin{aligned} R &= f_r mg \\ A &= \frac{\rho C_D A}{2} \\ J &= \frac{N_t N_f \eta_{tf}}{r} \\ G &= m + \frac{(I_e + I_t)N_t^2 N_f^2 + I_d N_f^2 + I_w}{r^2} \end{aligned} \tag{10}$$

The linearization point for the equation is $V = 2.5m/s$ (9 km/h) the car would travel at low speeds near that linearization point most of the time. V of 2.5 makes engine torque T_e equal to 2. Therefore:

$$\begin{aligned} V_* &= 2.5 \\ T_{e*} &= 2 \end{aligned} \tag{11}$$

After linearizing equation (7) we have:

$$a_x = -2 * \frac{I}{G}V_* V + \frac{J}{G}T_e \tag{12}$$

Simplify constants again:

$$a_x = -KV + LT_e \tag{13}$$

I took the Laplace transform and found the transfer function:

$$\frac{V}{T_e} = \frac{L}{s + K} \tag{14}$$

After computing the constants using parameters from table 1 to fill in the transfer function. Simulink [12] was used as the testing simulation software to find the PID values of the controller.

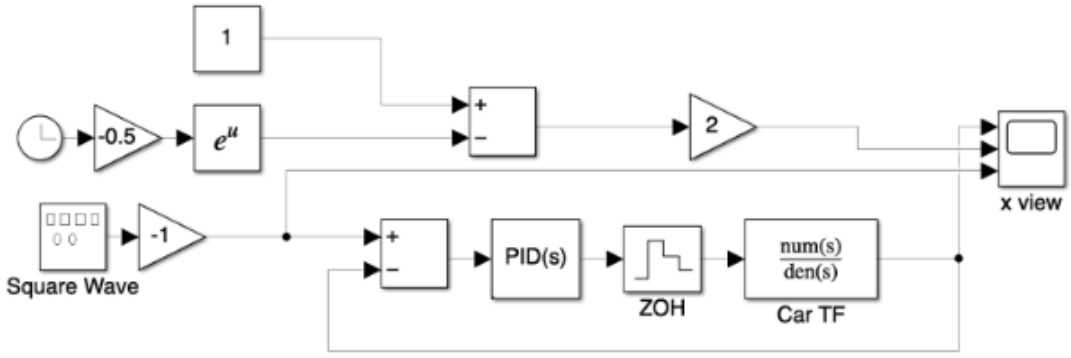

Figure 13: Block diagram of PID controller and ideal response

In the block diagram the PID controller has the values $P = 6, I = 0.2, D = 0.025$. A Zero-order hold as placed between the transfer function and PID to simulate a discrete system with a frequency of 20Hz. The velocity response is displayed along with a time-varying equation:

$$V = 2(1 - e^{-.5t}) \tag{15}$$

If the controller responds too quickly when changing velocity, the ride would be uncomfortable for the user, therefore the controller should respond similarly to equation (13).

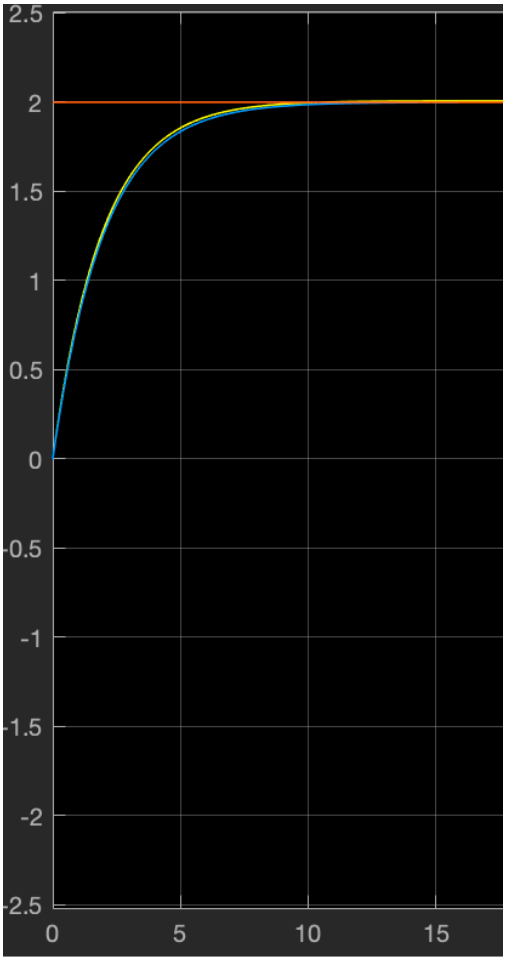

Figure 14: Response of PID controller (Yellow) compared to ideal response (Blue)

Variable	Description	Value (SI)	Source
m	Vehicle Mass	544 kg	Service Manual
r	Tire Radius	0.28 m	Measured
f_r	Rolling resistance coefficient	0.02	Estimated
C_D	Air drag coefficient	0.3	Estimated
A	Frontal Area	2.1 m^2	Calculated
N_t	Transmission gear ratio	3.1:1	Estimated
N_f	Differential gear ratio	5.8:1	Estimated
N_total	Total gear ratio	17.05:1	Service Manual
I_d	Driveshaft inertia	0.06 Kg * m^2	Estimated
I_w	Wheel and axle inertia	0.45 Kg * m^2	Estimated
I_e	Engine inertia	0.002 Kg * m^2	Estimated
I_t	Transmission inertia	0.01 Kg * m^2	Estimated
n_tf	Drivetrain efficiency	85%	Estimated
T_max	Maximum engine torque	11.6 N*m	Calculated
p	mass desnsity of air	1.22 Kg/m^3	Calculated

Figure 15: List of parameters used to calculate velocity control

3.3.2 Steering Control

Closely following the derivation of motion equations for a four wheeled vehicle[33]. Transfer functions for feedback control can be derived after some linearization. Presupposing that a two wheeled bicycle model is an accurate representation of a four wheeled vehicle. We can derive equations of motion for a bicycle and apply them to a car. To start deriving a transfer function for steering, all forces in the lateral direction are found. Steering of the front wheels, angle slip of the tires, and inertia around the center of gravity (CG) all cause a lateral force that need to be incorporated into the transfer function equation.

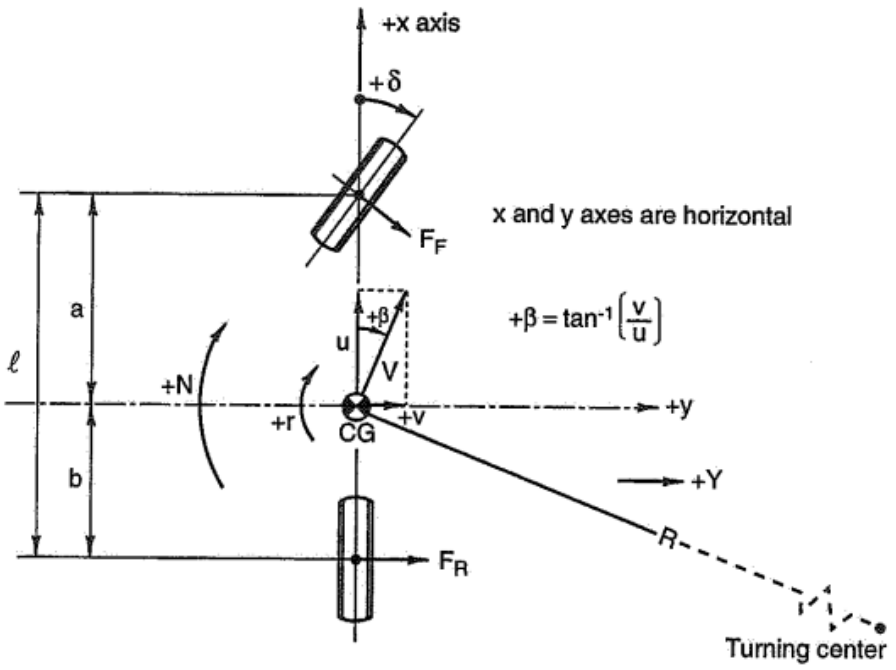

Figure 16: All forces acting on a 2 DOF bicycle model

a = Distance from CG to front axle
b = Distance from CG to back axle
l = Distance from front axle to back axle
r = Yaw velocity
δ = Steer angle
u = Longitudinal velocity
v = Lateral velocity
V = Car velocity vector
F_F = Lateral force at front wheel
F_R = Lateral force at rear wheel
R = Turning Radius
N = Yaw moment

Slip angles are a force from the rubber tires contact patch moving in a different direction from where the tires are oriented. Front and rear tires will have different slip angles due to only the front tires rotating when making a turn. Slip angles are caused by the elasticity of the rubber and "rolling over" on itself. Equation for slip angles can be approximated using distance from the CG, and angle of the tire. The direction of slip angles depends on if the car is under-steered of over-steered. Since the GEM E2 is front wheeled drive, we can assume the car has under-steer and, therefore, the slip angle goes in the opposite direction of the tire angle.

