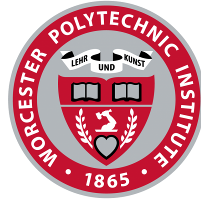


WPI

Modular Package for Autonomous Driving (MPAD)

A Major Qualifying Project

Submitted to the Faculty of Worcester Polytechnic Institute

In fulfillment of the requirements for the degree in

Bachelor of Science

By

Enzo Giglio de Azevedo (ECE/IE)

Antonio Jeanlys (ECE)

Chris Mercer (CS)

Julien Mugabo (ECE)

Eric Reardon (CS)

Taylan Sel (ECE)

Advised by:

Professor Pradeep Radhakrishnan (RBE/ME)

Professor Kaveh Pahlavan (ECE/CS)

Professor Walter T. Towner (IE)

May 7, 2020

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see http://www.wpi.edu/academics/ugradstudies/project-learning.html.

Table of Contents

Table of Contents 1

List of Figures 5

List of Tables 7

Abstract 9

Acknowledgements 12

Executive Summary 13

Authorship Table 14

1. Introduction 15
- 1.1 Report Organization 16

2. Previous MQP (2019-2020) 18
- 2.1 Project Summary 18
- 2.2 Moving Forward 24
- 2.3 Conclusion 26

3. Goals and Requirements 27
- 3.1 Lane Following 27
- 3.2 Web Dashboard 28
- 3.3 Sensor Package 29
- 3.4 Conclusion 29

4. Background 30
- 4.1 Axiomatic Design 30
- 4.2 Existing Self-Driving Vehicles 33
- 4.3 Sensors for Autonomous Drive 34
- 4.4 Competition 35
- 4.5 Web-Based Management 36
- 4.6 Raspberry Pi 37
  - 4.6.1 Raspberry Pi 3B+ 37
  - 4.6.2 Raspberry Pi 4 38
- 4.7 Python 38
- 4.8 OpenCV 39
- 4.9 Elegoo Mega 2560 39
- 4.10 C++ 39

5. Axiomatic Design 40
- 5.1 Functional Requirements (FRs) 40

5.2 Design Parameters (DPs) 44
5.3 Design Matrix 46
5.4 Conclusion 47

6. Methods 49
6.1 A term 49
6.2 B term 50
6.3 C term 52
6.4 D term 53
6.5 Conclusion 55

7. Sensor Implementation 56
7.1 Elegoo Mega 2560 56
7.2 Design of Sensor Package 56
7.3 Temperature Sensor 58
7.4 Inertial Measuring Unit 61
7.5 Hall Effect Sensor 65
7.6 Ultrasonic Sensor 68
7.7 Battery Monitoring 72
7.8 Sensor Selection 73
7.9 Sensor Layout 73
7.10 Sensors Integration and Block Diagram 77
7.11 Conclusion 78

8. Autonomous Driving Implementation 80
8.1 Components Used 80
8.2 Autonomous Driving Overview 80
8.3 Servo Motor and Brushed/Brushless Motor Control 83
8.4 Video Capture 84
8.5 Image Processing 86
8.6 Lane Detection 88
8.7 Lane Following 91
8.8 Sensor Reading in Raspberry Pi 93
8.9 Ultrasonic Image Calibration 94
8.10 Obstacle Avoidance 96
8.11 Conclusion 99

9. Dashboard Implementation 100
9.1 AI Design 100
9.2 MPAD Web Application 101
9.2.1 Technology 101
9.2.2 Functionality 102
9.2.3 Vehicle Controls 102

9.2.4 Color Selection Tool 104
9.3 Sensor Module Integration 106
9.4 Application Architecture 108
9.5 Startup Scripts 110
9.6 Conclusion 111

10. Technical Testing 113
10.1 Inertial Measuring Unit (IMU) Accuracy Testing 113
10.2 Ultrasonic Sensor Accuracy Testing 117
10.3 Hall Effect Sensor Accuracy Testing 119
10.4 Temperature Sensor Testing 120
10.5 Project Implementation 121
10.6 User Guide 122
10.7 Autonomous Driving Results 122
10.7.1 RC Car for Testing 123
10.7.2 Lane Following 125
10.8 Obstacle Avoidance 129
10.9 Dashboard Sensor Integration 130

11. Financial Analysis 133
11.1 Fixed Costs 133
11.2 Variable Costs 139
11.3 Financial Analysis Results 140
11.4 Conclusion 145

12. Discussion 146
12.1 MPAD Guides 146
12.2 MPAD Dashboard 147
12.3 MPAD Modularity 148

13. Conclusion 149
13.1 Future Work 150
13.1.1 Lidar/Radar Replacing Ultrasonics 150
13.1.2 Sensors Alerts 150
13.1.3 Cloud Computing 151
13.1.4 5G 151
13.1.5 Local Hotspot 151
13.1.6 Algorithm Switching 152
13.1.7 Scalable Server 152
13.2 Reflections 152
13.2.1 Enzo Giglio de Azevedo 152
13.2.2 Antonio Jeanlys 154
13.2.3 Chris Mercer 155

13.2.4 Julien Mugabo 155
13.2.5 Eric Reardon 156
13.2.6 Taylan Sel 157

14. References 159

15. Appendix 162
Appendix A: IMU Accuracy Testing Appendix 162
Appendix B: Sensor and AI Setup Guide 165
Appendix C: Arduino IDE Setup Guide 165
Appendix D: MPAD Gantt Chart 166
Appendix E: MPAD Code Repository 166
Appendix F: MPAD WPI CS Department Poster 166
Appendix G: MPAD WPI CS Department Video 166
Appendix H: MPAD WPI ECE Department Video 166
Appendix I: MPAD WPI ECE Project Presentation 166
Appendix J: MPAD WPI IE Department Elevator Pitch 166
Appendix K: MPAD WPI IE Department Poster 167
Appendix L: MPAD WPI IE Department Video 167
Appendix M: IRB 167
Appendix N: Average Loop Times for Self Driving Algorithms 167
Appendix O: Ray Tracing Algorithm 168
Appendix P: Dashboard Sensor Data Live Updates Video 168
Appendix Q: Dashboard Sensor Color Selection Tool Video 168

List of Figures

Figure 2.1: Pictures of Track 6 from (Kim et al., 2020)[6]	18
Figure 2.2: More Pictures of Track 6 from (Kim et al., 2020)[6]	19
Figure 2.3: Schematic of Track 6 from (Kim et al., 2020)[6]	19
Figure 2.4: (Kim et al., 2020)[6] Self-Driving Kit System Diagram	21
Figure 2.5: Self-driving and Sensor Kit Mounted on RC Car	22
Figure 5.1: FR0 and its children FRs	39
Figure 5.2: FR1 and its hierarchical decomposition	40
Figure 5.3: FR2 and its hierarchical decomposition	41
Figure 5.4: FR3 and its hierarchical decomposition	42
Figure 5.5: FR4 and its hierarchical decomposition	43
Figure 5.6: FR-DP relations of FR0 and its children	43
Figure 5.7: FR-DP relations of FR1 and its decomposition	44
Figure 5.8: FR-DP relations of FR2 and its children	44
Figure 5.9: FR-DP relations of FR3 and its children	45
Figure 5.10: FR-DP relations of FR4 and its children	45
Figure 5.11: Top Level Design Matrix of MPAD Project	46
Figure 7.1: Image of DHT11 Temperature Sensor	58
Figure 7.2: Temperature Sensor Setup Schematic	59
Figure 7.3: Image of BNO055 IMU sensor	61
Figure 7.4: IMU Sensor Wiring	62
Figure 7.5: IMU Pins and Corresponding Wires	62
Figure 7.6: Image of the KY-003 Hall Effect Sensor	65
Figure 7.7: Hall Effect Sensor Wiring	66
Figure 7.8: Image of the HC-SR04 Ultrasonic Sensor	68
Figure 7.9: Ultrasonic Sensor Setup Schematic Example	69
Figure 7.10: Image of the BX100 Low Voltage Buzzer Alarm	71
Figure 7.11: Sensor Layout in an RC car	74
Figure 7.12: The CAD Model of the Ultrasonic Sensors Bumper	75
Figure 7.13: System Block Diagram for MPAD	77
Figure 8.1: The flow diagram for lane following	80

Figure 8.2: Sketch of Steering Angles	83
Figure 8.3: A Frame from Raspberry Pi Camera Capture	84
Figure 8.4: Coordinates of a Frame	85
Figure 8.5: Image Cropping Visualized	86
Figure 8.6: Color Filter Results of a Single Lane	88
Figure 8.7: Lane Detection Flow Diagram	90
Figure 8.8: Steering Angle with Hard Boundaries	92
Figure 8.9: Sketch of Ultrasonic Sensor Pickup Cones and Calibration Test	95
Figure 9.1: Driving dashboard on MPAD's web application	101
Figure 9.2: Control panel on web dashboard, annotated for each feature	102
Figure 9.3: Lane color selection tool on the web dashboard	104
Figure 9.4: Abstracted system block diagram	106
Figure 9.5: Web application system architecture	107
Figure 10.1: IMU Orientation for testing	113
Figure 10.2: X-axis Orientation Error for the IMU	114
Figure 10.3: Y-axis Orientation Error of the IMU	115
Figure 10.4: Z-axis Orientation Error of the IMU	116
Figure 10.5: RPM Diagram of Hall Effect Configuration	118
Figure 10.6: PCB board sensor Layout	120
Figure 10.7: Turning Radius Calculations	122
Figure 10.8: Birdseye Image of RC Car	123
Figure 10.9: Sideways View of RC Car	124
Figure 10.10: Track in Higgins	126
Figure 10.11: Car in Track	127
Figure 10.12: Obstacle Avoidance Testing	129
Figure 10.13: 1) CPU Temp reading 2) Ultrasonic readings. Each ultrasonic is angled in a different direction. 3) Ambient temperature 4) Hall Effect 5) IMU data, with a line for each X, Y, and Z axis. 6) Humidity	131
Figure 11.1: First Part of ARENA Model for Financial Analysis	135
Figure 11.2: Second Part of ARENA Model for Financial Analysis	136
Figure 11.3: Break Even Chart	143
Figure 15.1: Ray tracing visual GUI screenshot	167

List of Tables

Table 10.1: Percentage Error of the IMU	113
Table 10.2: Difference in Degrees of the IMU	113
Table 10.3: Ultrasonic Sensor Accuracy Testing Readings	117
Table 10.4: Accuracy Testing Data for Hall-Effect Sensor	119
Table 10.5: Sample sensor readings from a serial string	130
Table 11.1: Warehouse Choice Costs and Features Table	133
Table 11.2: Bill of Materials of MPAD	139
Table 11.3: Sensitivity Analysis	142
Table 11.4: Time Value of Money Analysis Table	142
Table 15.1: Expected IMU X, Y, and Z Values	162
Table 15.2: Calculated IMU X, Y, and Z Values	163
Table 15.3: Percentage of IMU X, Y, and Z Values	164
Table 15.4: Loop Time for Different Driving Algorithms	166

Abstract

There have been major advances in autonomous vehicles through recent years. Features such as adaptive cruise control, emergency braking, lane assistance, and automatic parking have become standard in most new vehicles. There exist cars nearing level 5 autonomy where the vehicle can operate without a driver and under any condition. As self-driving technology cements itself in our society, this package turns a normally trivial task into a display of modern technology. Developing a car with these capabilities requires computer vision and a variety of sensors to recognize its surroundings and navigate obstacles[1].

Due to their size and scope, Radio Controlled (RC) cars are a great way for prospective engineers to learn real-world technical skills. The low cost and turnaround time allows for rapid development and testing, which can effectively teach many of the same principles as real cars. Autonomous driving is the most ideal way to test a custom RC car's mechanical integrity to avoid human driving bias. Current self driving RC car modules, such as Donkey Car[2], SunFounder PiCar[3] and Deep Pi Car[4], require hours of setup configurations and course-specific training before driving autonomously. Even after training, most modules do not provide the capability to avoid obstacles or traverse non-standard terrain.

An ideal self-driving module must be capable of being downloaded and installed onto an RC car regardless of dimensions. Since this project's audience is typically Mechanical Engineers with little computer science and electrical engineering experience, having controls and driving data be easily accessible in a website would prove to be extremely beneficial. Hence, we developed the Modular Package for Autonomous Driving (MPAD). This project will demonstrate how techniques in computer vision and sensors are used to create a truly modular

package. MPAD allows Mechanical Engineering students to test their work in a complete system, without requiring a coding or electrical design background.

MPAD began with an axiomatic design of the project, to lay out the steps for design and guarantee an efficient final product. The development boards used were the Raspberry Pi 4 8GB and Elegoo Mega 2560. The sensors are connected to the Elegoo, which then connects to the Raspberry Pi. The sensor package uses a Raspberry Pi camera for lane recognition and seven ultrasonic sensors for obstacle avoidance. Along with the ultrasonic sensors, the final sensor package includes an inertial measuring unit (IMU), two temperature sensors, an hall-effect sensor, and a battery sensing circuit. All of the sensors went through rigorous accuracy testing. The package is specifically designed to be easily transferable from one RC car to another. The self driving software uses computer vision to navigate a course without any prior training. The angles of the wheels are calculated using the coordinates of the pixels in the camera input along with the readings from the ultrasonic sensors. Testing revealed that autonomous driving was able to do U-turns as well as perform better at high speeds. While driving alone is visually interesting, engineers need something concrete: data. Everything the car is reading can be seen on a web portal in real time. Through customizable sensor graphs and camera views, the car can be highly configured and tested. If the students choose not to touch the program, they can rest assured knowing the web controller will quickly and reliably connect them to their car across any number of devices. As a result, we have designed a modular self driving package so Mechanical Engineers can bring their designs to life. MPAD was implemented in the Introduction to Engineering Design (ME2300) course at Worcester Polytechnic Institute (WPI). Eight teams of mechanical engineering students each received documentation in the form of two guides to create their own version of MPAD. The integration was successful and the students were able to

fully utilize MPADs capabilities with their own RC car design.The paper also discusses the implementation, testing details, and future work for a self driving modular package.

Acknowledgements

The authors of this paper would like to thank the following individuals for their contributions and assistance throughout the duration of this project:

Professor Pradeep Radhakrishnan, WPI Department of Mechanical Engineering
Professor Kaveh Pahlavan, WPI Department of Electrical and Computer Engineering
Professor Walter T. Towner, WPI Department of Industrial Engineering
Ms. Barbara Furhman, Administrative Assistant, WPI Department of Mechanical Engineering
Yash Yadati, Teaching Assistant, ME 2300 Introduction to Engineering Design Course

2020-2021 PARV: A 3D Printed Robotic Autonomous Vehicle Platform, ME Team

Dexter Czuba
Rajkumar Dandekar
Steven Gordon
Eric Stultz

2019-2020 Self Driving RC Car Team

Executive Summary

Here you will find videos, powerpoints, and posters that summarize all the work done for the Modular Package for Autonomous Driving:

https://wpi0-my.sharepoint.com/:f:/g/personal/egdeazevedo_wpi_edu/EnLFi-fWZNlOk5psnYriRxoBCQPIoQPGM0eMRH50SBrIDA?e=aCk2IA

Authorship Table

Section	**Primary Author**	**Editor**
Abstract	All	All
Executive Summary	All	All
Introduction	Enzo, Julien, Chris	Enzo, Julien, Chris, Eric, Antonio
Previous MQP	Enzo, Taylan	Taylan, Enzo, Eric
Goals and Requirements	Chris, Taylan, Julien, Enzo	Chris, Taylan, Julien, Enzo, Eric
Background	Enzo, Taylan, Julien, Chris, Eric	All
Methods	Eric, Taylan, Julien, Enzo, Chris	Eric, Taylan ,Julien, Enzo, Chris
Axiomatic Design	Enzo	Enzo
Sensor Implementation	Enzo, Antonio	Enzo, Antonio, Julien
Autonomous Driving Implementation	Taylan	Taylan, Chris, Eric
Dashboard Implementation	Chris, Eric	Chris, Eric
Technical Testing	Taylan, Enzo, Julien, Antonio, Chris	Taylan, Enzo, Julien, Antonio
Financial Analysis	Enzo	Enzo
Discussion	Enzo, Chris, Eric, Taylan	Enzo, Chris, Eric, Taylan, Antonio
Conclusion	All	All
References	Antonio, Julien	Antonio, Julien
Appendix	Enzo, Chris, Taylan, Antonio	Enzo, Chris, Taylan, Antonio

1. Introduction

Autonomous vehicles have become more advanced in recent years, generating more interest and discussion to the topic. In many scenarios, autonomous vehicles on the road could decrease accidents and improve traffic. However, there are also ethical concerns associated with autonomously driving vehicles when it comes to making critical decisions. Still, more and more companies are joining the race to full autonomy. Although the closest purchasable fully autonomous vehicles are those with adaptive cruise control (ACC), there are now driverless, fully autonomous commercial vehicles on the roads in the United States. Waymo, a subsidiary of Google entirely focused on self-driving, recently released a driverless ride-hailing service in Phoenix, named Waymo One[5]. This service is fully available to the public, but is not yet ready for denser city driving with a larger population and more obstacles.