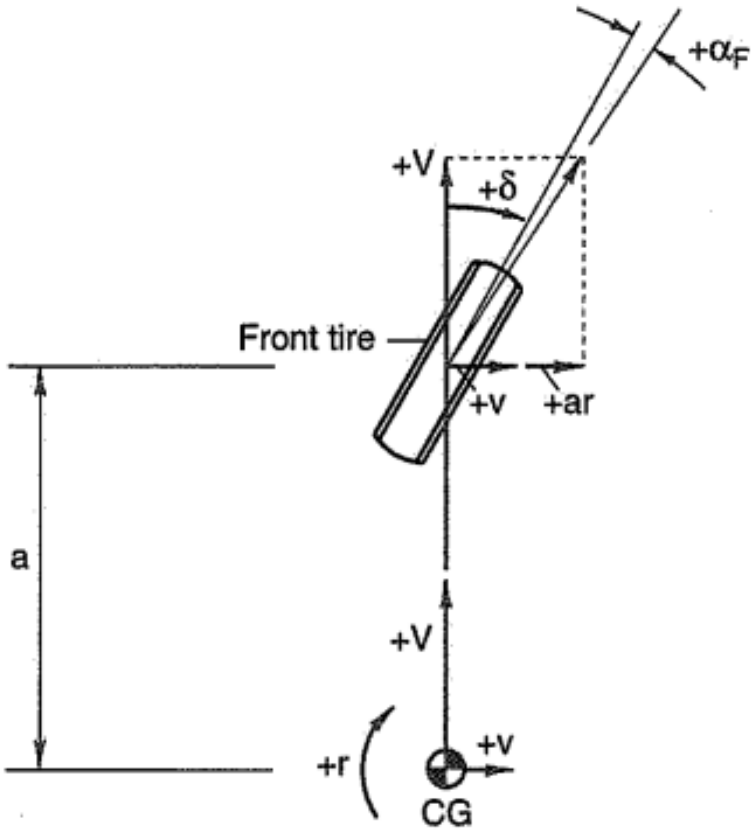

Figure 17: Forces on front tire cause a negative slip angle

The formula for slip angle is non-linear due to containing an arctan function. A simple linearization can be made by using small angle approximation which gets rid of the arctan for very small angles. This method works surprisingly well even when approaching larger angles such as 20 degrees and for our purposes is satisfactory.

$$\alpha_F = \arctan(\frac{v + ra}{V}) - \delta \approx \frac{v + ra}{V} - \delta \tag{16}$$

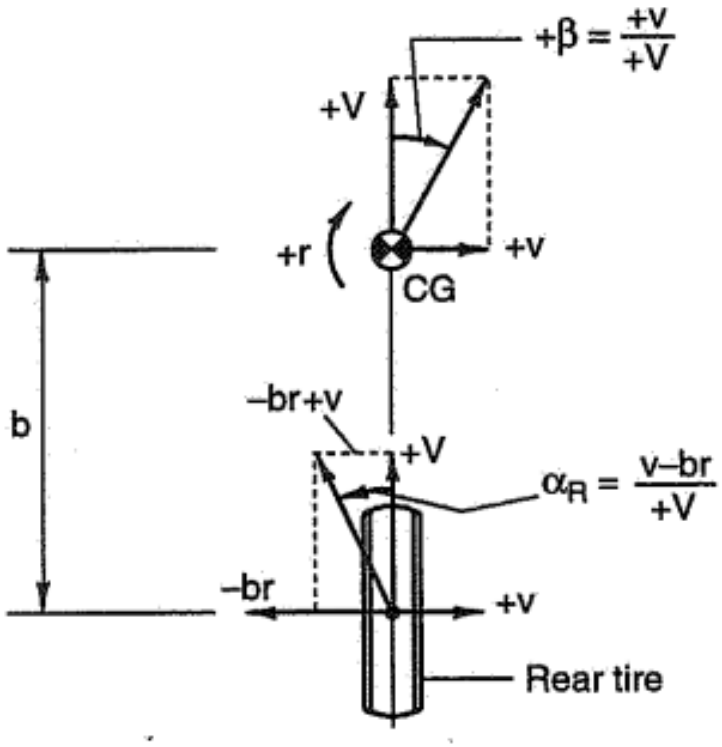

Figure 18: Forces acting on rear tire

Same small angle approximation can be made for the rear tire slip angles.

$$\alpha_R = \arctan(\frac{v - rb}{V}) \approx \frac{v - rb}{V} \tag{17}$$

Tire slip angles are then multiplied by the tires cornering stiffness coefficient C_F to find lateral force applied on the tires.

$$Y_F = (-C_F)\alpha_F = -C_F\frac{v}{V} - C_F\frac{ra}{V} + C_F\delta \tag{18}$$

$$Y_R = (-C_R)\alpha_R = -C_R\frac{v}{V} + C_R\frac{rb}{V} \tag{19}$$

Lateral acceleration consists of centrifugal force (V^2/R, since $r = V/R$ we get Vr) and direct lateral acceleration ($\dot{v}$).

$$a_y = \dot{v} + Vr \tag{20}$$

Applying 2nd law of motion:

$$(\dot{v} + Vr)m = Y_F + Y_R = -(C_F + C_R)\frac{v}{V} - \frac{C_F a - C_R b}{V}r + C_F\delta \tag{21}$$

$$I_z\dot{r} = Y_F a - Y_R b = -C_F a\frac{v}{V} - C_F\frac{ra^2}{V} + C_F a\delta + C_R b\frac{v}{V} - C_R\frac{rb^2}{V} \tag{22}$$

Solve for direct lateral acceleration and yaw acceleration.

$$\dot{v} = -v\frac{(C_F + C_R)}{mV} - r(\frac{C_F a - C_R b}{mV} - V) + \frac{C_F}{m}\delta \tag{23}$$

$$\dot{r} = -v(\frac{C_F a - C_R b}{I_z V}) - r(\frac{C_F a^2 + C_R b^2}{I_z V}) + \frac{C_F a}{I_z}\delta \tag{24}$$

Constants are grouped together.

$$\dot{v} = -Av - Br + C\delta \tag{25}$$

$$\dot{r} = -Dv - Er + F\delta \tag{26}$$

Took the laplace transform of both equations and by solving both for V(s) combined them into one equation.

$$\frac{-BsR(s)}{s^2 + As} + \frac{c\delta(s)}{s^2 + As} = -\frac{sR(s)}{D} - \frac{ER(s)}{D} + \frac{F\delta(s)}{Ds} \tag{27}$$

Manipulate equation to get an open loop transfer function from steer angle to heading angle.

$$\frac{R(s)}{\delta(s)} = \frac{Fs + FA - CD}{s^3 + s^2(A + E) + s(AE - BD)} \tag{28}$$

Because the A,B,D and E constants contain a longitudinal velocity V variable. Both lateral and yaw acceleration equations vary with longitudinal velocity. As a result, the steering angle transfer function will change as longitudinal velocity changes. To combat this issue, an adaptive PID controller was designed that changed values depending on longitudinal velocity to maintain similar performance throughout different velocities. First, some transfer function values at random velocities were calculated (withing the range of our car - 11 m/s or 25 mph).

$$0.5m/s -> \frac{81.73s + 1856}{s^3 + 281s^2 + 6513s + 0} \tag{29}$$

$$4.5m/s -> \frac{81.73s + 206.2}{s^3 + 31.28s^2 + 97.02s + 0} \tag{30}$$

$$11m/s -> \frac{81.73s + 84.36}{s^3 + 12.8s^2 + 30.25s + 0} \tag{31}$$

A PD controller was created in Simulink for each transfer functions. A square wave was generated, sending a high and low signal of 0.17 rad every 3 seconds. The wave would simulate sending a signal to turn the front wheels 10 degrees left and right. The signal was then converted from time domain into discrete domain using a zero order hold (ZOH) with a 30Hz frequency. This frequency was used because the cameras mounted on the GEM E2 by the Sensor team post data at 30Hz which is assumed to be the bottle neck in the whole system that determines the frequency rate. Proportional and derivative gains were found

that returned good performance. The ideal response had no overshoot and took most of the 3 seconds approaching the desired signal.