With an ACC system, the car provides driving assistance by automatically adjusting the car's speed and direction to maintain the distance between the cars ahead. This is one such example of autonomous driving gradually creeping into the public eye. As this functionality becomes less of a dream and more of a reality, it makes sense to familiarize the coming generations of students with the technology. However, the complexities of a full size car and the algorithms behind it make this difficult to teach at a college level. By scaling down to an RC car and pre-making the self driving module, students can make their own car without a background in electrical engineering or computer science.

This project's intent was to build a fully autonomous Radio-controlled (RC) car that has similar technology as autonomous vehicles. Most autonomous cars are equipped with sensors in order to detect objects and avoid obstacles. This is done with the aid of computer software that takes the physical readings of the car's surroundings and makes sense of not only where the car

currently is, but where it should be moving towards. The more sensor readings available to the car's software, the more informed the Artificial Intelligence (AI)'s decisions can be. Sensors can also give useful data on the car's performance. The current sensors implemented on our RC car are camera, ultrasonic sensor, IMU, temperature sensor and hall effect sensor. All these sensors provide data that is displayed on a dashboard and can be exported for further analysis. The sensors are placed in specific areas of the car. For example, the ultrasonic sensors are placed on the bumper to assist the camera in obstacle avoidance and the temperature sensors are placed in areas with high temperatures such as the motor and battery.

The deliverable was a self driving modular package that can be implemented on other RC cars. To test this functionality, our modular system is being used in the WPI Engineering Design course (ME 2300). The ME students are tasked with implementing this modular package on their designed RC cars. They are provided with a sensor and AI guide that is easy to follow and requires no prior knowledge on sensors or AI programming. This emphasis on ease of use ensures that students are spending time developing their car rather than worrying about configuring sensors or software. This was a guiding principle throughout the project in order to make sure the module could be used by students with any skill level. As it stands, there are not many courses at WPI that let students explore the benefits of integrating their designs with software. The intent was to provide a means of teaching how electrical engineering and computer science principles can be integrated with mechanical engineering to deliver a more complete final product.

1.1 Report Organization

The following report was organized into multiple chapters, each discussing a specific aspect of the project or a specific component of MPAD. Chapter 2 goes over the previous MQP

of the academic year 2019-2020, while Chapter 3 talks about the goals and requirements the team decided upon for this project. Chapter 4 presents background information and concepts that were used by the team when creating MPAD. Chapter 5 has the Axiomatic Design made for the MPAD product, while Chapter 6 goes over the methods and timeline used by the team members. Chapters 7 through 9 have the documentation of the sensors, autonomous driving, and dashboard implementations. Chapter 10 goes over all the testing done for the multiple parts of the project. Chapter 11 discussed the viability of making a business out of MPAD in the form of a financial analysis, while Chapter 12 discussed the integration of MPAD into the Introduction to Engineering Design (ME2300) course at WPI. Chapter 13 is the conclusion to the report and has the team's future work recommendations and reflections of the project. Chapter 14 contains the references list of the report and Chapter 15 has the appendices.

2. Previous MQP (2019-2020)

In this chapter, the team goes over the (Kim et al., 2020)'s work[6]. The chapter covers the team's main goals, concepts utilized, implementation data, testing results, and final deliverables. Alongside that, the chapter contains our experience of implementing the previous MQP and some of our impressions over their work.

2.1 Project Summary

The team's main goal was to develop a modular self-driving and sensors kit that can be integrated into any scale or remote-controlled car as a way to provide students in mechanical engineering exposure to autonomous driving and real-time data collection (Kim et al., 2020)[6]. At Worcester Polytechnic Institute, there is a senior-level capstone design course called Advanced Engineering Design where student teams design and develop a scale-car with a custom-gearbox and a steering linkage. The teams must test their car at the end of the seven-week course. While the mechanical design of the system may be robust, the sensors and control are all manual. Given that the students may not have enough practice driving scale-cars, the end-of-term race might not fully evaluate the car's ability to tackle varied driving conditions.

To develop a modular self-driving kit, the team first had to select a commercially available self-driving platform for us to use, and the platform selected was DonkeyCar[7]. Next, the team had to test this self-driving platform. The first goal with the self-driving package was to have a radio controlled car drive four laps around a track without human intervention using the self-driving package, and to achieve this goal, the team tested the self-diving package on seven different tracks. Figures 2.1 through 2.3 show an example track that the previous MQP team utilized to perform their testing.

Figure 2.1: Pictures of Track 6 from (Kim et al., 2020)[6]

Figure 2.2: More Pictures of Track 6 from (Kim et al., 2020)[6]

Figure 2.3: Schematic of Track 6 from (Kim et al., 2020)[6]

First, the self driving package was tested on a recommended car on simple tracks. Next, it was tested on a student-made car on simple tracks. Next, the self-driving platform was tested on a 3D printed car on simple tracks. Next, the self-driving platform on a student-made car on complex tracks. Lastly, the self-driving platform on a 3D printed car on complex tracks. With each test, the team gained experience creating neural network models and learned techniques on how to make a better neural network model, which in turn improved the self-driving performance of the remote controlled cars the team tested on until they were able to achieve their goal of a remote controlled car self-driving four laps around a track.

After running tests on the self-driving platform, the team needed to create a user guide that is easy for students in ME 4320 and ME/RBE 4322 to follow so that they can implement their self-driving kit on a scale car to create a self-driving scale car. They created a first draft of the user guide by buying all of the components necessary to make a second self-driving kit and taking notes as they constructed the second self-driving kit. The team formalized these notes to make their user guide. Next, they tested their user guide on two Mechanical Engineering students, Jesse Kablik and Brian King, so the team could evaluate their guide. They used the feedback that they gave us to improve the guide, completing their development of the modular self-driving kit and user guide. The kit is now ready to be deployed in ME 4320 and ME/RBE 4322.

The team had to make a sensor package and a user-guide on how to use the package. As per the objective on the modular sensor kit, they needed to determine what sensors should be used for data collection, as well as how they should go about implementing the sensors in an easy to understand and follow manner. This was accomplished with an Arduino Mega[8] and

RasPi[9]; both fairly common and simple to use electronics. For the sensor package, the team wanted to provide students with various sensors that could record useful data about the function of their scale cars. To do this, they had to wire various sensors into an Arduino, and then program that Arduino to send the data to a Raspberry Pi where the data could be saved to a useful filetype for later analysis. Figure 2.4 showcases the previous team's system block diagram of their self-driving kit.

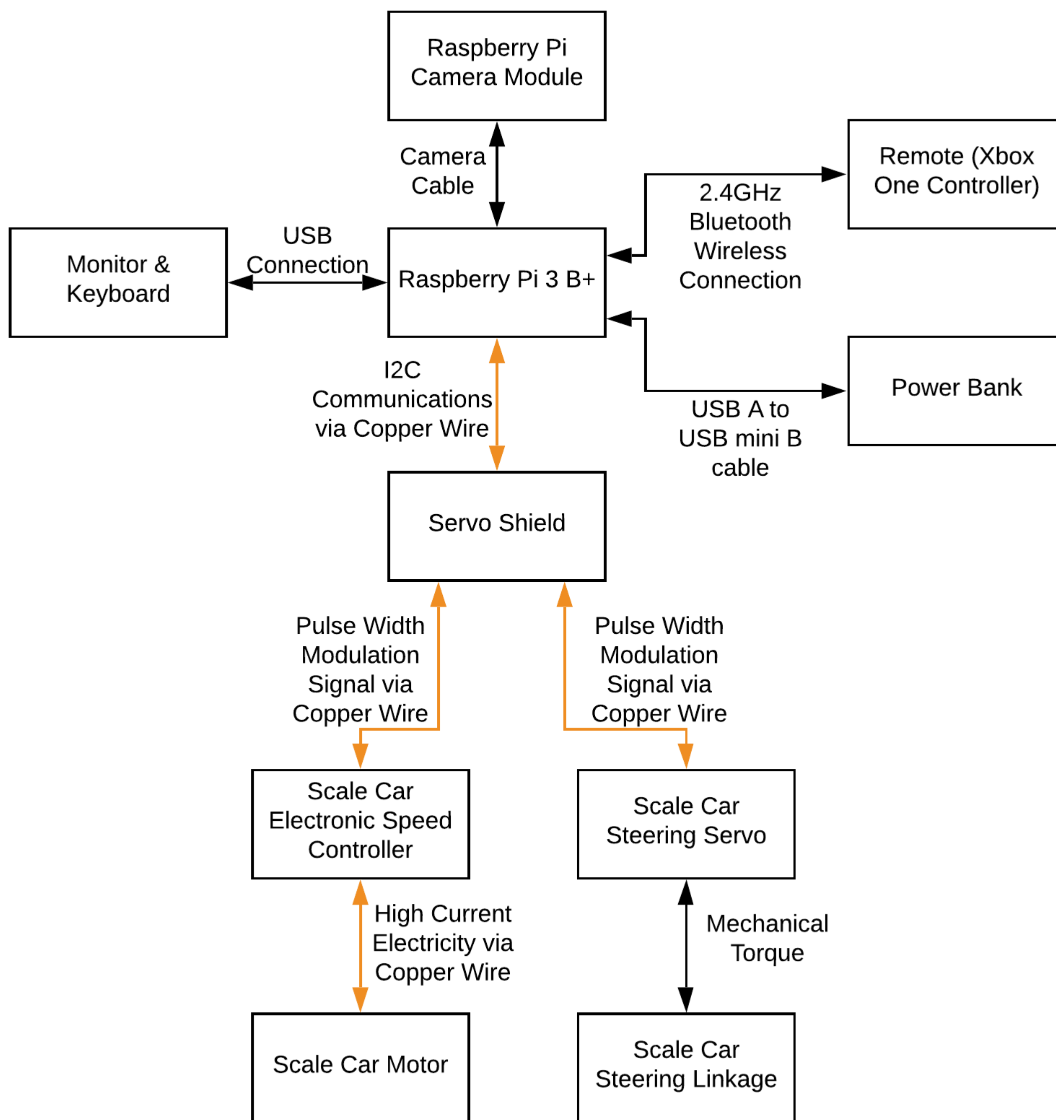


Figure 2.4: (Kim et al., 2020)[6] Self-Driving Kit System Diagram

The authors also developed a modular sensor kit that could be integrated into any scale-car to record speed of drive shafts, rotations around pitch and roll axes, and temperatures of motors, batteries and sensor boards. The sensor kit records data as the car is being driven around the race track. The sensor kit and the collected data can provide students exposure to real-time data collection and enhance their learning and understanding of the behavior of mechatronic systems, as well as serve for analysis of their cars performance. As with the self-driving package, the sensor kit was also tested on the same two Mechanical Engineering students, Jesse Kablik and Brian King, so they could evaluate their guide. With their limited experience in using an Arduino and Raspberry Pi, they provided valuable feedback for this second user guide, and like the self-driving user guide, this one is also ready to be deployed in ME 4320 and ME/RBE 4322. Figure 2.5 shows the previous MQP self-driving and sensor kit and how it looked on top of a RC car.

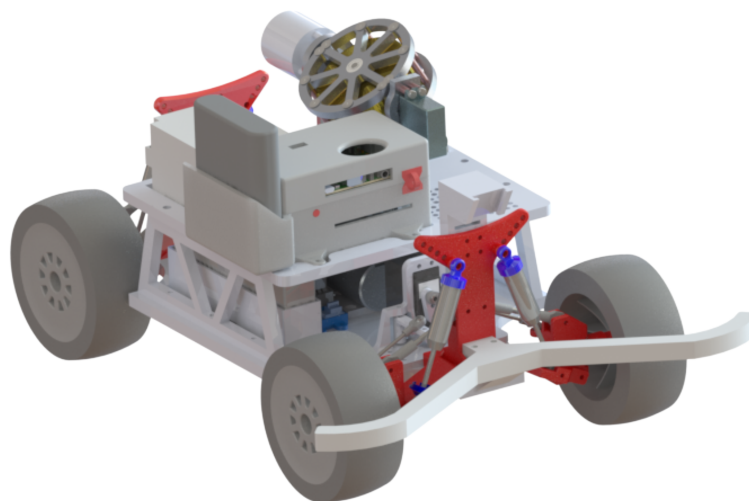


Figure 2.5: Self-driving and Sensor Kit Mounted on RC Car

2.2 Moving Forward

At the very beginning of the project, the advisor made it clear to us that he wished for the project to be used in one of his classes. Last year's project was meant to be used in his class as well, but due to Covid 19 and the difficulty in implementing the project the only two ME volunteers ended up testing the project as outsiders. Going forward modularity and ease of use was a major factor to keep in mind.

The first 7 weeks of the project was focused on replication of last year's project and outlining major design decisions. Last year's MQP did not make their own self-driving kit. Instead, they used DonkeyCar, a popular DIY self driving kit made for RC cars. The relevant steps outlined in the DonkeyCar setup guide are:

1. Install the Software
    a. Install Software on Host PC
    b. Install Software on DonkeyCar
2. Create DonkeyCar APP
    a. Create Donkeycar from Template
    b. Configure Options
    c. Joystick setup
    d. Camera Setup
    e. Keeping Things Up To Date
3. Calibrate steering and throttle
    a. How to adjust your car’s settings
    b. Steering Calibration

   c. Throttle Calibration

4. Get Driving.
   a. Start your car
   b. Driving with physical Joystick Controller
      i. Setup Bluetooth and pair joystick
      ii. Start Car
      iii. Joystick Controls
5. Train an autopilot
   a. Collect Data
   b. Transfer data from your car to your computer
   c. Train a model
   d. Copy model back to car

This is the shortened version cutting out parts of the guide that we found were not relevant. The naming and the structure are verbatim. The guide does not cover setting up a Raspberry Pi from scratch which requires a monitor, HDMI cable, a mouse, and a keyboard as well as an SD card to store the OS and files.

The majority of the time, interaction with the Raspberry Pi and the DonkeyCar software were through the Windows command line interpreter (CMD) and the Linux Terminal. Someone who might not be familiar with these tools would have trouble following the guide.

Donkey Car uses Machine Learning in order to drive. Given it does not call upon a previously established model, the user has to create their own. Creating your own model is a lengthy and complicated process which requires starting over if the model is flawed. Collecting

the training data requires manually driving the car around for 10-20 laps. The driving skill of the model will be dependent on the driving skill of the user. The driving data then has to be transferred to the user's personal computer where the training can take hours. The end product is track specific and cannot stay in the lane if changes are made to the track.

Avoiding the failings of DonkeyCar became the main goal for MPAD. The implementation of the package needed to be fast and easy. The driving itself needed to be training free and universal. Interfacing with the car needed to be intuitive.

2.3 Conclusion

This chapter provided an overview of what the previous MQP team attempted to do and some of the main characteristics of their project. It went over the concepts applied, the training performed, and the final product the team delivered. This chapter also covers how the previous MQP set up DonkeyCar and how our team wanted to go for a different approach for our project. The next chapter will go over the goals and requirements our team had in mind while creating MPAD.

3. Goals and Requirements

The main goal of this MQP is to create a modular package for autonomous driving in RC cars. For use in commercial or educational settings, modularity and ease of use are incredibly important. A package that needs hours of setup is impractical. Especially when it comes to driving, requiring the user to record manually driven laps and to train their own machine learning model is complicated for anyone besides a small subset of the population. The driving itself should be able to perform the basic functions of a human driver, such as following lanes, avoid obstacles, break, and go up and down ramps. As an educational tool MPAD also needs to be able to be controlled remotely as well as provide real time data in an easily digestible format.

3.1 Lane Following

Since there are many different ways to drive a car through a given course, it initially seems difficult to define what 'good' driving looks like. However, when broken down into specific characteristics it becomes less hazy:

- Throttle: At a minimum, the car should be capable of driving at a similar speed to that of a human driver. This includes the throttle speed at a given track location, such as how fast a sharp corner can be taken.
- Steering tightness: The car must not swerve unnecessarily. When looking at a given section of track, the car should be able to approximately take the shortest path to reach the end of the section.
- Accuracy: This trait is more clearly defined, as the accuracy is considered as how often the car stays within the boundaries of the track whilst still moving toward the end of the lap. This must hold for any course that is labeled with clear boundaries and obstacles.

The algorithm must adhere to these specifications to be considered complete. Once these conditions are met, the car is considered more and more capable as the throttle speed can be increased while still driving accurately.