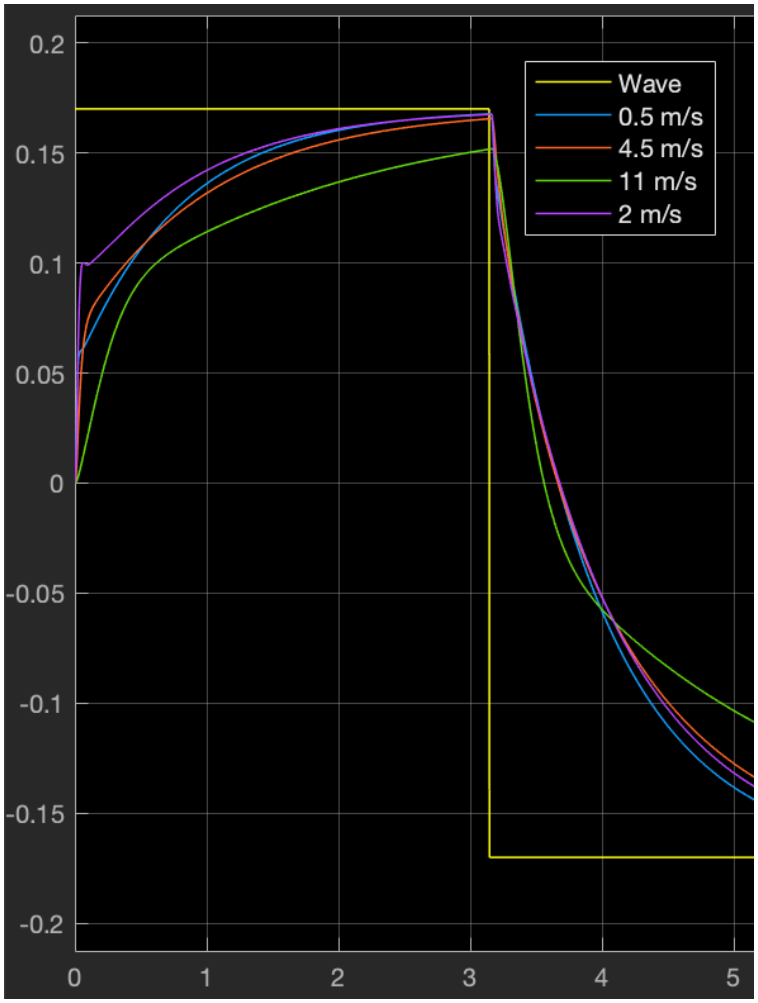

Figure 19: PD steering response stable at different longitudinal velocities

A response that instantly turns the wheels to the target angle was undesirable due to creating an uncomfortable driving experience. The proportional and derivative gains were graphed and an equation was found that could approximate PD values for any velocity of the car.

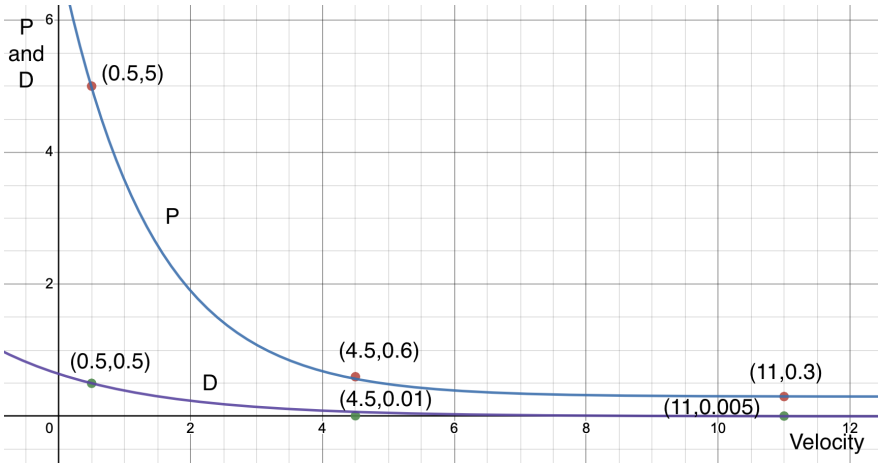

Figure 20: Polynomial functions found that fit both P and D values

$$P = 2.5 * V^{(-0.94)} \tag{32}$$

$$D = e^{(-\frac{V}{2} - 0.44)} \tag{33}$$

Variable	Description	Value (SI)	Source
m	Mass	544 kg	Service Manual
C_F	Front Tire Cornering Stiffness	40 000 N / rad	Model verification
C_R	Rear Tire Cornering Stiffness	30 000 N / rad	Model verification
a	Distance from CG to front axle	0.85 m	Estimated
b	Distance from CG to rear axle	0.9 m	Estimated
I_Z	Car moment of inertia around the z axis	416 Kg * m^2	Calculated using lumped mass model
F_BRAKE	Maximum tire braking effort		
δ_max	Maximum steering angle		

Figure 21: Vehicle parameters used in developing steering control

3.3.3 Path Following

In this section, we will delve into the path-following algorithm, which acts as a conduit, accepting input data from the path-finding algorithm and producing output data for the motion control feedback mechanism. We opted to employ the pure pursuit algorithm for path-following, and its performance was verified within the Gazebo simulator using a TurtleBot3 model. Importantly, we fine-tuned the pure pursuit algorithm to match the frequency of waypoints provided. However, due to sensor limitations, this algorithm was not implemented on the small-scale car.

For the purposes of validating our path-planning algorithm, we utilized the Gazebo simulator. Gazebo is an open-source tool that accurately simulates 3D dynamics. It provides various environments for indoor and outdoor testing, and it takes use of the power of the

Robot Operating System (ROS) [24] for communicating with the robotic models within the simulation. This ROS compatibility proved critical for our project, as both the small-scale and large-scale cars use ROS as their core communication framework.

As for the robotic model used within the Gazebo simulation, we chose one of the TurtleBot3 models. TurtleBot3 is a widely-used, open-source robot kit, equipped with pre-developed packages for use in simulation testing. These robot models are outfitted with LIDAR sensors, enabling us to obtain precise odometry data. This capability greatly facilitated the validation of our path-planning algorithm as we could assume the accuracy of the odometry data received.

The path-planning algorithm subscribes to the data published by the path-finding algorithm via ROS. This data comprises a list of coordinates on the x-y plane, defined in an absolute coordinate system with the origin point set by the Gazebo simulator. The first waypoint in the list represents the closest point for the robot to follow. By integrating the current robot location (using odometry data) with the initial set of waypoints the robot is intended to follow, we were able to generate a path for our motion control loops. A significant advantage of the pure pursuit algorithm is its ability to generate a naturally curved path, more conducive to a wheeled robot's movement compared to the linear trajectories produced by a Proportional-Derivative controller [30]. Moreover, our instructor recommended the pure pursuit algorithm due to its effectiveness and efficiency.

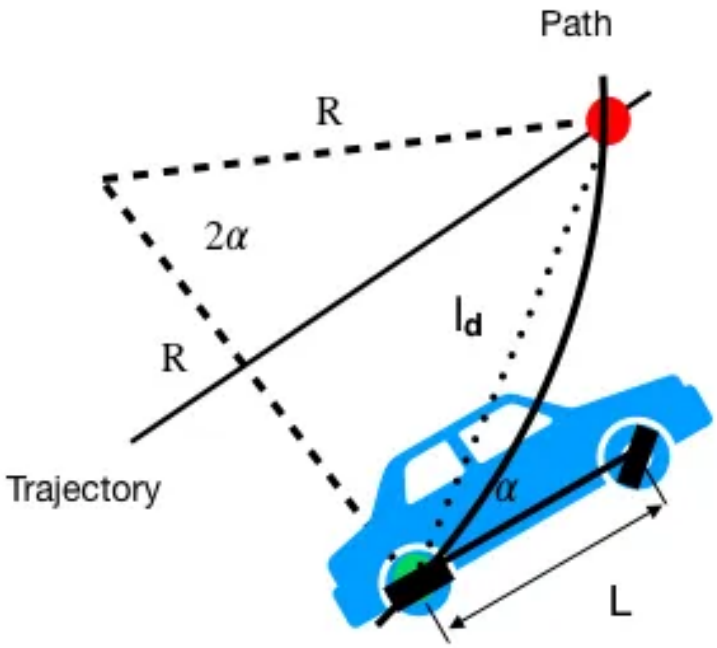

Figure 22: Pure Pursuit Geometry

The rear axle of the vehicle is set as the reference point in our pure pursuit algorithm. The look ahead distance is computed from the rear axle of the car to the designated look ahead point [30]. To formulate a unique curve from the rear axle to the look ahead point, an

isosceles triangle is constructed, where one side corresponds to the look ahead distance, and another side is vertical to the vehicle's wheelbase and passes through the reference point. The law of sines is then applied to calculate the vehicle's steering angle [30].

$$\delta = \arctan(\frac{2 * L * \sin(\alpha)}{l_d}) \tag{34}$$

δ = Steering angle

L = Wheel base

α = Desired angle

l_d = distance from waypoint

The desired angle is found with this equation:

$$\alpha = \arctan(\frac{y - rearY}{x - rearX}) \tag{35}$$

y = y coordinate of look ahead point

$rearY$ = y coordinate of center of rear axle

x = x coordinate of look ahead point

$rearX$ = x coordinate of center of rear axle

The selection of what waypoint to follow is an essential aspect of the pure pursuit algorithm. We determine a look ahead distance which forms a circle around the robot; the waypoint nearest to this circle is then selected. If the look ahead distance is too short, the ensuing path will exhibit high oscillation. This is due to the robot overemphasizing reaching a specific waypoint, rather than adhering to the overall path. On the other hand, if the look ahead distance is set too far, the robot will substantially cut through sharp turns and primarily move in a straight line, neglecting minor deviations in the path.

The illustrations below exemplify a look ahead distance that is too small, denoted as $l_d = 0.2$, as well as an appropriate look ahead distance, marked as $l_d = 2$. Additionally, we implemented a look ahead gain that increases the look ahead distance proportionately to the robot's velocity. As the robot's speed increases, it has less time to adjust its path towards a waypoint. Thus, the look forward gain ensures the selected waypoint remains at an optimal distance, irrespective of the robot's velocity [30]. In our case, we utilized a look ahead gain of 0.4.

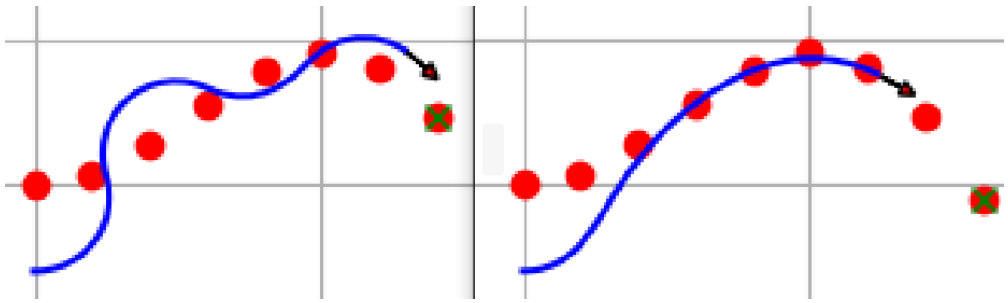

Figure 23: Too short look ahead distance VS appropriate look ahead distance

Once the desired steering angle toward a selected waypoint has been calculated, this value is transferred to the steering control system, as developed in the previous chapter. The current yaw angle is obtained from the odometry data and the difference between the desired yaw and current yaw is calculated. This difference is then multiplied by the proportional gain. Simultaneously, the angular velocity, which is also extracted from the odometry data, is multiplied by the proportional gain. The resulting value is then forwarded to the TurtleBot as the desired angular velocity.

For longitudinal velocity, a constant value was chosen as the target velocity. The TurtleBot3, using PID control, tries to match this target velocity. The error between the current velocity and the desired velocity is determined, just as it was for steering. The longitudinal velocity is computed by multiplying the error by a proportional gain, adding to that the linear acceleration multiplied by the derivative gain, and then adding an array of the most recent error values multiplied by the integral gain.

The velocity and steering values are then published through ROS for the TurtleBot3 to subscribe to, enabling it to move appropriately within the simulation. The TurtleBot3's response to a sharp turn was recorded and is displayed below:

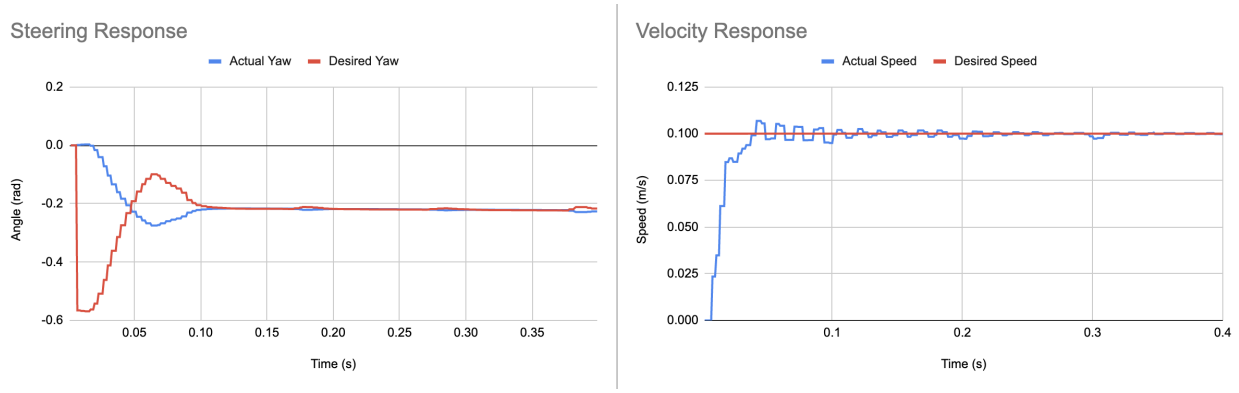

Figure 24: Steering and Velocity response of TurtleBot3 in Gazebo to a sharp turn

Both the steering and velocity feedback control mechanisms converge to the desired value in less than 0.1 seconds. The convergence of the steering angle at -0.2 rad illustrates one of the valuable features of the pure pursuit algorithm. Rather than directing the steering controller to execute a sharp turn until the robot's heading aligns with the desired heading, the pure pursuit algorithm instructs the controller to adjust the steering angle slightly and drive towards the desired waypoint following a curved path. This results in a movement that is far more natural for a four-wheeled vehicle.

One limitation of using TurtleBot3 models for testing the pure pursuit algorithm is the fact that TurtleBot3 models are two-wheeled vehicles. A two-wheeled vehicle lacks a wheelbase, which is a parameter in the pure pursuit equations. Therefore, a wheelbase of 1 meter was inserted into the equation in order to obtain similar results to that of our small scale car.

3.4 Testing

To test the effectiveness of our pure pursuit algorithm, we used the Gazebo simulator and one of the TurtleBot3 models to use within it. We loaded up a house map that is one of the default maps in Gazebo and includes one floor of a house with furniture that the robot can navigate through. We ran 50 test runs of placing the TurtleBot3 model in a random place on the map, and telling the robot to go to a desired waypoint with a specific end pose. If the Turtlebot3 was unable to reach the desired waypoint, then the test run failed. A sucesfull run would be reach within 1 foot of the desired waypoint.

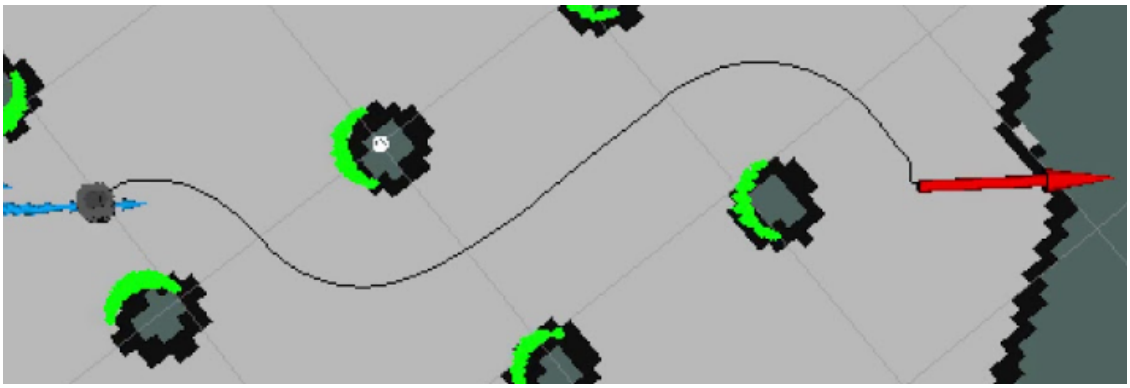

Figure 25: TurtleBot3 is placed randomly on the map and an end position and pose is selected (red arrow)

Out of 50 trail runs, the TurtleBot3 was able to reach the final destination 48 times. In two runs the robot got stuck on its way to the waypoint and could not get unstuck. The reason behind getting stuck was when the TurtleBot3 was trying to turn around a sharp corner and the planned out path was too close to that corner. As the robot tried to avoid crashing into the corner, obstacle avoidance on the robot caused it to further deviate from the path, causing it to attempt to follow the same path again, getting stuck in an infinite loop. Possible solutions to fix this problem are to generate paths further away from obstacles or pre-map an environment with smooth corners.