3.2 Web Dashboard

The dashboard must provide an intuitive means of interfacing with the car. Without any interaction from the user during startup, the server must serve dashboard pages in a predictable manner. Once loaded, these web pages must contain the following details:

- Sensor data displayed in well labeled graphs within 500ms of the reading being taken on the car's sensor array. Readings are to be polled every 1000ms or less.
- A live camera feed of at least 10 frames/second can be seen on the page, with a latency no greater than 500ms.
- The user must be able to click the camera's live feed to be able to choose which lane colors are to be used. Users may click as many times as they want, with each click adding the given selected pixel to the car's lane color range.
- The dashboard must be able to terminate the Raspberry Pi to allow for a safe shutdown.
- The dashboard should allow the following controls
    - Start self-driving algorithm
    - Stop self-driving algorithm
    - Alter throttle speed of self-driving algorithm
    - Alter a vehicle's steering aggressiveness during driving
    - Alter direction of car's driving motor to align with gearbox placement

3.3 Sensor Package

The sensor package is a vital part of both the driving and the user experience. Obstacle avoidance cannot be done with the camera alone, sensors to detect and find the distance to obstacles are needed. As an educational tool it is important for MPAD to provide useful information to the students.

- Sensors must display real time data on the dashboard
- The overall sensors package needs to be modular and can be implemented on other RC cars.
- Sensors must be able to detect obstacles on the track.
- Each sensor data output needs to be accurate.
- The sensors must provide the following information on the condition of the car:
  - Temperature of the motor and battery
  - Acceleration and Orientation
  - RPM
  - Distance between the RC car and obstacle object

3.4 Conclusion

The goals and requirements chapter was important to keep the whole team working towards the same end product. With the goals and requirements all laid out, the team was able to move forward with the project and work on the following steps. The next step was to research and develop a background on the possible systems and concepts that would allow the creation of the MPAD. The next chapter will cover all the background gathered to start implementing the project.

4. Background

The following chapter provides a background on various systems and concepts that were applied in the development of MPAD. Using the recommendations from last year's team (Kim et al., 2020)[6] and the goals and requirements chapter of this paper, the team researched relevant topics and important information that allowed us to create MPAD. This background chapter will go over the Axiomatic Design, Existing Self-Driving Vehicles and their sensors and lastly information on the softwares and boards that were used.

4.1 Axiomatic Design

One of the main goals of any engineering project is to create an efficient system design. However, before the 1990s, there had not been any proven way to measure the quantitative efficiency of a system design. Some methods that people tried to implement were the Heuristic and empirical methods. These system design methods have comprehensible downfalls, such as requiring constructed models of the design or basing themselves on qualitative guidelines, such as making the design as simple as possible or asking the five whys. These have also proven to be extremely expensive for companies that want to physically create models to test them and find out that something in the product went wrong. Some other design methods include decision theory or dimensional analysis. However, these do not provide good rational tools for design and sometimes only validate the acquired results. Another issue encountered by these methods is that non-physical systems such as software would not support it.

A system with multiple functional requirements makes it hard to satisfy all customer requirements, given that lower functional requirements and design parameters can affect higher requirements that define the system. The solution for having an efficient system is using

axiomatic design. The advantage of using the axiomatic design is that it allows reducing the complexity of the design by breaking it into multiple levels and allowing the engineer to see the right design decisions. It is also a design approach that works in any system, machine, or software. There will always be the need to satisfy functional requirements, constraints, and the Independence and Information Axioms. However, for each of these designs, the relationships between functional requirements and their specific design parameters might differ.

The first step for creating an axiomatic design is defining what the customer wants. The customer needs are the same as the attributes of the final design. Functional requirements and constraints are then created to satisfy the customer's needs.

Suh has fully programmed to-fledged the minimum set of independent requirements that the design must satisfy, while the constraints are limitations encountered when creating the system. Alongside each functional requirement, there is a specific and unique design parameter. A design parameter is the design embodiment that corresponds to a functional requirement. They can vary greatly depending on the system design. They may be physical parameters, for example, parts or components, or in the case of software, they can be code modules or programs. The next step of an axiomatic design would be to define design parameters corresponding to the functional requirements previously defined.

After matching the functional requirements with the design parameters, the next step to create an axiomatic design is to map the pairs and create a hierarchical decomposition. These hierarchical decompositions will have the most important and broad functional requirements on higher levels, while the lower levels will contain functional requirements that branch out of their higher requirements. The relationship between a higher-level functional requirement and a lower level one can also be defined as a parent and child relation. So, if a top-level functional

requirement has three lower-level ones below it, it is common to say that the higher-level functional requirement has three children, or that the three children have the same parent. When creating these relationships, a zigzagging process will go through all the higher-level functional requirements and design parameters before going down to the lower levels. This process establishes an appropriate architecture for the design and helps to create a design matrix so that the system is uncoupled or decoupled.

During the mapping phase, it is also vital to keep in mind that there is a need to satisfy both the Independence and Information Axioms. The Independence Axiom states that "There is a need to maintain the independence of the functional elements of a design." The Information Axiom states that "Among all the designs that satisfy the independence axiom the one that possesses the least information is the best"(Suh, 1998)[10]. These two axioms make the basic concept of all axiomatic designs. While creating an axiomatic design, these are the two axioms that must remain true for the best possible design. Another way to put both designs into words is that the functional requirements must be mutually exclusive and collectively exhaustive to follow both the Independence and Information axioms.

The axiomatic design uses linear algebra, where a functional requirement vector is equal to the product of a design matrix A with the corresponding design parameters vector. This relationship is present at any level of the hierarchy design. While creating the design matrix, there will be a correlation of which design parameters affect each functional requirement. An X can is used whenever a design parameter affects a functional requirement, while if there is no correlation, zero is inputted. It is also important to note that any correlation made in the lower hierarchical levels of the design matrix should reflect towards the highest level.

To satisfy the Independence Axiom, the system needs to have an uncoupled or decoupled design. Whenever each of the functional requirements can be satisfied independently by a single design parameter, the design matrix A will be seen as a diagonal matrix and will represent a fully uncoupled design (Suh, 1998)[10]. Whenever the independence of the functional requirements is only guaranteed by a certain sequence of design parameters, the design matrix A will assume a triangular design. A triangular design is a sign of a decoupled design. Any other designs would be called a coupled design and would violate the first axiom of independence. In most large system designs, it will be hard to achieve an uncoupled design matrix. So most designs usually aim to achieve a decoupled design with a triangular matrix. Convention-wise, a triangular matrix would generally be a lower triangular matrix, where all elements above the diagonal are zero. However, there is no problem in having an upper triangular matrix, where all elements below the diagonal are zero.

4.2 Existing Self-Driving Vehicles

Initially, an RC car seems out of place in a research lab. However, RC cars serve a deeper purpose than just entertainment. Full size cars and RC cars face the same challenges when considering the design of a self-driving algorithm. The underlying technology that makes vehicles autonomous is mostly the same, be it on a small RC car or a full size SUV. Because of this overlap, research into self-driving car companies like Waymo and Comma.ai is useful to understand the current state of autonomous technology[11]. The most popular implementations rely on a mixture of camera and LIDAR input to map the surroundings of the car. Once the car's location is mapped, potential paths are calculated before choosing the most effective route. Lastly, steering and acceleration/braking data is calculated to match the given route. This

multi-step process is repeated in a loop to allow the car to quickly respond to not only the road but obstacles moving relative to the car.

A separate approach uses a single camera but forgoes the LIDAR sensor. Without this 3D mapping, the on board GPU must estimate the 3D surroundings by using computer vision techniques. This is more challenging, but has the benefit of being easier to install and manufacture.

4.3 Sensors for Autonomous Drive

Having reliable vision capabilities is a must in any autonomous vehicle. A combination of sensors allows the car to recognize and interpret what is occurring around itself. The key to having an optimal vision method is using different sensors and other equipment that allow for the car to verify if the reading of one sensor is accurate. Small autonomous vehicles can use lightweight sensors; however, these sensors are subject to constraints such as processing and power. Inertial sensors, acoustic sensors, compasses, and cameras are options to determine estimates of the vehicle's orientation and acceleration. Given the previously mentioned constraints (most notably processing), choosing which sensors that are beneficial in different scenarios is a complex task. The ability of a system to determine its orientation solely based on its sensors is called observability. Observability is affected by a variety of factors. One of the main factors that contribute to observability is the sensor sampling rate[12]. Another factor that also influences the observability is the control choice and trajectory. Commonly, a vehicle will perform a similar or cyclical path of choice. Choosing the appropriate sensors to record these data patterns is an essential part of any car's design.

In the case of smaller vehicles, the three primary technologies used in autonomous RC cars are cameras, radars, and lidars. The camera is mainly responsible for identifying objects and

keeping the car in the correct lane. It is also crucial for features such as identifying obstacles and driverless parallel parking. The radar can pinpoint objects and their distances from the RC car. The Lidar serves almost the same function as the radar, but it also allows for 3D modeling of the surroundings. Together these three sensors can provide the RC car with the precise location of objects in its surrounding.

Additionally, other sensors can also help the car self-drive. Some of the typical auxiliary sensors are Global Positioning Systems (GPSs), ultrasonic sensors, and IMUs. GPSs are good at helping the car decide the fastest path to go from one location to another. It also helps provide the exact location of the vehicle. Ultrasonic sensors can serve as a way to verify the distance of objects from the car. Ultrasonic sensors tend to be cheaper than both Lidar and Radar, but they are also less precise and have a smaller coverage. IMUs show positioning and orientation in space. They generally work alongside a GPS or a Hall-effect sensor to provide accurate position data of the car. There are additional sensors that improve autonomous cars like temperature, proximity, and battery sensors[13].

4.4 Competition

The concept of a self-driving or autonomous RC car is not new. For the same reasons, we have chosen an RC car as a platform, others have done the same for some years now. The difference, however, is that almost all of the driving algorithms commercially available rely on machine learning. While the differences will be more intimately explained in the next section, the main difference is that machine learning approaches take hours of training in order to be able to drive on a given course as explained in chapter 2(Last year's MQP). One of the main competitors, Donkey Car[14], is a prime example against our logic-based computations. To train their algorithm, users must remotely log into the Raspberry Pi and initialize the training program.

From there, users record 10-20 laps around the track before transferring the data to their personal computer for the intensive model training. With as many as 20 thousand camera frames in total, this training process can take upwards of 5 hours to finish. When considering that this process must be done for each car configuration and specific course, the benefits of choosing DonkeyCar seem to become less and less worthwhile.

4.5 Web-Based Management

Other competitors had longer set up times than what students would likely find acceptable. However, another aspect of the existing technologies that did not seem approachable to non-computer science students was remote access. In order to start, stop, or otherwise configure the RC car, users had to remotely connect to the Raspberry Pi onboard. This is a fairly straightforward concept for tech-savvy users but proved cumbersome for first-time students. It quickly seemed necessary to devise a method of controlling the RC car without having to directly interface with the computer that controlled the driving algorithm. In order to stream to and from a Raspberry Pi, SocketIO[15] repeatedly comes up as a solid choice. SocketIO enables streaming content of any kind, including video and sensor data. These streams are called 'web sockets' or just 'sockets'. A socket can be instantiated on a web page to give users real-time views of the car's camera and sensor readings. Rather than using a RESTful API[16], the streams are kept alive for the duration of the user's interaction with the car. This means that data can move quickly to provide near-instant updates while still using low resources from the server. Since SocketIO exists as a Python library, it also became apparent that the underlying server framework should be written in Python as well. For our purposes, Flask[17] fits the bill quite well. There are many web framework choices, but Flask is straightforward to use and implement.

Once committed to the project repository, the Flask code can quickly be deployed to a server in order to test changes.

Separate from the server, the client must have code to display the video stream and render the graphs detailing the sensor data. JavaScript is the obvious choice here, with support in every major browser.

4.6 Raspberry Pi

Raspberry Pi (also known as RPi or RasPi) is a brand of single-board computers. It is very popular with hobbyists and students due to its low cost, modularity, and open design. A Raspberry Pi OS [9](previously known as Rasbian) can boot with a Raspberry Pi. The Raspberry Pi OS is a modified version of the Linux distribution Debian. The operating system can be loaded onto an SD card on another computer and inserted into a Raspberry Pi board. This project uses Raspberry Pi 3B+ and Raspberry Pi 4 boards. The Raspberry Pi is a fully-fledged computer. It allows any program to be run in a Linux environment. It is easy to use with a large amount of documentation, it is quite powerful given its size and cost. These factors along with the fact that The Raspberry Pi board comes with many ports (most importantly a camera port) makes it a good fit for our goals. The Raspberry Pi was also used in the 2019-2020 MQP with Donkey Car.

4.6.1 Raspberry Pi 3B+

The Raspberry Pi 3B+ uses BCM2836 as its SoC (main processing chip) and has a 64-bit processor at a 1.4 GHz clock rate[18]. The GPU is a Broadcom Video core-IV, which has 1GB LPDDR2 SDRAM as its RAM. The Raspberry Pi 3B+ storage depends on the capabilities of the Micro-SD card inserted, and the board has 40 GPIO pins.

It also has Gigabit Ethernet, 2.4 GHz, and 5 GHz 802.11b/n/ac Wi-Fi alongside Bluetooth 4.2. The ports are HDMI, 3.5mm analog audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI). The board is 82 mm long, 56 mm wide, and 19.5 mm thick. Figure 4 shows the schematics of the Raspberry Pi 3B+.

4.6.2 Raspberry Pi 4

The Raspberry Pi 4 uses a BCM2711 as its SoC (main processing chip) and has a 64-bit processor at a 1.5 GHz clock rate[19]. The GPU is a Broadcom Video core-VI. It has 1 or 2 or 4 GB LPDDR4 SDRAM as its RAM. Its storage depends on the capabilities of the Micro-SD card inserted. The board has 40 GPIO pins.

It has Gigabit Ethernet, 2.4 GHz, and 5 GHz 802.11b/n/ac Wi-Fi as well as Bluetooth 5.0. The ports are two micro-HDMI 2.0, 3.5 mm analog audio-video jack, two USB 2.0, 2 × USB 3.0, Gigabit Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI). The board is 88 mm long, 58 mm wide, and 19.5 mm thick.

4.7 Python

Python is an interpreted high-level programming language, which is famous for its readability and general purpose[20]. It comes with a comprehensive standard library as well as a vast collection of third-party libraries. It is also garbage-collected, which increases its ease of use. Python comes installed on the Raspberry Pi OS and features many libraries useful to our project. Its ease of use and clarity makes it optimum in group projects as well.

4.8 OpenCV

OpenCV is an open-source computer vision library. Well-known companies such as Google, Microsoft, Intel, and IBM all use OpenCV[21]. While it has C++, Python, Java, and Matlab, our project only uses OpenCV in Python. OpenCV is an extensive library that includes both simple image manipulation tools and advanced machine learning algorithms. This project primarily uses OpenCV to apply a color filter to images and locate the coordinates of pixels of a specific color.

4.9 Elegoo Mega 2560

Elegoo Mega 2560 is an Arduino-IDE compatible board. The chips on the board are Atmega2560-16au and Atmega16u2[8]. It contains 54 I/O pins and 16 analog inputs. It also has 4 UARTs, a 16 MHz crystal oscillator, a USB port, a power jack, an ICSP header, and a reset button. It can be connected to the computer using a USB-B to USB-A serial connection. A large number of I/O pins makes the board optimal for our sensor setup.

4.10 C++

C++ is a general-purpose programming language[22]. C++ has object-oriented, generic, and functional features. C++ can also do low-level memory manipulation, which makes it popular for systems programming and embedded applications. It has many libraries for sensors as well, which makes it a good choice for our project.

5. Axiomatic Design

In this chapter, the team will go over the axiomatic design created for the project. The axiomatic design created was focused on the educational aspects of this project, where the RC car design would be used to teach students of mechanical engineering concepts of mechanical, electrical, and ai design. The team used the Acclaro software for the creation of the axiomatic design of the project.

5.1 Functional Requirements (FRs)

Considering that the axiomatic design will be looking over the educational aspect of this project, and that the main goal is to teach mechanical engineering students how to design, the team decided that as their *FR0* they would have “Develop autonomous RC cars to help engineers learn design principles”. *FR0* is the most important FR and is also a representation of one of the ultimate goals of the project. The decomposition continues with the top level FRs 1 to 4 being to drive autonomously, provide opportunity to learn programming, provide opportunity to learn electrical engineering skills, and to provide opportunity to develop mechanical engineering skills. Figure 5.1 shows *FR0* and the four top level FRs that set the base for the development of more specific FRs.

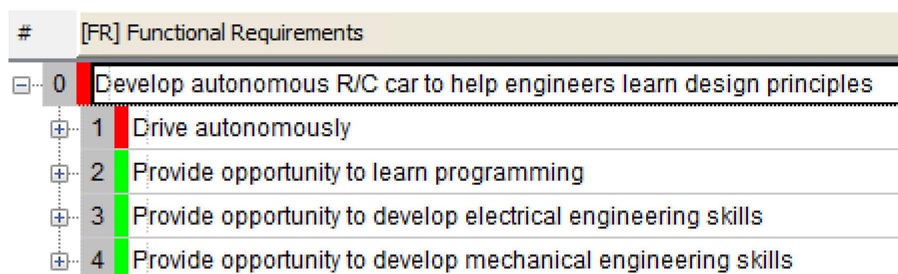


Figure 5.1: FR0 and its children FRs

The next step of creating the axiomatic design was to continue the specific decomposition by specifying further FRs under the top level FRs. Starting with *FR1*, the team was able to come up with four children FRs, with some of these also having their own children. The *FR1.1* to *FR1.4* were respectively: Provide ability to move car, remain in correct path, detect obstacles and course challenges, and avoid obstacles. All of these FRs touch in points that are needed for any autonomous vehicle. *FR1.1* and *FR1.3* were also decomposed further, where *FR1.1* looked into the different movement requirements that the RC car would need to perform, while *FR1.3* delved into more specific detection features. Figure 5.2 shows the *FR1* decomposition detail.

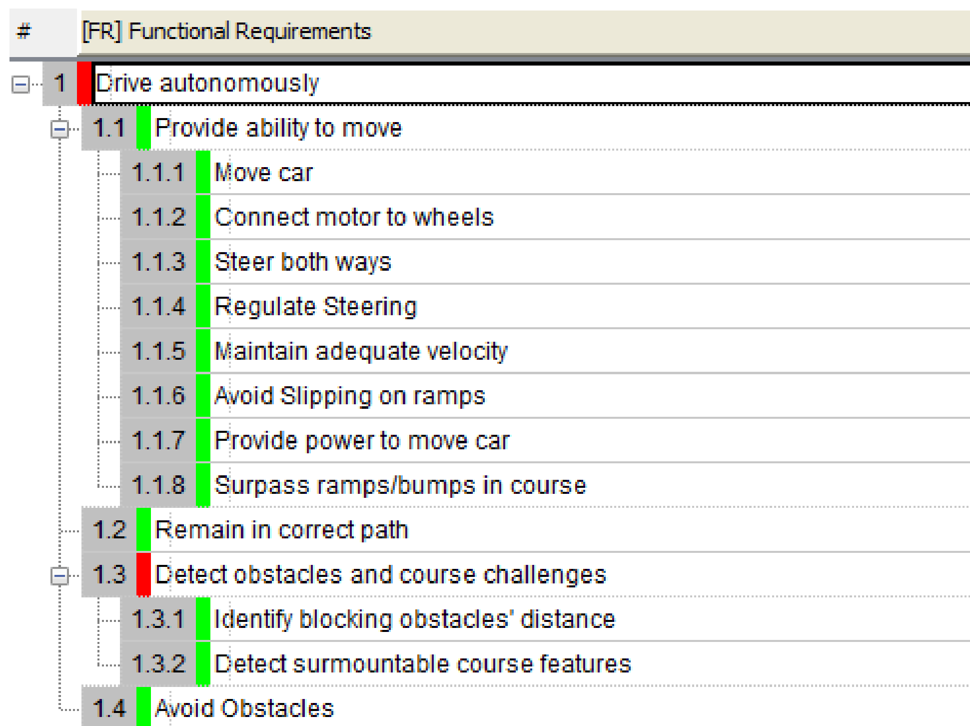


Figure 5.2: FR1 and its hierarchical decomposition

The decomposition for *FR2* was more straightforward compared to *FR1*. *FR2* was only decomposed into four children FRs and only one of those had children of its own. The second FR had to do with teaching engineering students how to use programming and how to design their

own code. The four subsequent FRs were: adapt to different courses, control electrical and mechanical systems, provide data visualization tools, and document programming codes. From these the only one that had children was *FR2.3*, which went deeper into the requirements for having a data visualization tool and the different steps that would need to be accomplished. In Figure 5.3 we can see the specifics of this decomposition.

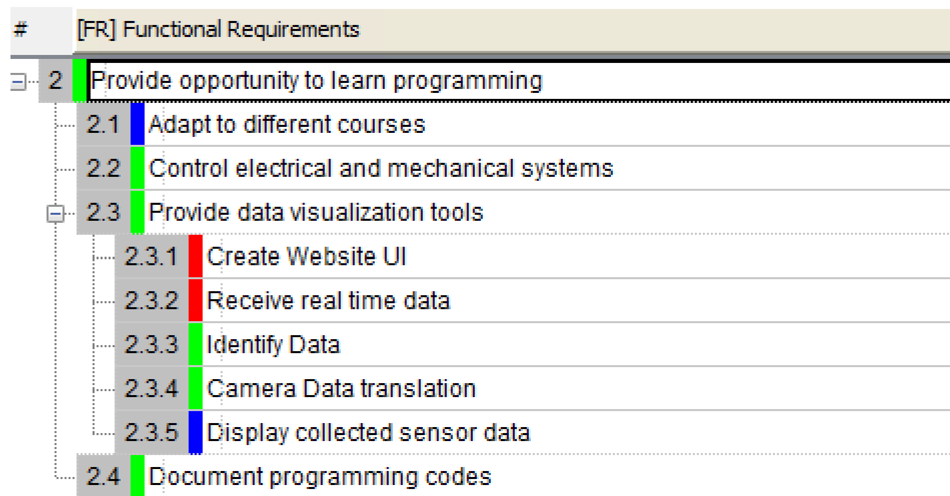


Figure 5.3: FR2 and its hierarchical decomposition

The third functional requirement has to do with the electrical engineering skills that would be useful for a mechanical engineering student to develop. It was decomposed further into four children FRs with some of them also having children of their own. The four new FRs were respectively: collect data on car's performance, learn microelectronics concepts, provide a platform of interchangeable components, and provide assistance to students regarding electrical systems. These FRs provide a balance between theoretical concepts and practical skills, as well as provide help for students through the documentation of steps (*FR3.4*). *FR3.1* and *FR3.2* were

also further decomposed to account for the different sensors and modules that would make up the autonomous RC car. Figure 5.4 shows us this decomposition.

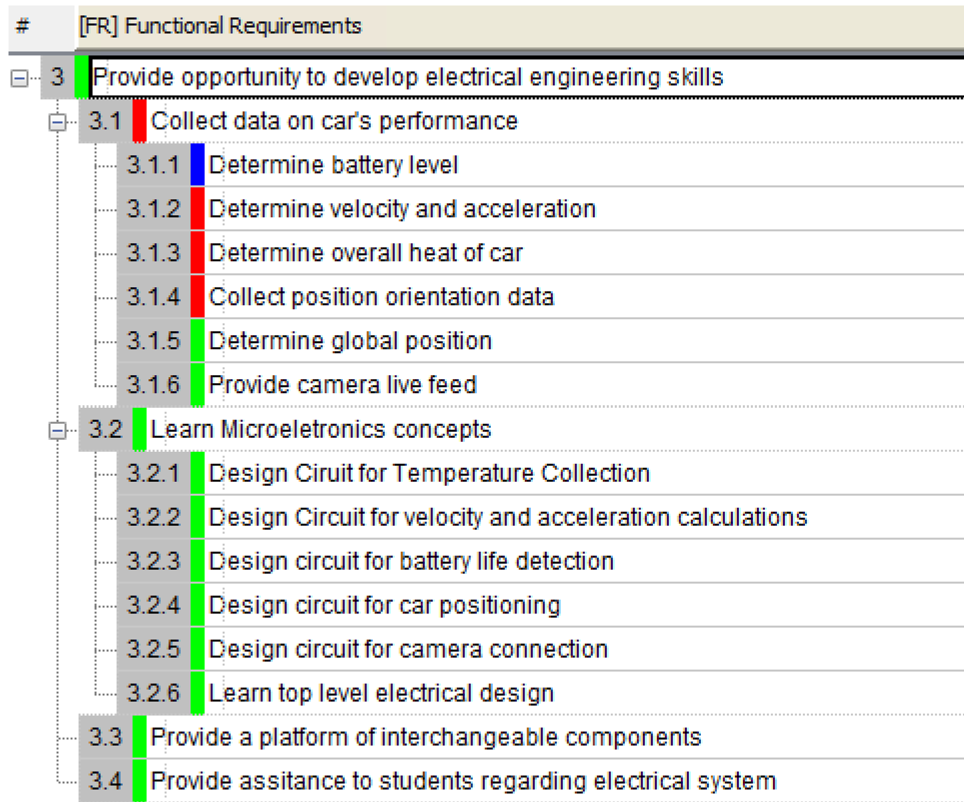


Figure 5.4: FR3 and its hierarchical decomposition

The last top level FR had to do with the additional mechanical skills that the students would learn through this project. *FR4* had only 2 children FRs with only one of them having its own children FRs. The two children FR were: provide opportunity to create parts and provide mechanical assistance to students. *FR4.1* is divided into two more FRs and the *FR4.1.1* is divided again into two more FRs. All of the FRs under *FR4.1* have to with the creation of parts for the RC car, while *FR4.2* deals with any assistance students might need when assembling the car. Figure 5.5 shows the decomposition of *FR4*.

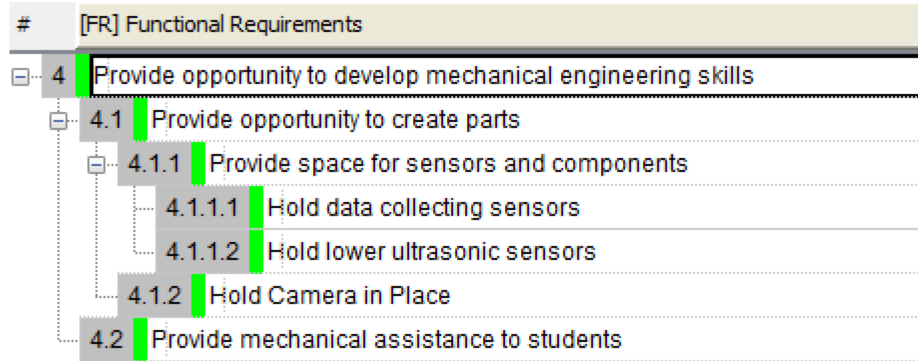


Figure 5.5: FR4 and its hierarchical decomposition

5.2 Design Parameters (DPs)

After defining all of the FRs needed for the RC car design, the team worked on assigning appropriate DPs to each of the FRs. The DPs assigned to each FR will be the physical parts, pieces of codes or systems that will be used to complete the project. To perform the DPs defining process, the team used a technique called zigzagging, where the designer cycles between the top level FRs domains and goes making its way down into the lower level FRs. Another important aspect of axiomatic design that the team had to keep in mind was that each FR could only be assigned to one unique DP. Figures 5.6, 5.7, 5.8, 5.9, and 5.10 will showcase all the FR-DP relations created for this project.

#	[FR] Functional Requirements	[DP] Design Parameters
0	Develop autonomous R/C car to help engineers learn design principles	Autonomous Car
1	Drive autonomously	Autonomous System
2	Provide opportunity to learn programming	Interconnection of Board and Internet
3	Provide opportunity to develop electrical engineering skills	Electrical System
4	Provide opportunity to develop mechanical engineering skills	Mechanical system

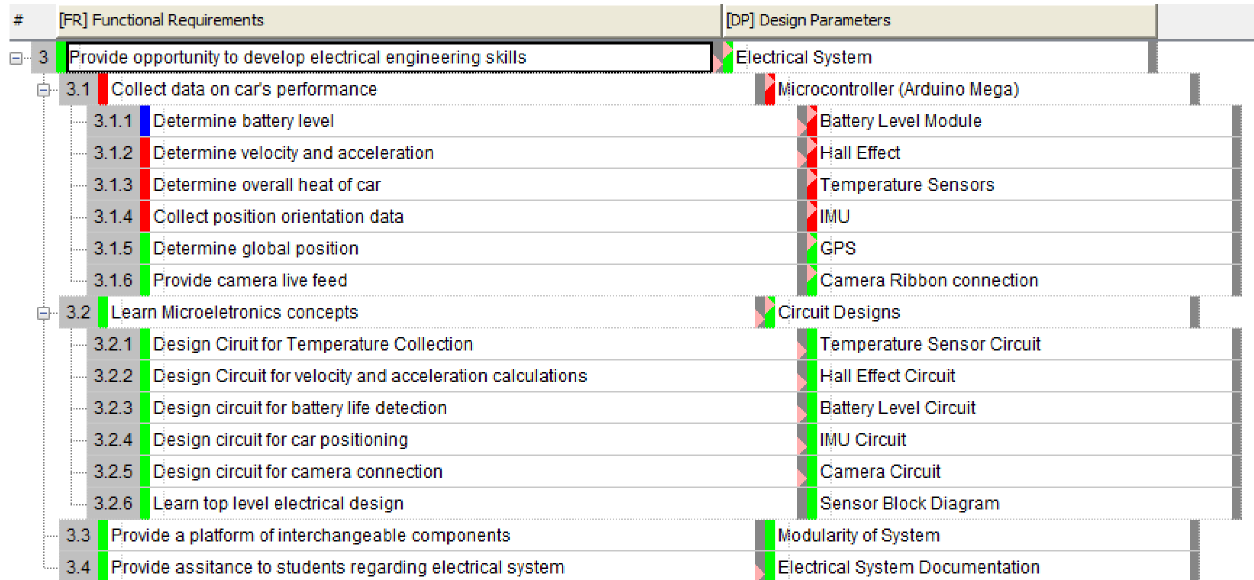
Figure 5.6: FR-DP relations of FR0 and its children

#	[FR] Functional Requirements	[DP] Design Parameters
1	Drive autonomously	Autonomous System
1.1	Provide ability to move	Motor
1.1.1	Move car	Four Wheel Drive
1.1.2	Connect motor to wheels	Gears
1.1.3	Steer both ways	Steering Linkage
1.1.4	Regulate Steering	Servo
1.1.5	Maintain adequate velocity	Speed Regulator
1.1.6	Avoid Slipping on ramps	Traction Wheels
1.1.7	Provide power to move car	Battery
1.1.8	Surpass ramps/bumps in course	Suspension
1.2	Remain in correct path	Camera Line Detection Code
1.3	Detect obstacles and course challenges	Camera
1.3.1	Identify blocking obstacles' distance	Upper Ultrasonic Sensor
1.3.2	Detect surmountable course features	Lower Ultrasonic Sensors
1.4	Avoid Obstacles	Obstacle Navigation Code

Figure 5.7: FR-DP relations of FR1 and its decomposition

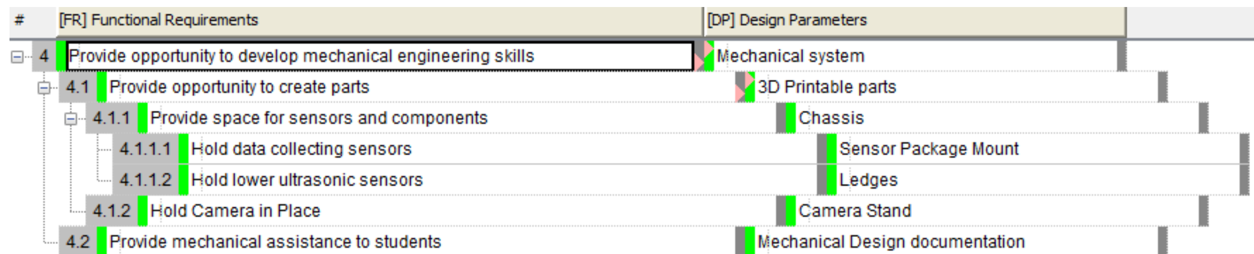
#	[FR] Functional Requirements	[DP] Design Parameters
2	Provide opportunity to learn programming	Interconnection of Board and Internet
2.1	Adapt to different courses	Machine Learning Mechanisms Code
2.2	Control electrical and mechanical systems	General Purpose Computer (Raspberry Pi)
2.3	Provide data visualization tools	Website App
2.3.1	Create Website UI	Flask Framework
2.3.2	Receive real time data	RF connection between Raspberry and Website
2.3.3	Identify Data	Tech Stack
2.3.4	Camera Data translation	Bird's Eye View Code
2.3.5	Display collected sensor data	Front End Dashboard
2.4	Document programming codes	Github

Figure 5.8: FR-DP relations of FR2 and its children



#	[FR] Functional Requirements	[DP] Design Parameters
3	Provide opportunity to develop electrical engineering skills	Electrical System
3.1	Collect data on car's performance	Microcontroller (Arduino Mega)
3.1.1	Determine battery level	Battery Level Module
3.1.2	Determine velocity and acceleration	Hall Effect
3.1.3	Determine overall heat of car	Temperature Sensors
3.1.4	Collect position orientation data	IMU
3.1.5	Determine global position	GPS
3.1.6	Provide camera live feed	Camera Ribbon connection
3.2	Learn Microeletronics concepts	Circuit Designs
3.2.1	Design Ciruit for Temperature Collection	Temperature Sensor Circuit
3.2.2	Design Circuit for velocity and acceleration calculations	Hall Effect Circuit
3.2.3	Design circuit for battery life detection	Battery Level Circuit
3.2.4	Design circuit for car positioning	IMU Circuit
3.2.5	Design circuit for camera connection	Camera Circuit
3.2.6	Learn top level electrical design	Sensor Block Diagram
3.3	Provide a platform of interchangeable components	Modularity of System
3.4	Provide assitance to students regarding electrical system	Electrical System Documentation

Figure 5.9: FR-DP relations of FR3 and its children



#	[FR] Functional Requirements	[DP] Design Parameters
4	Provide opportunity to develop mechanical engineering skills	Mechanical system
4.1	Provide opportunity to create parts	3D Printable parts
4.1.1	Provide space for sensors and components	Chassis
4.1.1.1	Hold data collecting sensors	Sensor Package Mount
4.1.1.2	Hold lower ultrasonic sensors	Ledges
4.1.2	Hold Camera in Place	Camera Stand
4.2	Provide mechanical assistance to students	Mechanical Design documentation

Figure 5.10: FR-DP relations of FR4 and its children

5.3 Design Matrix

Once all the DPs were assigned to the FRs, the team was able to move on to the design matrix to check if the design would satisfy the Independence Axiom. The way to check if the design would satisfy the axiom is to check if the design matrix is uncoupled or decoupled. Generally, it is virtually impossible to achieve a fully uncoupled design matrix in large projects due to the complexity of the project's systems, so our team looked forward to finding a decoupled matrix. Figure 5.11 shows the top level design matrix of the project.