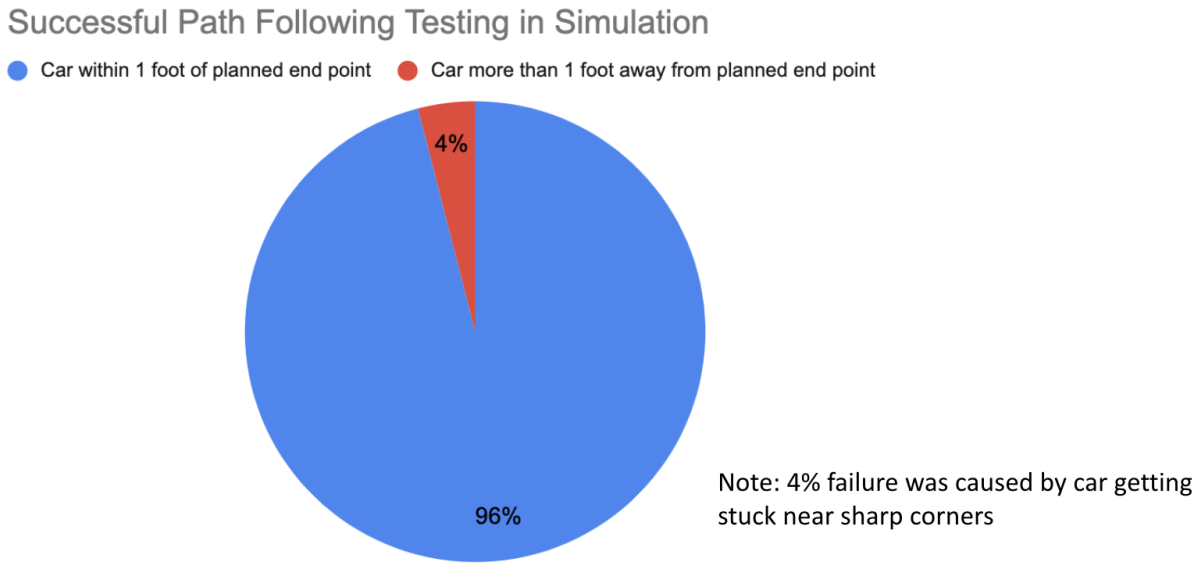

Figure 26: Success rate of TurtleBot3 reaching a desired destination in Gazebo

3.5 Framework and Visualization

3.5.1 ROS

We decided to use ROS [24] for our main framework. Every Autonomous System requires a message queue framework to handle various different protocols and different autonomous system nodes. It is open-source and has a large collection of libraries and tools for us to use and create new software libraries with.

It can allow us to make our platform scalable and interoperable with various platforms. as ROS runs on various Linux compute platforms such as Ubuntu, Debian, Linux. It is currently now maintained by the Open Robotics Organization [25].

ROS has many of main items we can utilize to build our framework. ROS works by using Nodes, Messages, Topics, Services, Parameters Server, Packages, Tools and Simulation ecosystems.

3.5.1.1 Nodes

Nodes are the building blocks of ROS [24]. Each software module will be a node and an executable will be a Node. It will perform a very specific task given various inputs of the robot system. This could range from computing PID values to controlling a motor to getting data from a sensor. Nodes use a very specific way of communicating with the through ROS Messages.

Nodes communicate with each other by passing very structured messages. These structured message will be data structures that contain specific formats of information. This format can be any data format and is data structure agnostic. It can be things such as Linked Lists or just a simple string or int variable. Nodes can be used to publish these

messages to a Topic to transfer information collected. This is where the publisher-subscriber mechanism comes into place. This mechanism allows for there to be asynchronous communication [24, 25]. This means that the nodes we create can operate independently and transfer information without having to be blocked.

Nodes are also meant to be independent to run concurrently. This means they can be modular to have parallel processing so there is prime efficiency in processes completing and computational resources being used correctly.

Nodes are also programming language agnostic. ROS supports multiple programming languages, including C++, C, Python, and more [24, 25]. Nodes can be implemented in different languages, as long as they adhere to the ROS communication protocols. Our team is very happy with this his flexibility since it allows us to choose the programming language we are most comfortable with or utilize existing code bases and packages.

Nodes in ROS are relatively lightweight. This means that they do not take up that much space or use a large amount of computational power to run. This allows for the deployment of multiple nodes on a single computer and especially a small compute board like the Jetson NANO where thermal throttling is a huge issue [6]. This scalability enables the creation of complex robot systems that involve multiple sensors, actuators, and computational units.

Nodes have a well-defined life cycle within the ROS framework. This means that they can be started, stopped, paused and resumed as needed and whenever regardless of time. This lifecycle management facilitates system initialization, and shutdown procedures

Nodes also have a naming configuration that allows us to be flexible. These are done using Namespaces similar to what we do in C++. Namespaces provide a way to organize nodes and topics hierarchically, allowing for better organization and reducing naming conflicts in large systems.

Nodes have a concept called Nodelets which ROS also provides. Nodelets allow for more efficient communication and resource utilization by running multiple nodes within a single process. This reduces overall computational power and memory usage. This is extremely useful when running complex program and stacks in compute platforms such as Jetson Nano to get most performance out of it [6].

3.5.1.2 Messages

Nodes exchange data by sending and receiving messages [24]. A message is a data structure that contains information about a particular aspect of the robot's state, such as sensor readings, motor commands, or pose information. Messages are defined using a language called the Robot Description Language (RDL) and can be customized to fit the specific needs of the robot application.

RDL provides a syntax for describing the structure and content of messages, including fields, data types, arrays, and nested structures. Messages can contain various data types, including primitive types such as integers, floating-point numbers, strings, and boolean values. Additionally, messages can include more complex data structures, such as arrays, nested messages, and custom-defined types [25]. This sort of flexibility of data structures allows us to represent various data points in various data types.

ROS provides specific standard message types we can use out of the box. These cover common robotic movements and sensor data. For example, it covers images, point cloud

data, control commands, odometery data, and specific transformation data.

3.5.1.3 Publisher-Subscriber

The Publisher Subscriber model defines how nodes and topics communicate through ROS [24, 25]. A node publishes a message on a specific topic and then another node subscribes to that topic. This asynchronous independent way of communication allows us to have a great way of message communication. This also enables multi-node communication where multiple nodes can publish or subscribe to the same topic [25].

A publisher is a node that sends messages on a specific topic. It creates instances of the message type, fills them with data, and publishes them on the topic. Publishers are responsible for making data available to other nodes that are interested in receiving it.

A subscriber is a node that receives messages from a topic. It subscribes to a topic and waits for messages to be published on that topic. When a message is published, the subscriber's callback function is triggered, allowing the subscriber to process the received data.

Publish-Subscribe enables asynchronous communication, meaning publishers and subscribers can operate independently and at their own pace. Publishers can continue to publish messages regardless of whether there are active subscribers, and subscribers can receive messages whenever they become available. This asynchronicity allows nodes to work concurrently and efficiently, without waiting for immediate responses [25].

3.5.1.4 Packages and Environment

The entire codebase we wrote is a package in ROS [25]. A package consists of a directory with a specific structure. The main components of a package include source code files, configuration files, launch files, message and service definitions, and any additional resources required by the package. Each package is identified by a manifest file named package.xml. This file contains metadata about the package, such as its name, version, dependencies on other packages, maintainers, and a description. The manifest file is essential for ROS to recognize and manage the package. Packages can specify their dependencies on other packages in the package.xml file [25]. This allows ROS to automatically resolve and manage package dependencies during the build process. ROS provides a package manager called rosdep to handle dependency installation, ensuring that all required dependencies are available. We make these packages use the catkinmake command.

3.5.2 RVIZ

RVIZ is an environment that we used for real-time transfer and visualisation of autonomy data [16]. we chose Rviz because of its popularity in the field of self-driving cars. Rviz supports visualizing point clouds, laser scans, images, 3D meshes, occupancy grids, and other sensor data. This sort of tool is essential for debugging any issues and to recreate any scenario we want.

There is a 3D Gui that allows us to visualize sensor data [16]. We can use the GUI to get different perspectives of the scenario. ROS [24] also provides TF packages to have different coordinate frames and visualize them by transforming them. For example, our IMU gave

sensor data represented in Quaternions and we used the Tf package to transform it into the Cartesian Coordinate Frame.

RViz is tightly integrated with the ROS ecosystem [16]. It can subscribe to ROS topics to receive sensor data and robot states for visualization. RViz also publishes selected visualization data as ROS topics, allowing other nodes to access the visualized information for further processing or analysis.

We primarily used RViz as a debugging and development tool in ROS. It allowed us to inspect sensor data, check car states, and validate algorithms by visualizing the results. RViz aids in understanding the behavior of the full system and diagnosing issues during development and testing.

This visualization aids in perception tasks like object detection and tracking, providing a comprehensive understanding of the vehicle's surroundings. Furthermore, Rviz offers a 3D representation of the environment, including the road, obstacles, traffic signs, and other relevant objects. This representation is crucial for planning and decision-making algorithms, as it allows developers to visualize the scene from the car's perspective and analyze how the vehicle interacts with its surroundings.

Another essential feature of Rviz is trajectory visualization. It can display planned trajectories or paths for the self-driving car. This functionality enables developers to assess and fine-tune path planning algorithms, ensuring that the vehicle follows smooth and safe paths in different scenarios. Rviz also provides interactive controls, allowing users to modify the visualization settings and view parameters [16]. With the ability to zoom in/out, rotate the view, and change perspectives, developers can examine the system's behavior in detail and gain a comprehensive understanding of its functionality.

Lastly, Rviz serves as a valuable tool for debugging and analysis purposes. It allows users to visualize and inspect the data generated by various components of the self-driving car system, facilitating the identification and resolution of issues or discrepancies.