	DP0: Autonomous Car	DP1: Autonomous System	DP2: Interconnection of Board and Internet	DP3: Electrical System	DP4: Mechanical system
FR0: Develop autonomous R/C car to help engineers learn design principles	X				
FR1: Drive autonomously		X	O	X	X
FR2: Provide opportunity to learn programming		O	X	O	O
FR3: Provide opportunity to develop electrical engineering skills		O	X	X	O
FR4: Provide opportunity to develop mechanical engineering skills		O	O	O	X

Figure 5.11: Top Level Design Matrix of MPAD Project

The current system shown in the matrix above is a decoupled system. This means that the axiomatic design is following the first axiom of independence. Further work could be done on this matrix to make it a triangular matrix, which would make it more obvious that the design matrix is uncoupled. To achieve that, some of the FRs and DPs would have to be moved around. By knowing that the design is decoupled the team was able to see that the design was adequate, and that they could start focusing on creating the DPs to assemble the autonomous RC car.

5.4 Conclusion

After going through the Functional Requirements, Design Parameters, and Design Matrix the group had a pretty good idea about what the flow and decomposition of the project would

look like. With Axiomatic Design it was possible to plan which design parameters should be done before others, so that the design of the car was efficient and contained minimum flaws. The next steps after completing the Axiomatic Design, is to follow the design matrix and work on the various aspects of the project. In the next chapters, the team will cover the methods of the project. The methods chapter contains the timeline of what was done during every term.

6. Methods

In the methods section of this paper, the team covers how the MPAD was created and the steps involved in the car's sensors, autonomous driving, and dashboard development. The chapter is divided into terms to make it easier to follow along the team's progress over time.

6.1 A term

A term was the starting phase for this project. Our team met three times a week to discuss various aspects of the project. The first task was reading and getting familiar with the 2019-2020 Autonomous vehicle MQP project. This included reading their reports and analyzing their data. For their project, they choose to use the Donkey car platform. We followed their guide and remodeled the donkey car using the information they provided. Some of the challenges faced by our team when replicating the previous MQP were the lack of instructions of setting up the Raspberry Pi, requiring previous knowledge on Linux and bash commands, and the use of machine learning to self drive. Most documentation of the previous MQP was not easy to use and required a lot of background knowledge on specific subjects. The machine learning approach also caused a lot of issues with regards to the gathering of data, which would take hours with no guarantee on the quality of the data. Another problem with machine learning was that multiple times we would reach the memory capacity of the Raspberry Pi and not even finish collecting all data. After trying out the previous MQP's methods,we started researching on how to implement lane following, display real time data and create a non-track specific system.

From the sensor package side, the team looked for ways to improve last year's sensors and implement new ones. We took inventory of all the previous team's parts and we set up the previous team's sensor package. We then created a gantt chart based on our axiomatic design,

which allowed us to plan our project and keep track of the milestones. Our full gantt chart can be found in Appendix D. We finished the term with a complete list of the sensors we would implement, the programming boards, and the relevant code libraries.

6.2 B term

The primary purpose for B term was to implement the sensors and lane following system. For the sensors, we began by creating schematics for each sensor. We then moved on to wiring each sensor on the breadboard, after that we began programming the sensor on the arduino board. For each sensor we began testing them for their output. This included figuring ways to test each sensor and recording the data. Lastly we were communicating with the ME team on the positioning of the sensors and their dimensions for the RC car they were designing.

At the start of B term, we also quickly reviewed the research we had conducted in A term and finalized the framework and technologies to use for the web application. With this, the team sketched out what the driving dashboard would look like and began to create a skeleton for feedback. As the term went on, we made design revisions to remove some unnecessary components, such as a joystick for manual control of the car. Through feedback from the professor and the team, we also found it necessary to add new components, such as more graphs for sensors and steering/directional controls once we began testing. At this point nothing was functional, just intended to be a foundation for when development begins. As mentioned before, the main focus was to implement the lane-following code.

The initial step for lane following was lane recognition. We started familiarizing ourselves with OpenCV. We attempted to implement line detection, birds eye view, and a color filter. After testing, the color filter was determined to be the only filter of use. The color filter needed a range of RGB values to be inputted so we developed a color finder that returned these

based on mouse clicks. With clear lanes, three different lane following techniques were developed in parallel:

1. Ray-tracing: In this approach, the program would draw a multitude of 'rays' from the car's current location to the edges of the camera's sight. Each ray was followed from the origin until it encountered a lane. The longest line was considered the best path, with the car steering to match the angle.
2. Trigonometric: In this approach, the program used both a Color filter and Line Detection filter. The Color filter was applied to a lower and upper bound of HSV values to highlight only the course lanes. Then, a Line Detection filter was applied to get the average angle that the path was directing. The program then took the horizontal offset from the center of the lane (using the average of the x-coordinates), the angle of the lane, and applied trigonometry to determine the final angle the car should proceed.
3. Lane distance comparison: In this approach the only filter used was the color filter. Tha camera image is split in two as the first horizontal third of the image and the last third of the image. Getting the x coordinates of the two lanes, the program calculated the distance of each lane from the center. By subtracting the distance of the left lane from the distance of the right lane and dividing the result by the width of the image, the program arrived at a small number between -0.5 and 0.5. Multiplying this number with 180 and adding 90 to it resulted in the desired steering angle.

Techniques 1 and 2 turned out to be too computationally taxing. All three approaches were tested in a track, and they directly affected the car's ability to drive smoothly. The lane distance comparison method proved to be the best choice due to its low runtime. The low latency allowed the car to go faster and smoother. The track was completed reliably and smoothly and lane following was implemented.

6.3 C term

In terms of driving, C term was focused upon obstacle avoidance. In order to still stay in the lane while avoiding obstacles, the camera has to be used alongside the ultrasonic sensors. This meant that we had to create a calibrations program that translated each ultrasonic sensor reading to a pixel on the frames. During testing, we saw how the shortcoming of the ultrasonics would affect the driving and modified our sensor layout. By the end, there were five ultrasonic sensors at the front along with one on the left and one on the right side of the car. Of the five ultrasonics sensors in the front, two of them were angled around 45 degrees in order to catch surfaces that the forward facing ones would miss. By the end of C term, small tests were done but due to the need of a custom 3-D printed bumper, no large scale tests were conducted.

Another goal in C term was to finalize the MPAD Web Application. The team had created a framework in B term, so we steadily began implementing features. During this term, the ECE team and CS team had to share a car, so coordination was required to do real testing. For that reason, another big goal for this term was for the CS team to create a local testing routine, so that the team could simulate a car's data streaming. The team expected to completely finish the dashboard by the end of the term, but ran into more problems than expected with camera and data streaming, which put us slightly behind schedule. The team had to postpone proper testing with an actual Raspberry Pi on a car until after these problems were resolved. Finally, the CS

team wrote the framework for the startup script this term. This was the final part to implement into MPAD once everything else had been properly tested.

From the sensor aspect of the project, the team started planning on how to simplify the circuit design and provide a concise way to present it to the students of the Introduction to Engineering Design (ME2300) course. Different options were considered and the team decided to go with a soldered board approach, so that the connections were easily recognizable. While working on the ultrasonics, the team also noticed that the ultrasonics were causing a lot of latency issues to the arduino code and added a timeout feature where if the ultrasonics did not find any obstacle after about 1 meter, the function would just timeout and continue with the remaining of the code. Lastly, during this term, the team started preparing the Sensor and AI Setup Guide and the Arduino IDE Setup Guide.

6.4 D term

In the first few weeks of D term, the team worked hard to fully integrate the web dashboard with a functional RC Car. An important step was to first create a virtual environment. Because external libraries and dependencies were used in the driving and startup code, we needed to make sure that all of these requirements would come pre-installed for the students. By creating a virtual environment, we were able to install only the dependencies that our self-driving module needed, and nothing else that would take up space on the Raspberry Pis. Before making copies on our master SD card, we first verified that our car was auto-enrolling to our server via the startup script, streaming camera feed consistently without crashing, and responding to user driving controls in real time. The team spent a week or two of debugging and testing to make sure that everything was working properly. Finally, the team just needed to adjust the Arduino

and startup script code to reference the correct naming scheme, and the dashboard was successfully able to graph MPADs live sensor data.

In terms of driving in D term, the two main focuses were finishing obstacle avoidance using practical test and refining lane following in order to make sharper turns along with more complex tracks. The main failing of the B-Term driving code was that it ignored the middle of the camera input and that it could not make a decision when the lane was horizontal in front of it, not vertical. This led to the implementation of y coordinate based lane following for the middle third of the screen. This meant that instead of comparing how much to the left or right of a lane was, we compared how much closer to the bottom of the screen a lane was. The feature was fully implemented and the increase in functionality became clear in testing. Obstacle avoidance was finally able to be tested with the bumper and ultrasonics in place. Expectedly, some issues emerged. The entirety of the ultrasonic calibration code and the obstacle avoidance code was rewritten. The obstacle avoidance code just colors the area of the image where an obstacle is detected as a lane and lets the lane following code do the rest. By the end of the term, the car was able to maneuver around obstacles while still staying in the lane as long as there was a space at least 20% larger than the width of the car.

This term was also the main term where the team worked on transferring all of our information and data into the ME2300 course. From the sensors side, the team soldered 8 boards and prepared multiple kits to provide for the teams in the course. The team finished the guides for the mechanical engineering students and eventually implemented it. While the course was still ongoing, the team went and talked with the mechanical engineering students about the MPAD implementation in their RC cars. In order to collect data from students about their experience with MPAD, the team submitted an IRB referenced in Appendix M. The students

provided feedback on what they believed was still hard to understand in the guides and the team was able to update them accordingly.

6.5 Conclusion

In this chapter, the team went over what was done during the four different terms of this project. With the guidance from the axiomatic design and the gantt chart the team was able to keep track of the requirements for every aspect of the project. In the next three chapters, the team covers the implementation of the main parts of the project, which were the sensors, autonomous driving, and dashboard.

7. Sensor Implementation

This chapter covers how the Elegoo Mega 2560 is used to compile the sensor package and how the code for each sensor works. Alongside that, this chapter also goes over the sensors' schematics and explains the reasoning behind their design.

7.1 Elegoo Mega 2560

The Elegoo Mega 2560 is a programmable circuit board with Arduino IDE (Integrated Development Environment) compatibility[23]. Arduino itself creates its own programmable boards as well as its own software development environment. For the goals of this project, we focused on the software development environment. The code was written in C++, a system development language. The sensor schematics listed below allowed the team to create a sensor package to be implemented in the MPAD.

7.2 Design of Sensor Package

The first task that the group decided to do regarding the sensor package design was to create a block diagram to represent how each sensor would be connected to the electrical part of the car. The team decided that most of the sensors should be connected to a separate board from the Raspberry Pi 4, to allow for more processing power on the Raspberry Pi when the car was driving autonomously. The secondary board chosen was an Elegoo Mega 2560, which is compatible with Arduino's IDE.

After choosing the main processor boards, the next step was to choose which specific sensors should be used to meet the requirements of this project. From the Goals and Requirements (Chapter 4), the sensors must provide the following information on the conditions of the car:

- Temperature of the motor and battery
- Acceleration and Orientation
- RPM
- Distance between the RC car and obstacle object

To measure the temperature of the motor and battery, a simple temperature sensor would suffice. This sensor would not need to be extremely accurate; instead, it would just need to be able to comprehensively regulate its readings based on its surroundings. This is the case because the temperature sensor is there mainly to keep the temperature of the RC car under a certain threshold decided by the specific design of the car and the electronic parts of the RC car. To meet the second requirement, the group chose to use an Inertial Measuring Unit (IMU). The IMU is an extremely complete sensor usually containing gyroscopes and accelerometers that allow it to give a good idea of the positioning and orientation of any 3D object. In this project, the IMU was crucial for the use of the RC cars in ramps because they would allow the Raspberry Pi to call for subroutines allowing the RC car to go through ramps. To measure the RPM, the chosen sensor was a hall-effect sensor due to their low cost, high efficiency, and regular use in mechanical heavy projects. The last of the requirements was the hardest to actually decide which sensor to use. After a lot of debate among the team members, the chosen sensor to measure the distance between the RC car and the obstacles was the ultrasonic sensor. The two main competitors for the ultrasonic sensors were radar and lidar sensors, however, the team decided that ultrasonics was the way to go mainly to keep the budget to the low side. Both radar and lidar are great sensors for normal cars because they are worth the investment since they actually provide a revenue for their manufacturers, however, when doing experiments on RC cars lidars and radars are to the expensive side, sometimes even surpassing the price of the Raspberry Pi and Elegoo

Mega2560. Ultrasonics sensors would have its limitations, but would still be able to fulfil the requirement to its entirety. And with that, the team decided to use the ultrasonic sensors, temperature sensors, IMU, and hall effect sensor as their main data gathering sensors connected to the Elegoo Mega 2560. The sensor package still had a camera that would be connected to the Raspberry Pi and would provide the data for the AI side of the project. In the succeeding parts of this subsection of the methods chapter, the team will break down the design of each specific sensor part.

7.3 Temperature Sensor

The temperature sensor the group decided to use was the DHT11[24]. Some of the specifications for the DHT11 are:

- 3 to 5V power and I/O
- 2.5mA max current use during conversion (while requesting data)
- Good for 20-80% humidity readings with 5% accuracy
- Good for 0-50°C temperature readings ±2°C accuracy
- No more than 1 Hz sampling rate (once every second)
- Body size 15.5mm x 12mm x 5.5mm
- 4 pins with 0.1" spacing

Figure 7.1 shows a photo of the DHT11 temperature sensor.

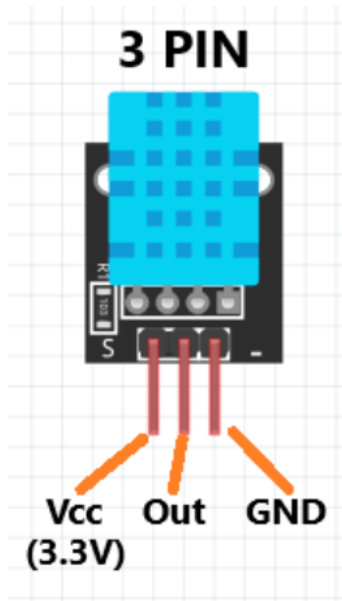


Figure 7.1: Image of DHT11 Temperature Sensor

This temperature sensor proved ideal due to its low cost and reliable accuracy. The configuration on the board can be seen in Figure 7.2. We used three temperature sensors in our car. One was positioned near the motor and the others close to the batteries since these are the three parts that heat up the most. The temperature sensors were connected to the Elegoo with one pin connected to power and another one connected to the ground. The middle pin in the temperature sensor is the data out pin, which provides the temperature reading. To improve the results/resolution from the data collected by the temperature sensor, we added a pull-up resistor of 5kΩ.

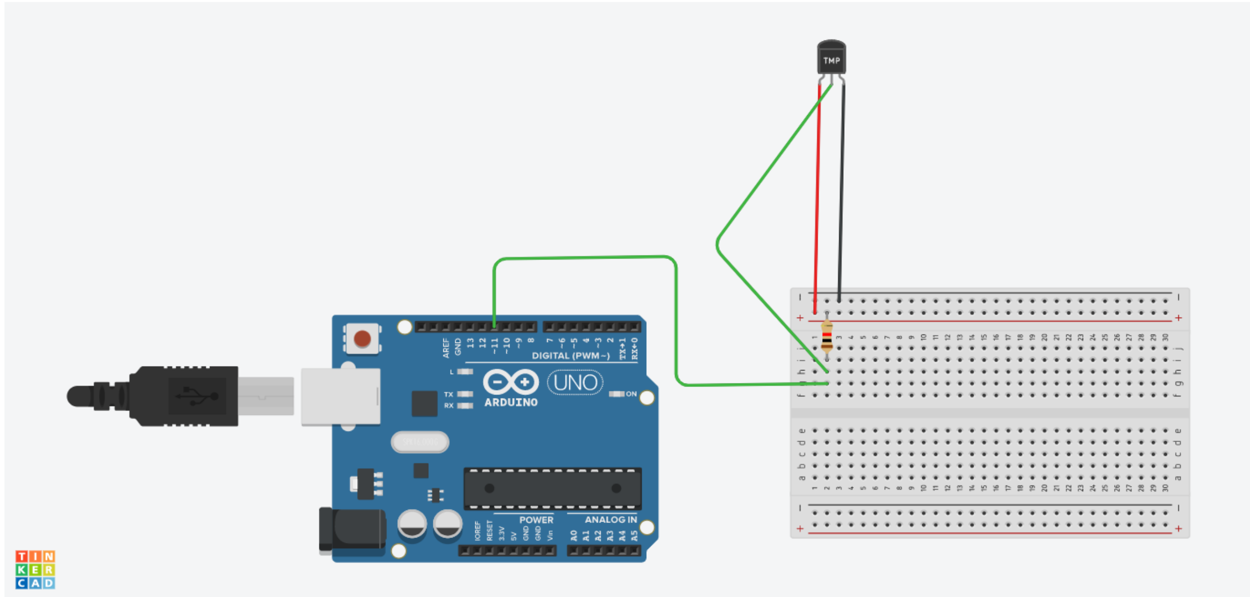


Figure 7.2: Temperature Sensor Setup Schematic

The Arduino IDE comes with useful examples to construct the code needed for these sensors to function as intended. The commands implemented to read temperature were from the DHT.h and DHT_U.h libraries. The *.readHumidity()* may be useful in the future, however the most useful commands under the scope of this project were the *.readTemperature()* and *.computeHeatIndex()*. This command was stored in a float variable and updated every three seconds. In order to incorporate the delay of three seconds, we used a counter and modular operand as displayed below.

```
acc = acc + 1 ;
    if((acc % 30) == 0) {
        float h = dht1.readHumidity();
        // Read temperature as Celsius (the default)
        float t = dht1.readTemperature();
        // Compute heat index in Celsius (isFahreheit = false)
        float hic = dht1.computeHeatIndex(t, h, false);
```

The values were stored serially in both Fahrenheit and Celcius and then exported to the Raspberry Pi.

7.4 Inertial Measuring Unit

The Inertial Measuring Unit (IMU) allows us to know the orientation of the car. Our IMU of choice was the BNO055 from Adafruit[25]. Figure 7.3 displays the BNO055 Inertial Measuring Unit (IMU). This specific IMU model was chosen due to its ease of use (it is easy to find Adafruit libraries in the Arduino IDE) and great compatibility with DHT11 temperature sensors having the same manufacturing company in Adafruit. The BNO055 comes with a 3.3V regulator, logic level shifting for the Reset and I2C pins, and an external 32.768KHz crystal. It can output the following sensor data:

- Absolute Orientation (Euler Vector, 100Hz) Three axis orientation data based on a 360° sphere
- Absolute Orientation (Quaternion, 100Hz) Four point quaternion output for more accurate data manipulation
- Angular Velocity Vector (100Hz) Three axis of 'rotation speed' in rad/s
- Acceleration Vector (100Hz) Three axis of acceleration (gravity + linear motion) in m/s^2
- Magnetic Field Strength Vector (20Hz) Three axis of magnetic field sensing in micro Tesla (uT)

- Linear Acceleration Vector (100Hz) Three axis of linear acceleration data (acceleration minus gravity) in m/s^2
- Gravity Vector (100Hz) Three axis of gravitational acceleration (minus any movement) in m/s^2
- Temperature (1Hz) Ambient temperature in degrees celsius

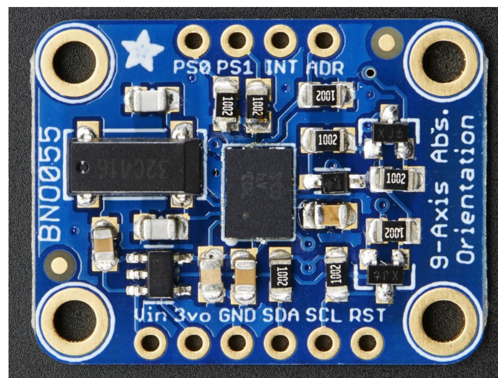


Figure 7.3: Image of BNO055 IMU sensor

The IMU is essential for any ramp-related controls the RC car has. The connections for the IMU were simple. We connected the IMU to the power and ground rail in specific positions. Next, we connect the SDA and the SCL directly to the SDA and SCL pins of the Elegoo Mega. We also soldered the header connector male pins to the IMU to make the connections stable and to move the IMU with more ease. Figure 7.4 and Figure 7.5 show how the IMU is connected to the Elegoo Mega 2560 and power.

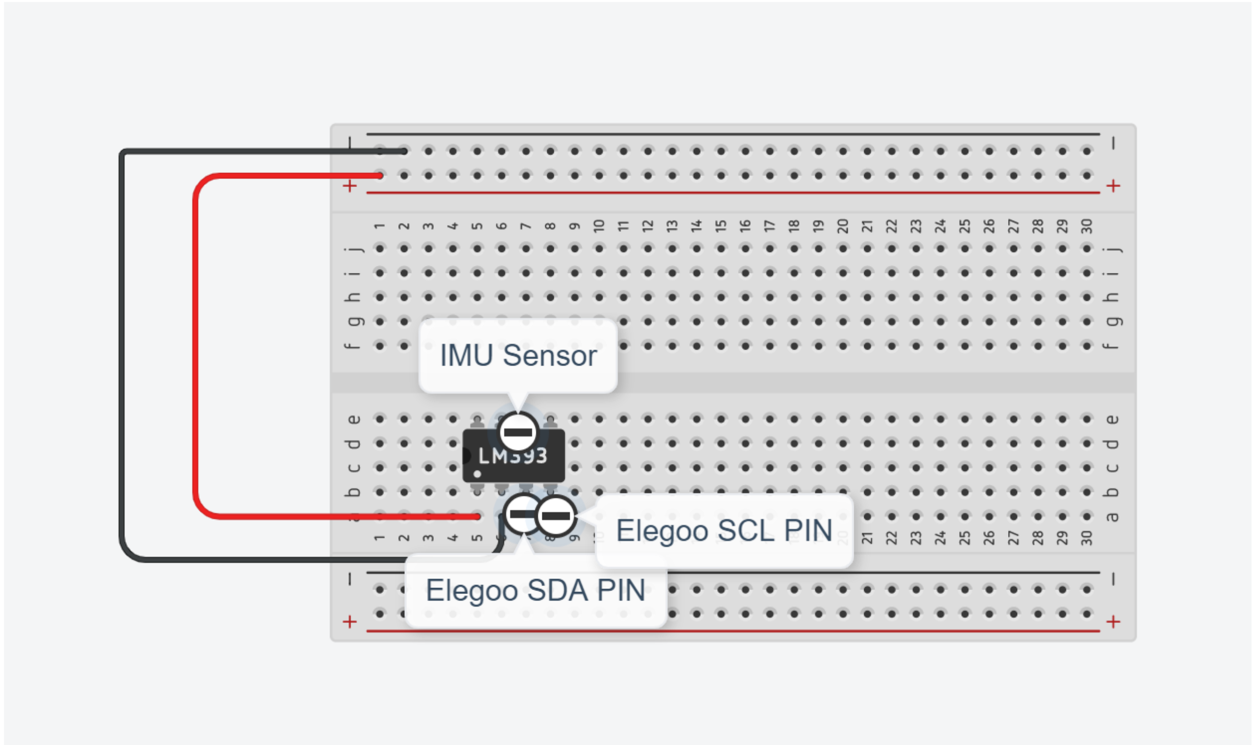


Figure 7.4: IMU Sensor Wiring

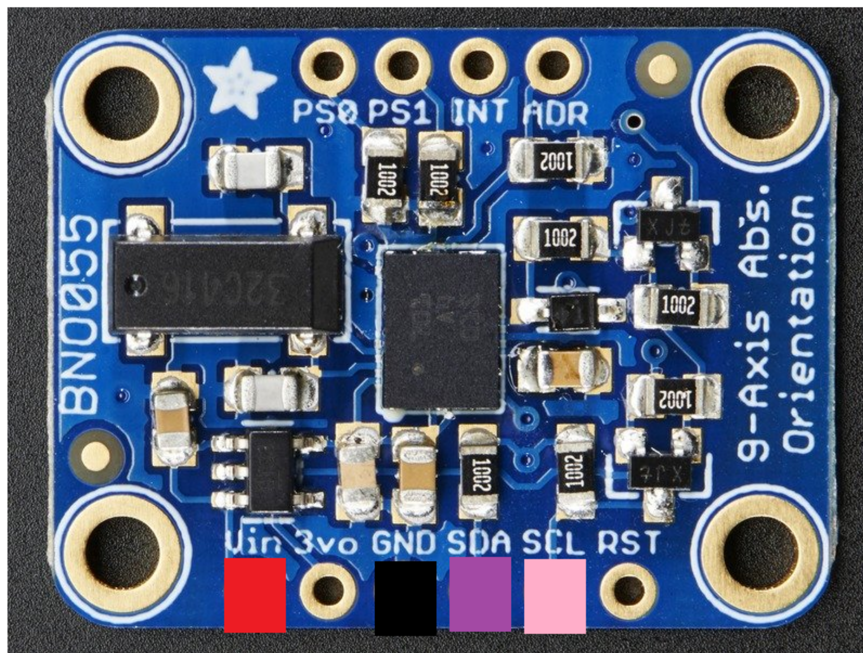


Figure 7.5: IMU Pins and Corresponding Wires

Figure 7.5 represents the IMU, (although the BNO055 actually has 10 ports, the ones we need are specifically labeled). To wire up IMU, there needs to be at least 4 pins wires soldered/connected: Vin, GND, SDA, and SCL. As seen in Figure 7.4, Vin connects to the positive rail on the soldered board. GND connects to the negative rail on the soldered board. SDA is plugged in directly into the SDA pin on the Elegoo Mega 2560. SCL is plugged in directly into the SCL pin on the Elegoo Mega 250.

To instantiate the IMU we included the adafruit BNO055 library using the line of code

```
#include <Adafruit_BNO055.h>
```

The library includes how to set up and initialize the sensor by using a *.begin()* command. The code also consists of a fail-safe in the case that the IMU is not detected by the Elegoo Mega. The code to set up the IMU is shown below:

```
void setup_imu() {
    Serial.println("Setting Up IMU");
    Serial.println("");
    //alert coder that IMU detection and set up has began

     /* Initialise the sensor */
     if (!bno.begin())
    // If IMU not detected
     {
       Serial.print("Ooops, no IMU detected ... Check your wiring or I2C ADDR!");
       while (1);
    // Holds printed line until issue is fixed
     }
     delay(1000);
```

```
bno.setExtCrystalUse(true);
}
```

To read the measurements we required, position on the x y and z-axis we used the *.getEvent(&event)* command. This allows the user to get a live look at the status of the IMU while setting the initial condition of the IMU (0,0,0). In order to get the reading the code below was used.

```
if ((acc % 5) == 0) {
        x_box = event.orientation.x;
        y_box = event.orientation.y;
        z_box = event.orientation.z;
 if (x_box > 180){
        x_box = -(360-x_box);
 }
...
```

Using the *.orientation.* command reads the orientation of the IMU in that iteration of the loop. Also, notice that the counter and modular operand were implemented to the IMU as well. Instead of a delay of three seconds, this use of the operand sets the delay to read the data every half second. Lastly, at the end of the function, there is the *if (x_box > 180)* statement. This ensures the orientation in the x-direction stays within the bounds of 0 to 180 degrees.

7.5 Hall Effect Sensor

A hall effect sensor was used to measure the RPM of the car. It was placed directly next to the motor and read with a magnet. The hall effect sensor used was the KY-003[26]. This sensor

has three pins: Vin, GND, and the output data pin connected to the Elegoo Mega 2560. The main reason for choosing this specific model was due to the low cost and good accuracy. Figure 7.6 shows the KY-003 model Hall Effect Sensor.

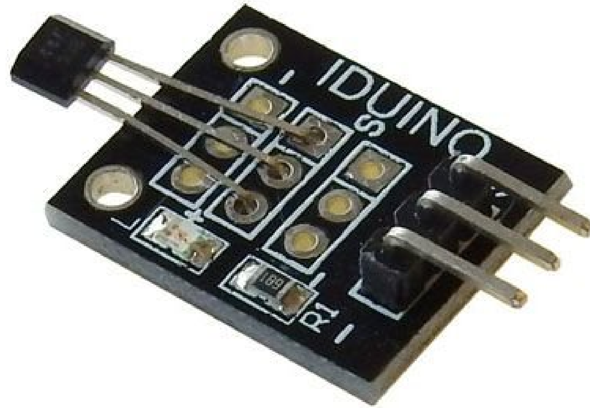


Figure 7.6: Image of the KY-003 Hall Effect Sensor

To make the connections for the hall effect sensor, the team had the Vin pin connect to the positive rail of the breadboard, the GND pin connect to the negative rail of the breadboard, and the Data Out pin was connected pin number 2 in the Elegoo Mega 2560. Figure 7.7 shows the representation of how the hall effect sensor should be connected.

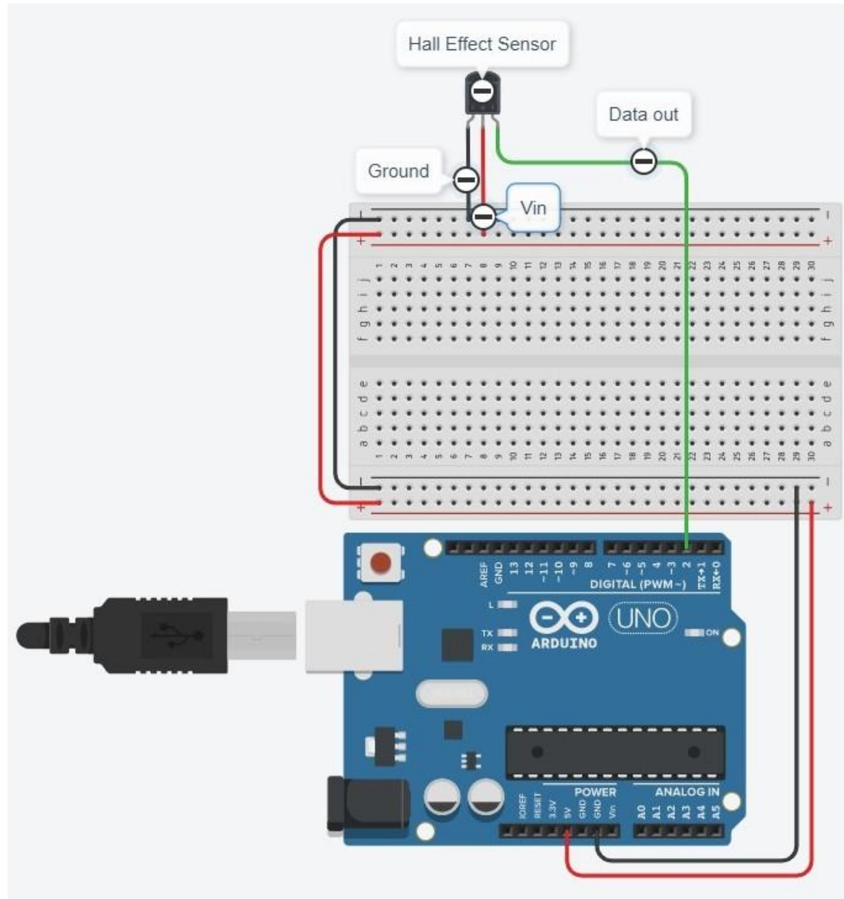


Figure 7.7: Hall Effect Sensor Wiring

The hall effect sensor works by reading the magnetic pulses provided by a small anpro magnet. When a magnetic pulse is detected by the sensor the sensor will return a value of 1 or logic high. Otherwise, the sensor will output a value reading of 0 or logic low. For the RPM readings to work, a simple counter algorithm was implemented to the sensor package.

```
if (digitalRead(hall_pin)==0){
    if (on_state==false){
      on_state = true;
      hall_count+=1.0;
    }
```

```
...
    // print information about Time and RPM
  float end_time = micros();
  float time_passed = ((end_time-start)/1000000.0);
  float rpm_val = (hall_count/time_passed)*60.0;
  Serial.print(rpm_val);
  Serial.println(" RPM");
  delay(1);        // delay in between reads for stability
```

The purpose of the counter was to take a few data points per rise time of the clock and average the data collected over a minute to produce RPM. The reading of an example oscilloscope reading can be seen below.