4 Integration

Our team successfully transferred our software stack from a simulated environment in Gazebo onto a small-scale car, eventually extending its implementation to a full-scale vehicle. The platform for these experiments was an inexpensive RC car, which we extensively modified to accommodate the necessary sensors and computational hardware. We developed a modular three layer structure for our RC car, designed to facilitate straightforward modifications and troubleshooting across all components of the vehicle. The selection of sensors and computational boards for the small-scale car mirrored those chosen for the full-scale vehicle to mitigate potential integration issues down the line. The small-scale car demonstrated its capability to autonomously navigate a pre-mapped environment. In addition to the vehicle modifications, we also elaborate on the construction of the small-scale track that mimics real-world driving conditions at a reduced scale. After successful integration and testing on the small-scale car, the team began preliminary work on the full-scale vehicle, yielding promising results.

4.1 Small scale car

The small-scale car we designed is a heavily modified OSOYOO RC car, which is capable of autonomous driving on a small-scale track. This car features a three-layer structure for component placement. The bottom layer houses the steering and throttle components. The steering is accomplished through an SG90 servo, which is attached to a steering shaft. When the car is stationary, the steering shaft points towards its rear. As the servo moves the steering shaft from left to right, the center link simultaneously adjusts both front wheels to the same angle.

Figure 27: Small scale car steering system controlled by servo

The rear wheels are powered by a 24V brushed DC motor, model PZ22GR9120R-084BBH. This motor has a gear attached to the motor shaft, which interacts with another gear attached to the rear axle. These gears, having a 1:1 gear ratio, power both wheels. The wheels are friction-fitted onto the rear axle. The car currently lacks a differential on both the front and back wheels. This has led to problems, as the rear wheels tend to fall off when the car repeatedly turns in a single direction. The motor is powered by an H-bridge, integrated into the Arduino microcontroller. The maximum voltage the Arduino can send to the motor is 9V. We found that the motor starts propelling the car at 2V, establishing our velocity range corresponds to 2-9V. The original motor in the OSOYOO kit had to be replaced due to insufficient torque. After we added a third level to the car for the sensors and microcontroller, the additional weight was too much for the original motor. The replacement, a 24V brushed DC motor, has double the required torque to move the car. However, it comes with the trade-off of a lower RPM. This slower speed doesn't pose an issue for our project, as it allows our software stack additional time to execute lane-following algorithms.

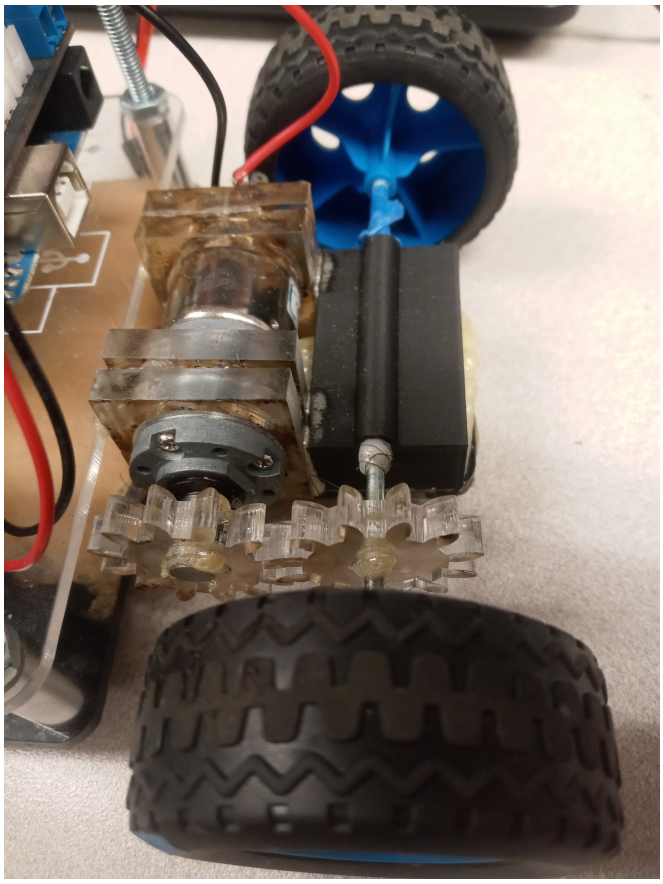

Figure 28: Small scale car with custom drive-train system

The second layer of the car accommodates the Arduino and a battery that powers it. The Arduino is responsible for controlling the RC servo and the DC motor that facilitate the car's movement. The Jetson Nano, situated on the top layer, sends motion commands to the Arduino using the UART protocol. We employ the Arduino in this manner due to the Jetson Nano's lack of PWM pins, which are typically used to control the servo and motor. The Arduino, on the other hand, does possess PWM pins, and the Arduino IDE simplifies the task of motion control.

When transmitting over UART, the desired steering angle in degrees is sent first, followed by the motor throttle, which is expressed in percentages. One hundred percent corresponds to 9V, while zero percent gives the motor less than 2V. Due to the RC servo's limitations, the steering angle is capped at forty-five degrees in both directions. To differentiate between the steering and throttle values, a colon is placed between the two commands. To verify the packet's integrity, a checksum is added at the end of the message. We utilize the Berkeley Software Distribution algorithm for our checksum calculations. The complete packet follows this structure: $\{+\#\# : \#\# : \#\}$, which denotes the steering angle in degrees, followed by a colon, then the throttle in percentages, another colon, and lastly, the checksum. The Arduino is powered by a Li-ion 9V 650mAh battery, which was included in the OSOYOO kit.

The top layer houses the Jetson Nano, battery, buck converter, camera, IMU, WiFi

antenna, and ToF sensor. We selected the Jetson Nano as the microcontroller for our small-scale car due to its architectural similarities with the Jetson Orin [6]. The sensor team working on the full-scale car chose the Jetson Orin because it's designed for industrial systems and has the capacity to process large neural networks, making it ideal for computer vision tasks. However, the Jetson Nano is more appropriate for the small-scale car due to its lower cost and power consumption. Although the Nano isn't as powerful as the Orin, this trade-off is acceptable given our constraints. Since we are only using one camera positioned at the front of the car, a less powerful compute board suffices. The lower cost, weight, and power consumption make the Jetson Nano a more suitable choice for small-scale testing in comparison to the Orin.

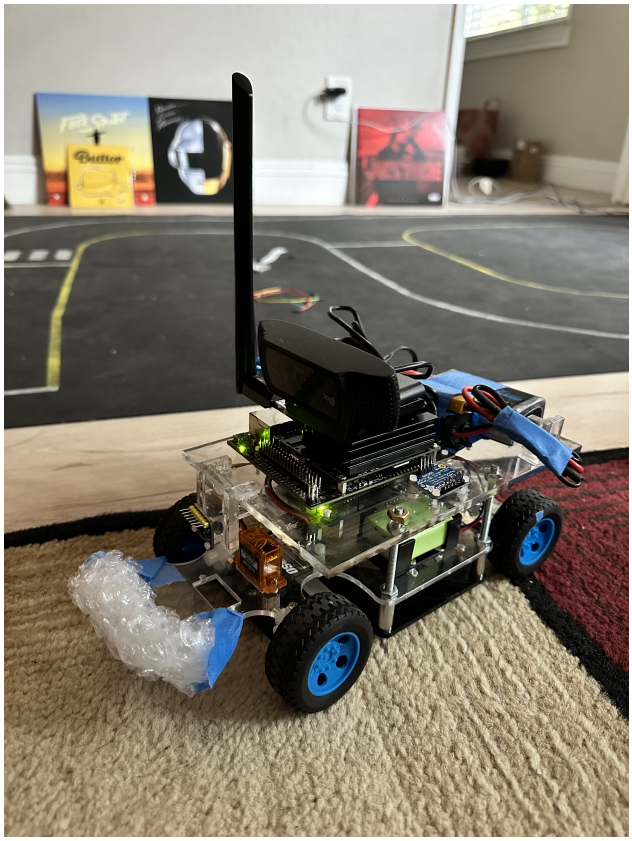

Figure 29: Small scale car with every sensor implemented

The Jetson Nano is powered by a Li-Po 11.1V, 1300mAh battery. This battery outputs 11V DC, which is too high for the Jetson Nano that operates at 5V DC. Therefore, we purchased a buck converter that steps down voltages from a 8-20V range to a more manageable 5V. The Li-Po battery also exceeds the minimum power output of 10W, ensuring that it meets the Jetson Nano's requirements.

For object detection, we implemented a ToF VL6180X sensor, strategically placed at the front of the car facing forward, to detect any objects in its path. The VL6180X sensor can detect an object up to 10cm away with up to 1mm precision. Given that our small-scale car is approximately 20cm in length, the ToF sensor can effectively detect an obstacle up to

half a car length away. Should the ToF sensor detect an object, it notifies the Jetson Nano, which in turn sends a stop command to the Arduino. Thanks to the car's low-rpm motor, it has sufficient time to halt before colliding with the detected object.

The IMU BNO055 sensor was used for position tracking. We opted for the BNO055 sensor as we had previously used it in another class and had readily available calibration code that we could repurpose for this project. The IMU's accelerometer sensor aids in estimating the car's position. However, due to the inherent drift error accumulation in the IMU, it's crucial to use other sensors in conjunction with the IMU for more accurate position estimation. The data from the IMU is transmitted to the Jetson Nano via the I2C protocol.

Our computer vision capabilities are powered by the Logitech C920 HD camera. This camera, connected to the Jetson Nano via one of its USB-A ports, is mounted on top of the Jetson Nano to provide the best possible field of view. Ideally, the car should be able to view both lanes on the small-scale track at all times. However, while turning, the inside lane often slipped out of view. We remedied this by positioning the camera atop the Jetson Nano's heat sink. Overheating concerns for the Jetson Nano were mitigated by streamlining the Linux kernel and disabling some features to prevent overheating while processing computer vision on the camera feed.

A WiFi antenna, connected to the Jetson Nano via a USB-A port, enables the car to be remotely controlled [6]. Through a computer connected to the car via WiFi, commands can be sent to halt the car when necessary.

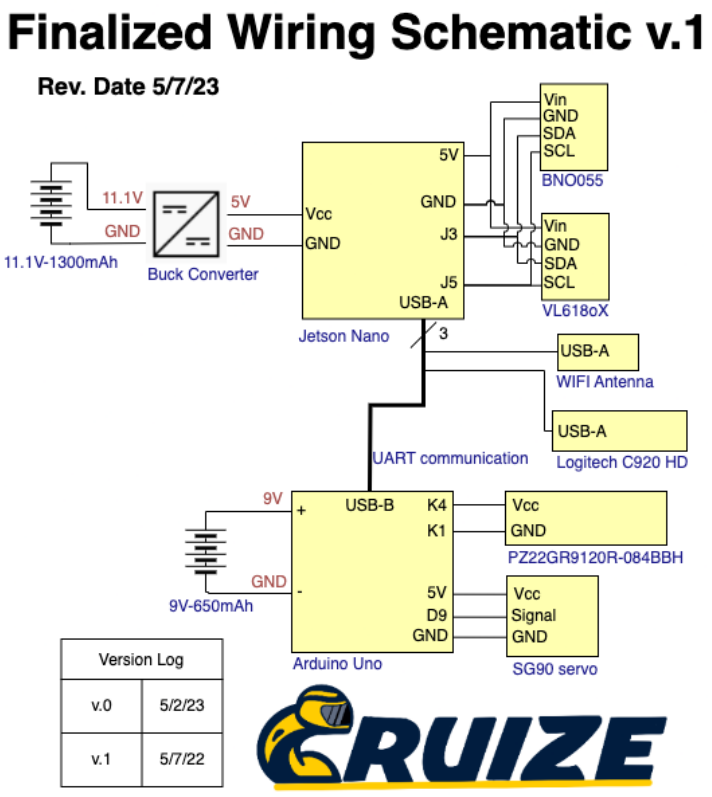

Figure 30: Wiring schematic showing what sensors are controlled by what compute board

Our small-scale car was thoughtfully designed with easy disassembly in mind, specifically to facilitate debugging efforts. All three layers of the car are interconnected by four long screws. Spacers are placed between the first and second layers to ensure a consistent distance between them. The third layer rests on four nuts attached to the screws, and additional nuts are placed above the third layer, effectively sandwiching it. The third layer can be easily removed by unscrewing the four topmost screws and disconnecting the UART cable. The second layer can then be detached by removing the screws that secure it. This modular design allows for straightforward access to any component as needed.

We also constructed a testing track to simulate real-world conditions for the car, aiming for data consistency with full-scale environments. The track is a small-scale version of a road, complete with a sidewalk, a stop sign, multiple turns, and an intersection. Measuring 4 feet in width and 7 feet in length, the track is made entirely from rubber, chosen to mimic the texture of an actual road. Road lanes were painted using water-resistant white and yellow paint, ensuring that our lane detection code could recognize both colors. The car's size is approximately one-sixteenth that of an average-sized vehicle, so lane widths and inter-lane distances were accordingly scaled down to 0.8cm and 22cm, respectively. Likewise, the turn radius of curved road sections was designed to replicate those of real-life intersections; a typical intersection's turn radius of 4.57m translates to 30cm on our scaled-down track.

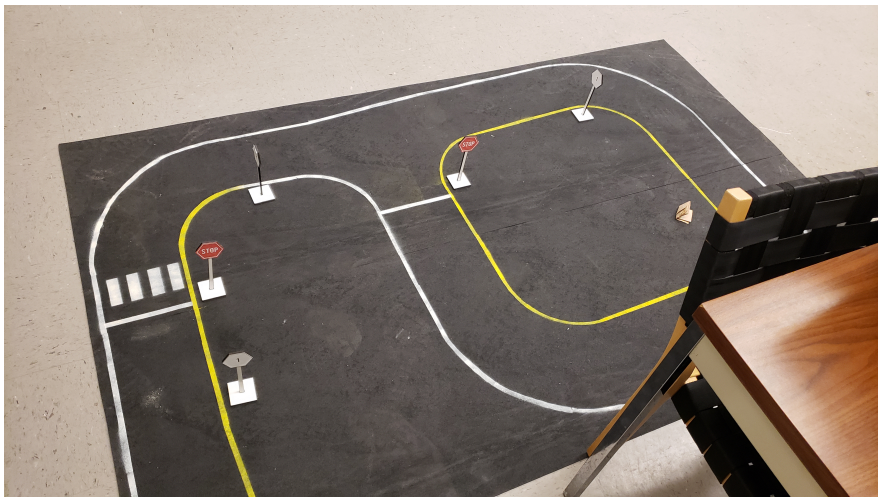

Figure 31: Portable Testing track made from Rubber

Road signs were positioned at various points along the track to evaluate the object classification capabilities of the car. The signs were also scaled down, designed to align with the camera on the small-scale car, mirroring the camera placement on a full-scale vehicle. The signs were manufactured from MDF and manufactured using a laser cutter. When the car detects a sign, a counter is activated. Upon reaching its limit, the counter triggers the car to stop. This delay mechanism was implemented due to the car's ability to detect signs from a substantial distance; we wanted the car to stop closer to the sign, mimicking real-world reactions to a stop sign.

Figure 32: Small scale Road signs used to test object detection

4.1.1 Validation of Small scale car

To assess the mechanical durability of the small-scale car and the reliability of our software, we conducted a series of endurance tests involving multiple trial runs. The car was initially positioned at the start point on the track and programmed to autonomously navigate in clockwise loops around the circular section of the track. We then recorded the number of laps completed before any mechanical or software failure occurred. As part of our success criteria, we established a target of a minimum of ten laps before encountering any performance issues. This target would not only represent a benchmark for mechanical stability but also provide ample time to gather significant data for analysis. The endurance tests helped us identify potential weaknesses in the car's mechanical design and the software's ability to function over extended periods, which are both crucial aspects of an autonomous vehicle's reliability.

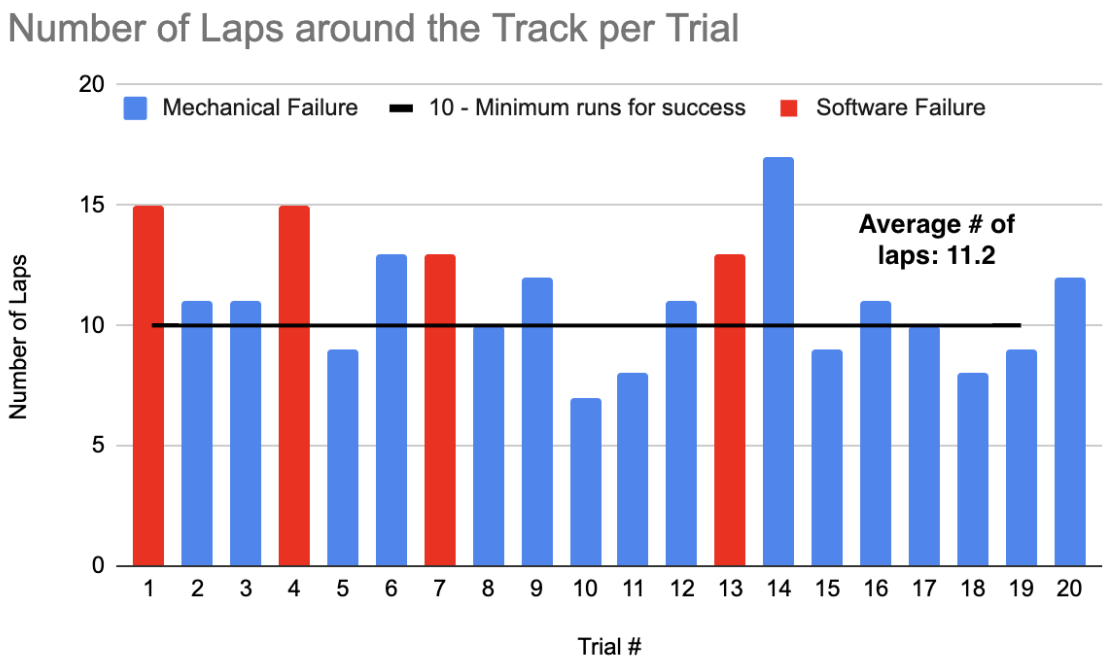

Figure 33: Car successfully navigates around the track more than 10 times

Averaging slightly over eleven laps, the car demonstrated sufficient mechanical stability for data collection purposes in our research. The major recurring mechanical issue that led to failures in almost every run was the detachment of the rear right wheel. This problem was primarily caused by the absence of a differential on the rear axle and the constant right turns on the circular portion of the track. During a right turn, the inner rear right wheel naturally tends to rotate slower than the outer wheel. However, the lack of a differential prevents this, creating a force that pushes the wheel off its axle. Given that the wheel was only fitted through friction, it inevitably detached over time. This issue could be easily addressed by future researchers either by integrating a differential into the drivetrain or opting for higher quality wheels that are not merely friction-fitted to the rear axle.