7.6 Ultrasonic Sensor

The ultrasonic sensor used is the HC-SR04[27].This model is powered with 5v DC and can measure from 200cm to 400 cm. Given the width of our car, we decided to have multiple ultrasonics to cover the whole range. The ultrasonic sensor has four pins, which are Vcc and GND. The Trig pin is for signal transmission and the echo pin is for outputting the signal. This model was chosen because it is inexpensive, and our team had previous experience using it in past projects. Figure 7.8 displays the ultrasonic sensor with model HC-SR04 used in this project.

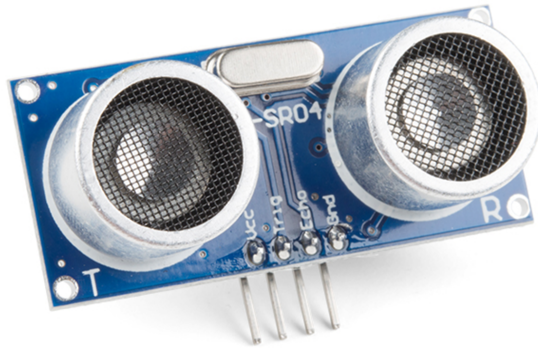


Figure 7.8: Image of the HC-SR04 Ultrasonic Sensor

This sensor is one of the most important parts of the sensor package. It gave the car the capability to know how far away obstacles are within the constraints of the track. Ultrasonic sensors work by sending a sound wave pulse through its transmitter or trigger side and once that wave hits an object it is received via the receiver or echo side. Once the signal is back, it calculates how far away the object was by using the speed of sound multiplied by half the time it took for the signal to come back. To wire the ultrasonic properly, we used the schematic design shown in Figure 7.9.

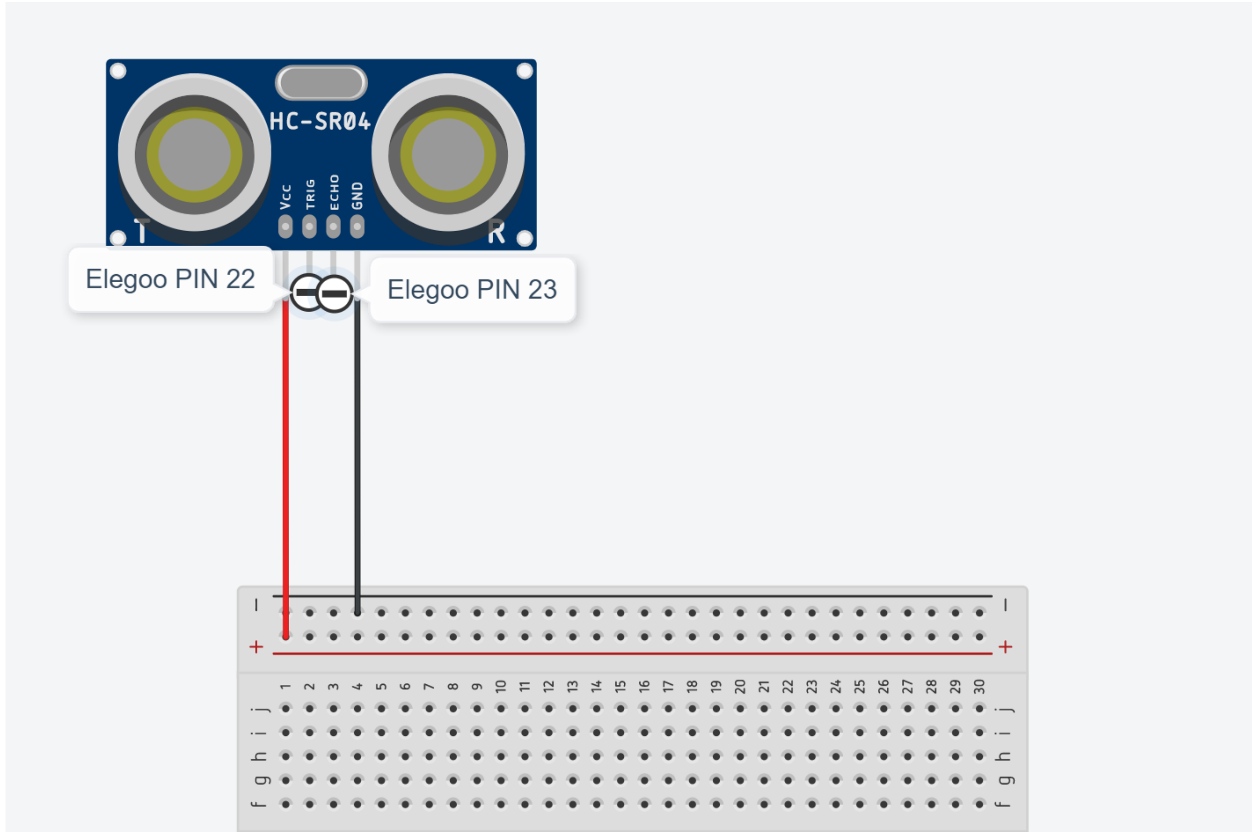


Figure 7.9: Ultrasonic Sensor Setup Schematic Example

In figure 7.9, we can see that we have a single ultrasonic sensor connected to power, ground, and to pins 22 and 23 in the Elegoo Mega 2560. Each different ultrasonic sensor needs a specific echo and trigger pin connection in the Elegoo. If they all had the same trigger pin the sound pulses might interfere with one another, while if they had the same echo pin the values of our readings would be wrong. For the trigger and echo pins, we connected them directly to the ultrasonic sensors. In our sensor package code, the following pins in the Elegoo Mega 2560 were responsible for the trig and echo functions of the ultrasonic sensors:

- Sensor 1: Trig → 22 | Echo → 23
- Sensor 2: Trig → 24 | Echo → 25
- Sensor 3: Trig → 26 | Echo → 27
- Sensor 4: Trig → 28 | Echo → 29
- Sensor 5: Trig → 30 | Echo → 31

- Sensor 6: Trig → 32 | Echo → 33
- Sensor 7: Trig → 34 | Echo → 35

Instantiation of the ultrasonic sensors took two *const int* assignments. One specified for the trigger pins and one for the echo pins. To keep the board clean and the code simple, we made all trigger pins odd numbers and all echo pins even numbers on the digital ports of the Elegoo Mega. We also set two global *floats* for the duration of the ultrasonic pulse and the reading of the distance. During the setup, the only necessary step was to distinguish which pins were for output and which were for input. All trigger pins are set to output because they produce ultrasonic pulse and the echo pins are set to input because they receive the signal from the trigger. Lastly to send the signals and read them into the float variable *duration#* We used the *digitalWrite()* and *pulseIn()* functions as shown below.

```
digitalWrite(trigPin1, LOW); //Signal not being sent
  delayMicroseconds(2);
  digitalWrite(trigPin1, HIGH); //Signal sent for 10 microseconds
  delayMicroseconds(10);
  digitalWrite(trigPin1, LOW); //Signal not being sent
  duration1 = pulseIn(echoPin1, HIGH); //Read and store the received signal
  distance1 = (duration1*.0343)/2; //Take the time it takes for signal to be
received and conver to distance
  data_out.concat("|Distance 1:"); //add distance to serial output
  data_out.concat(distance1);
```

7.7 Battery Monitoring

Another idea the team had for a sensor to be added to the car was a battery monitoring unit. The first idea the team came up with was to have a voltage divider circuit which would measure the total voltage stored in the lipo battery. However, this proved to be a big challenge since the connection between lipo battery and the ESC could not be splitted to get the actual reading from the lipo battery. This means that either the lipo battery powers the whole mechanical system or our circuit gets the reading from it, but the latter has no useful function with the car not getting powered. With that in mind, our team still searched for a way to provide somewhat of a battery monitoring feature and was able to find a low voltage buzzer alarm (model BX100)[28], which would start beeping whenever the lipo battery would start running low. In the Conclusion Chapter 13, the team mentions battery monitoring in the future work section. Figure 7.10 shows the BX100 model for the low voltage buzzer alarm image.

Figure 7.10: Image of the BX100 Low Voltage Buzzer Alarm

7.8 Sensor Selection

When selecting the final sensor, we chose to use five ultrasonic sensors placed in the front for obstacle avoidance. The reason for the need of so many ultrasonic sensors was the fact that whenever an ultrasonic sensor sends a sound pulse towards an angled object, there is a high chance that the sound will not come back to the ultrasonic. Hence, the need for multiple ultrasonics with different angles, to capture any object (angled or not) that appears in front of the RC car. Additionally, the ultrasonics have a narrow range (a total of 30°), which is not optimal to implement obstacle avoidance. This is further discussed in the obstacle avoidance section in chapter 8. The ultrasonic are able to detect objects within a short-range of 200cm to 400 cm. Two temperature sensors were chosen to collect thermal data on the motor and battery. The temperature sensor has the ability to detect if any component's temperature is high. This information is also displayed on the dashboard for the end user. The next sensor is the IMU for collecting the acceleration, orientation, and velocity of the car. This is useful for collecting data when the car is driving in various environments, especially offroad. One IMU sensor is able to collect the data and we choose to directly solder it on the PCB board. Lastly, we used one hall effect sensor to calculate the RPM of the car. The hall effect sensor was placed directly on the rotating motor with a magnet.

7.9 Sensor Layout

Once the sensors were wired to receive data on the state of the car, the temperature sensor, ultrasonic sensor, hall effect, and IMU sensor were placed on the car, as seen in Figure 7.11. The five ultrasonics' unique layout can be seen in Figure 7.12. below. The reason for this layout is due to the ultrasonics sensor ability to detect objects. One ultrasonic sensor can only see

up to a range of 15 degrees, the way that they are placed in the bumper covers the entire front of the car and part of the side. Figure 7.2 shows the CAD model of the bumper where the front ultrasonic sensors were positioned. As seen in the bumper image, there are three ultrasonic sensors pointing directly straight and there are two other ultrasonics that are positioned at an angle of 45 degrees to improve the object detection range. Ultrasonic sensors 6 & 7 are placed on the side to ensure that the object has been fully cleared and that there is nothing the car is potentially bumping into driving past it. The camera is mounted elevated above the ultrasonic sensors and angled downward to be able to see and read the lanes. The IMU and PCB are positioned in the middle of the board. The reason for this is because we want an accurate orientation of the car, therefore keeping in the center provides the most accurate readings. The temperature sensors are adjacent to what was listed above, the motor and batteries. Lastly, the hall effect sensor is placed adjacent to the motor for measuring the RPM.

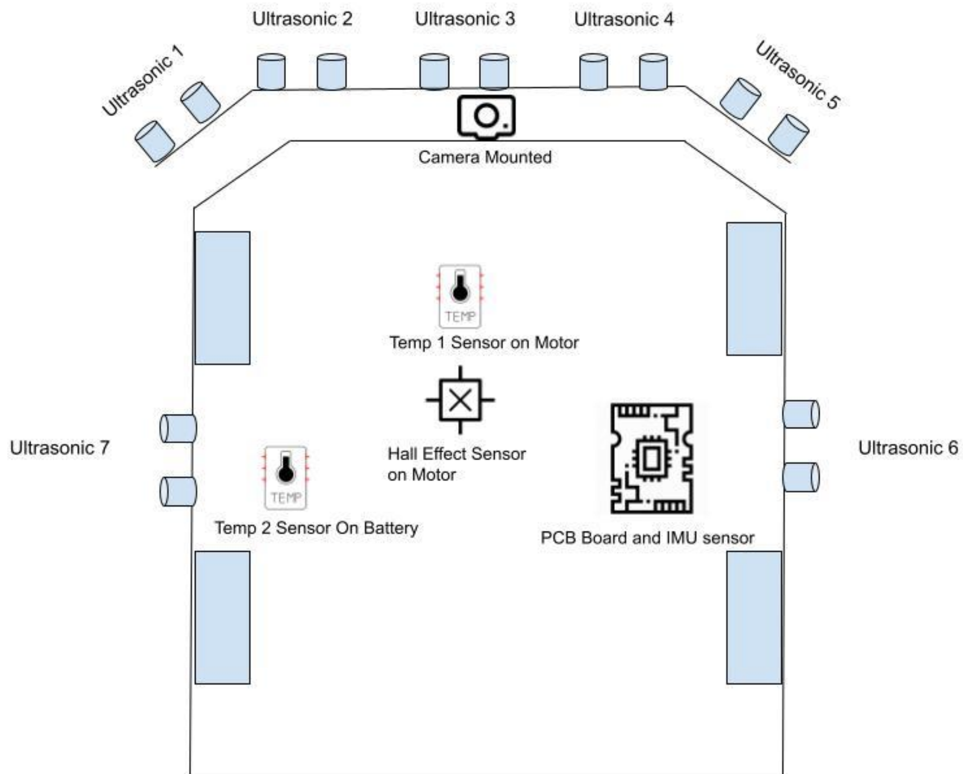


Figure 7.11: Sensor Layout in an RC car

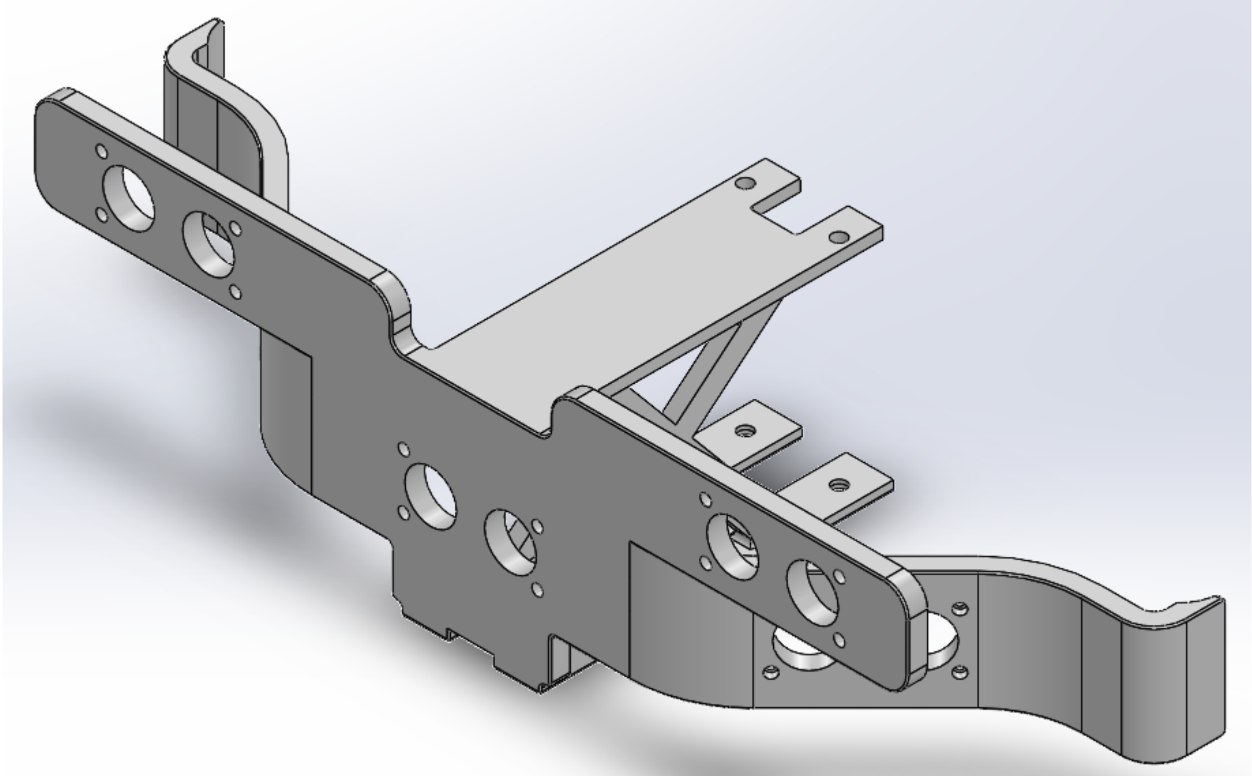


Figure 7.12: The CAD Model of the Ultrasonic Sensors Bumper

7.10 Sensors Integration and Block Diagram

The final sensors' block diagram can be seen below in Figure 7.13. Starting from the Elegoo and the sensors, the sensors mentioned above in the Sensor Layout section of this chapter are wired to an on car breadboard (or printed circuit board). From there the sensor data is compiled in the Elegoo's software platform and sent serially to the Raspberry Pi. The Raspberry Pi takes three other inputs; the camera for lane following, the portable battery as a power supply, and the server for user inputted driving conditions. From these inputs the Raspberry Pi controls the Servo Shield. The Servo Shield then sends those controls to two outputs; the Steering Servo for steering and the ESC for velocity of the car. The ESC also requires a Lipo Battery to have a battery sensor for the car. This makes the Raspberry Pi the main processor responsible for providing the commands to the mechanical parts of the car. The temperature data, acceleration, and orientation data gathered from the sensors by the Elegoo Mega will be sent to the Raspberry Pi, which in turn will forward it to an online UI dashboard alongside the camera's live feed. All of these components combined create MPAD.