4.2 Full Scale Car

In addition to integrating and testing our code on the small-scale car and the full-scale car, we began preliminary integration and testing on the full car from the hybrid systems lab. Reaching this stage was a stretch goal for the project, and while we were unable to do extensive testing, we were able to get prove our code's transferability between platforms. We began the full-scale car integration process by downloading our code onto the Jetson Orin used in the full-scale car and verifying it could run on the platform. As expected, our code developed on the Jetson Nano also ran on the Jetson Orin [5] without issue.

Figure 34: An image of the full-scale car from the hybrid systems lab

We next began the testing of our perception code on the full-scale car. We set up our code to receive the inputs from the sensor team's sensors, and had the code log the recorded videos and frames from the car. Videos of these tests can be found here. In each of the provided videos, lane detection runs to highlight found lanes red. Object detection locates pedestrians and draws green bounding boxes around, labels them, and outputs the model's confidence in its prediction. Additionally, in one of the tests, "Depth Test", we demonstrate our successful depth perception by displaying the depth map from the stereo cameras and the predicted distances of pedestrians in meters.

Since integration on the full-scale car was a stretch goal, we did not have time to do any thorough testing of the accuracy of the perception when running on the full-scale car's hardware. Instead, we simply proved that our code could run on it, although further calibration may be necessary. Due to the car's drive-by-wire not being done yet, we were not able to have the car autonomously drive. However, we still proved our full pipeline is capable of running on the car and our perception software can interface with the car's sensors.

5 Conclusion

The team successfully developed a research platform, designed and validated in a simulated environment, on a small-scale car, and ultimately integrated onto a full-size vehicle. Leveraging the modularity of the ROS framework, individual components could be added or removed seamlessly without disrupting the overall functioning of the software stack. Throughout all three stages of the pipeline, the on-board computer system effectively processed sensor data to inform future vehicular actions. In both the simulation and the small-scale car, the Nvidia Jetson Nano proved capable of generating a navigational path within a pre-mapped environment based on received sensor data. The Nano was then able to make real-time decisions on the actions to take, translating these plans into commands for the motion actuators via an Arduino board. This demonstrated the robustness of our software stack in a simulated environment, as well as on the small-scale car; both platforms were able to successfully navigate their respective environments, whether it be any point on a pre-mapped virtual terrain or laps around a physical small-scale track. The team also demonstrated the interoperability of our software stack by transferring the software stack from the Jetson Nano environment to a Jetson Orin [5] on the full-size car. Our software stack was again successful in receiving sensor data and generating executable plans, although the actual car's motion actuators were not functional at the time of the project. In summary, the team not only met but exceeded the goals of the project by beginning the process of full-scale car integration, a task initially mentioned as a stretch goal. The Cruize AI software stack has proven its effectiveness in all three stages of development, laying a solid foundation for future research and development in the realm of autonomous driving algorithms.

6 Challenges and Learning

Throughout this project, we learned as a team how to use the Agile methodology effectively to organize our teams workload and how to communicate among team members. Even though we are a three person team, at the beginning of winter quarter there was much enthusiasm to go design parts of the self driving car individually. Because of that many days of work have been lost due to working in software that was not compatible with the rest of the teams design. As an example we created a 3D model of the large scale car in Blender to be used in the Carla simulator. If there was frequent communication among the team members we would have realized that the Gazebo simulator was the better option given the context of the project and we could have saved two weeks of work. Once we understood how to use the Agile methodology, we knew very well through daily stand-ups and meeting minutes what each team member was doing and how our part of the project fit into the big picture.

7 Next Steps

Our team in conjunction with the self-driving car sensor team made tremendous progress on both the hardware and software for Hybrid Systems Lab's self-driving car. However, there is much more work to be done on the full-scale car. The most pressing issue is the lack of

functional drive-by-wire. Drive-by-wire was never fully integrated onto the full-scale car. For this reason, we were unable to test our full software pipeline on the car and instead, we had to rely on testing through simulation and small-scale models. Finishing the drive-by-wire system would be a priority for any capstone team continuing this project. In addition to this, there will likely be more work to do to get our software stack fully and safely integrated onto the full-scale car.

Next, our software pipeline was created with a pre-mapped environment in mind. Our code only does rudimentary SLAM (simultaneous localization and mapping) and instead relies on the assumption of a known environment. For the car to work more generally in unknown environments, a future team would have to put more time and work into creating a robust SLAM algorithm optimized for city streets where the car operates.

References

[1] The 2005 DARPA grand challenge: The great robot race.

[2] Atmospheric pressure in santa cruz (monterey bay) with trend indicator.

[3] GEM e2 owners manuals.

[4] J2452_201707: Stepwise coastdown methodology for measuring tire rolling resistance - SAE international.

[5] Jetson AGX orin developer kit user guide.

[6] Jetson nano developer kit | NVIDIA developer.

[7] Newly released estimates show traffic fatalities reached a 16-year high in 2021 | NHTSA.

[8] Santa cruz climate: Average temperature, weather by month, santa cruz water temperature.

[9] VL6180x - time-of-flight (ToF) proximity sensor and ambient light sensing (ALS) module - STMicroelectronics.

[10] A. Broggi, P. Medici, P. Zani, A. Coati, and M. Panciroli. Autonomous vehicles control in the VisLab intercontinental autonomous challenge. 36(1):161–171.

[11] Tausif Diwan, G. Anirudh, and Jitendra V. Tembhurne. Object detection using YOLO: challenges, architectural successors, datasets and applications. 82(6):9243–9275.

[12] Simulink Documentation. Simulation and model-based design, 2020.

[13] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Pearson, 2nd edition edition.

[14] Thomas Gillespie. *Fundamentals of Vehicle Dynamics, Revised Edition*. SAE International, 506 edition.

[15] George Heinzelman. *Autonomous Vehicles, Ethics of Progress.*

[16] HyeongRyeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. RViz: a toolkit for real domain data visualization. 60:1–9.

[17] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. 4(1):15–24.

[18] S. LaValle. Rapidly-exploring random trees : a new tool for path planning.

[19] James Patrick Massey. Control and waypoint navigation of an autonomous ground vehicle.

[20] VanQuang Nguyen, Heungsuk Kim, SeoChang Jun, and Kwangsuck Boo. A study on real-time detection method of lane and vehicle for lane change assistant system using vision system on highway. 21(5):822–833.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library.

[22] Dean A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann.

[23] Simon J. D. Prince. *Computer Vision: Models, Learning, and Inference.* Cambridge University Press, 1st edition edition.

[24] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source robot operating system.

[25] Morgan Quigley, Brian Gerkey, and William Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System.* O'REILLY.

[26] M V Rajasekhar and Anil Kumar Jaswal. Autonomous vehicles: The future of automobiles. In *2015 IEEE International Transportation Electrification Conference (ITEC)*, pages 1–6.

[27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. arXiv.

[28] Robert Sedgewick and Philippe Flajolet. *Introduction to the Analysis of Algorithms.* Addison-Wesley Professional, 2 edition.

[29] Charles Thorpe, Miriam Herbert, Takeo Kanade, and Steven Shafer. Toward autonomous driving: the CMU navlab. part i - perception. 6:31–42.

[30] Steven Waslander. Coursera self-driving cars specialization.

[31] Junqing Wei, Jarrod M. Snider, Junsung Kim, John M. Dolan, Raj Rajkumar, and Bakhtiar Litkouhi. Towards a viable autonomous driving research platform. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 763–770. IEEE.

[32] Xianwen Wei, Zhaojin Zhang, Zongjun Chai, and Wei Feng. Research on lane detection and tracking algorithm based on improved hough transform. In *2018 IEEE International Conference of Intelligent Robotic and Control Engineering (IRCE)*, pages 275–279.

[33] Edward M. Kasprzak William F. Milliken Douglas L. Milliken. *Race Car Vehicle Dynamics - Problems, Answers and Experiments.* SAE International, pap/cdr edition edition.

8 Relevant Links/Appendix

Github Repo: Link

Small scale car parts list: Link

Full Scale Car Integration Demos Link