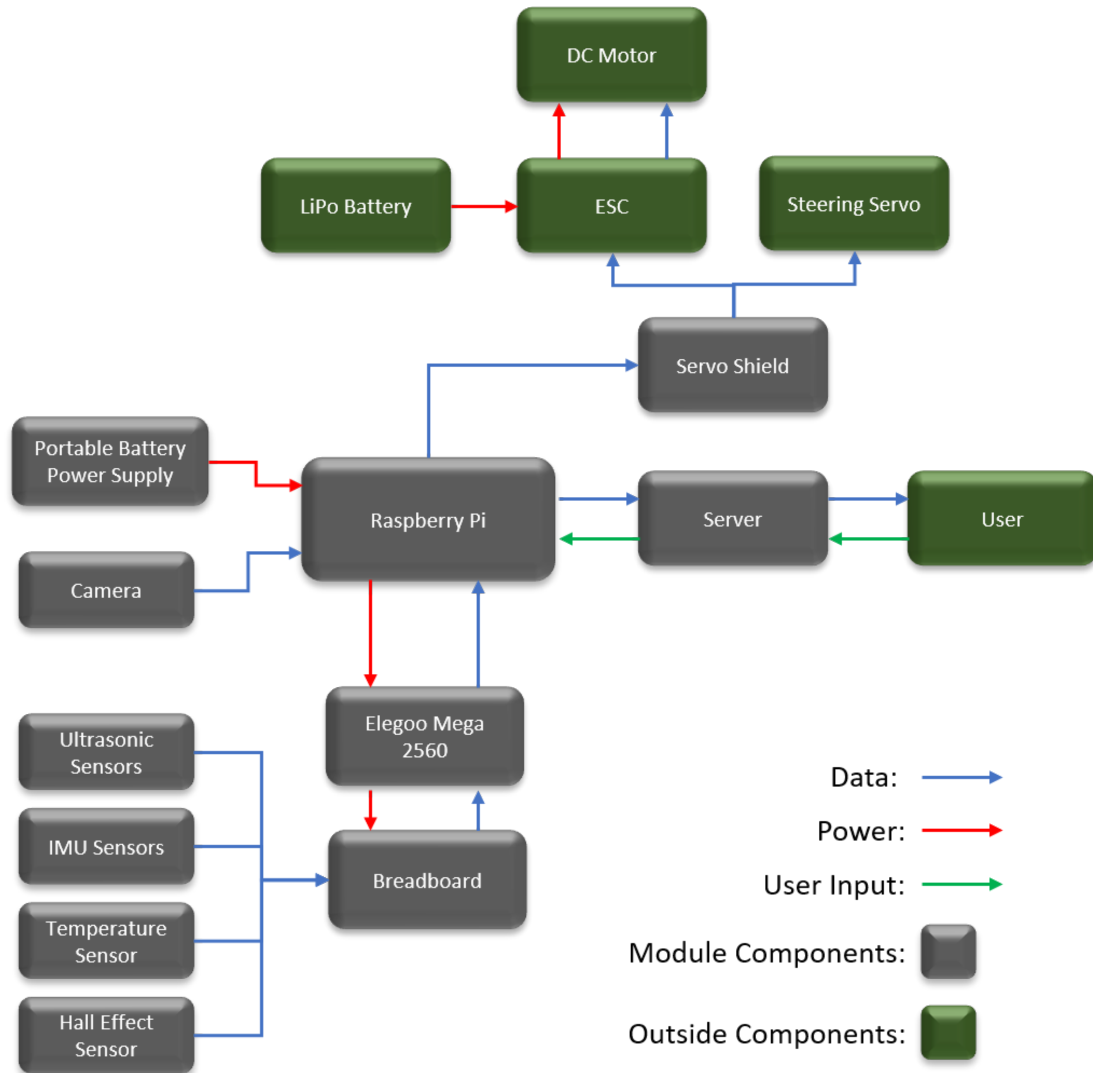


Figure 7.13: System Block Diagram for MPAD

7.11 Conclusion

In this chapter of the report the team went over the sensor choices, selections, layout, and implementation into a sensor package. The reasoning for the specific sensor types and models were provided, with specifications provided for each sensor. The sensor package contained two temperature sensor (one near the battery and another close to the motor) to avoid overheating issues, one IMU to allow for drivability through ramps , one hall effect sensor to know the rpm

of the motor, and seven ultrasonic sensors (one in each side and five in the front) to enable the obstacle avoidance feature of the car. The sensor package was an essential part of the fully integrated MPAD providing sensor data for display in the dashboard and allowing for features enabled by the AI aspect of the project. All of this information can also be found in the main guides for sensor setup, which are the Sensor and AI Setup Guide and the Arduino IDE Setup Guide. The next chapter will dive deeper into the implementation of autonomous driving and will cover the AI driving aspect of the project.

8. Autonomous Driving Implementation

This chapter discusses the implementation of the video capture, lane following, and obstacle avoidance. Autonomous driving is at the core of the project so it is important for the implementation to be clear and easy to test. Python3 is used for the entirety of the driving code.

8.1 Components Used

The components used during the implementation of autonomous driving are as follows:

- A Raspberry Pi 3 B+ / Raspberry Pi 4
- A 3-wire Servo
- ESC (XR10 JUSTOCK SPEC ESC)
- Adafruit Servo Driver PCA9685
- Raspberry Pi Camera Module
- 7 Ultrasonic Sensors4245749798
- LiPo Battery
- USB Portable Charger

A visual system block diagram can be found in Figure 7.13 in Section 7.10. A detailed description of the circuitry and the setup (including the camera) can be found in Appendix B: Sensor and AI Setup Guide.

8.2 Autonomous Driving Overview

Autonomous driving is a complex and multi part issue, this is why this chapter will provide an overview of things to come. Figure 8.1 contains a code block diagram outlining the lane following process.

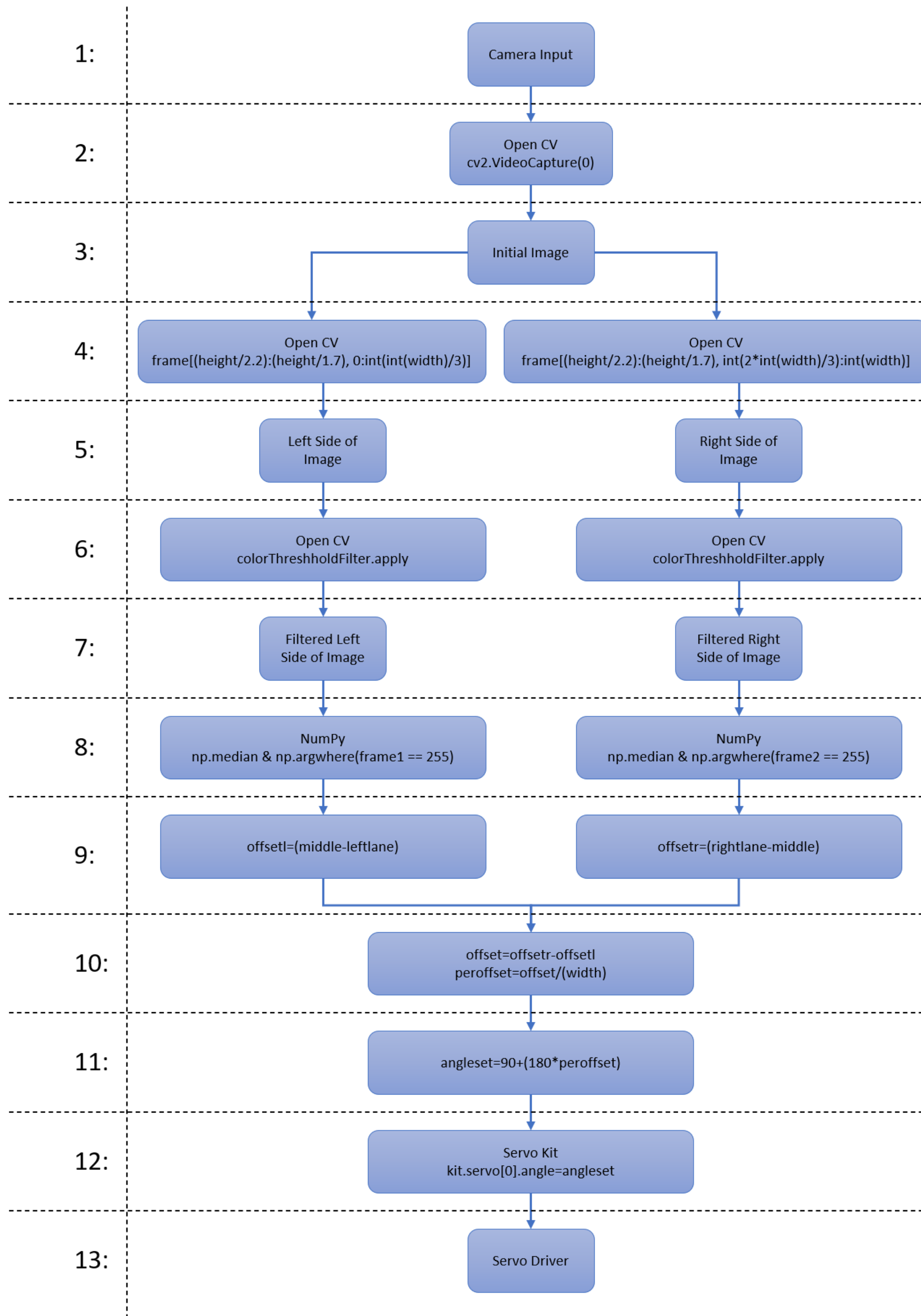


Figure 8.1: The flow diagram for lane following

- Block 1 represents the physical camera attached to the Raspberry Pi. It is covered in chapter 8.4 Video Capture.

- Block 2 is the specific OpenCV function used to begin recording and capture images. It is covered in chapter 8.4 Video Capture.
- Block 3 represents the image file saved with the Raspberry Pi. It is covered in chapter 8.4 Video Capture. The image can be found in Figure 8.3.
- Block 4 represents the splitting and the cropping of the initial camera input. It is covered in chapter 8.5 Image Processing. The image can be found in Figure 8.5.
- Block 5 represents the two split images. It is covered in chapter 8.5 Image Processing. The image can be found in Figure 8.5.
- Block 6 represents the color filter being applied to the two images. It is covered in chapter 8.6 Lane Detection. The example image can be found in Figure 8.6.
- Block 7 represents the two black and white images post color filter. It is covered in chapter 8.6 Lane Detection.
- Block 8 represents the NumPy functions used to gather the coordinates of the white pixels. It is covered in chapter 8.6 Lane Detection.
- Block 9 represents calculating the distance of the lanes from the center of the image. It is covered in chapter 8.7 Lane Following.
- Block 10 represents comparing the two distances from the lane to calculate which direction to go. It is covered in chapter 8.7 Lane Following.
- Block 11 represents transforming the turning direction from percentage to angle form. It is covered in chapter 8.7 Lane Following.
- Block 12 represents sending the angle information to the servo driver. It is covered in chapter 8.3 Servo Motor and Brushed/Brushless Motor Control.

- Block 13 represents the physical Servo Driver which controls the Servo Motor. It is covered in chapter 8.3 Servo Motor and Brushed/Brushless Motor Control and in chapter 8.1 Components Used.

8.3 Servo Motor and Brushed/Brushless Motor Control

The servo motor controls the steering of the front wheels. A brushed or brushless DC motor controls the back wheels. The servo and the motor is controlled using Python with the Adafruit_servokit library[29]. The Raspberry Pi is connected to the servo driver which then communicates the driving instruction to the servo and the ESC.

The Servo Driver PCA9685 is initialized with:

kit = ServoKit(channels=16)

The throttle is is controlled by the function:

kit.continuous_servo[1].throttle = X

Where X is a number between -1 and 1. The sign decides the direction of rotation for the motor and the magnitude decides the speed. -1 would be a full speed reverse and a 1 would be full speed forward.

The angle of the wheels is controlled with the function:

kit.servo[0].angle = Y

Where Y is a number between 0 and 180. 0 is the left-most turn and 180 is the right-most turn. 90 is straight ahead. Figure 8.2 is a visual representation of the possible values.

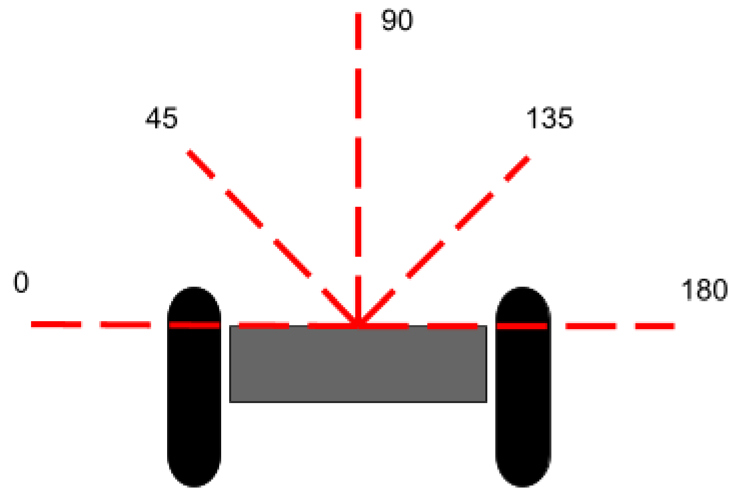


Figure 8.2: Sketch of Steering Angles

8.4 Video Capture

The camera is the primary input for detecting lanes which makes it the single most important sensor for autonomous driving. Because a camera has a wide field of view and captures content in color, it is ideal for aiding a self-driving algorithm in a cost effective way. Mounted at a 60 degree angle relative to the face of the car, the camera captures the majority of the car's forward perspective at roughly 8 inches above the ground. Video capture is done utilizing the OpenCV python library using the Raspberry Pi Camera Module[21]. There are many ways of capturing video on a Raspberry Pi but given the use of OpenCV in the image processing, it is the best method for video capture for MPAD. This is because of OpenCV's extensive function library that reduces the amount of code necessary for understanding the contents of a given image. The video capture is at 600x500 pixel resolution at 30 frames per second using the Raspberry Pi Camera Module. Each frame acts as an image for the purposes of the driving code. The entirety of the driving code is within an infinite loop to allow the car to run indefinitely. Each loop analyzes a single frame for lanes and determines which direction is the most effective

for moving the car forward without running into obstacles. This is done due to the simplicity in processing a single frame. If the algorithm can determine which way to steer given a single image, repeating the process over and over allows the car to steer towards the end of the course without using complex spatial processing tools. Figure 8.3 shows an example frame.

The video is captured using the following function:

cap = cv2.VideoCapture(0)

Each frame is stored as an image using the following function:

_, frame = cap.read()

Figure 8.3: A Frame from Raspberry Pi Camera Capture

Upon capturing each frame, the code passes it through a series of filters and equations that determine which direction the car should steer towards.

8.5 Image Processing

Image processing is used for splitting the frame into multiple parts and removing irrelevant sections of the image. Image processing continues the use of OpenCV. In terms of cropping and coordinates (0,0) is the top-left corner of the image. The values increase as you move down and to the right of the image. This can be seen in Figure 8.4.

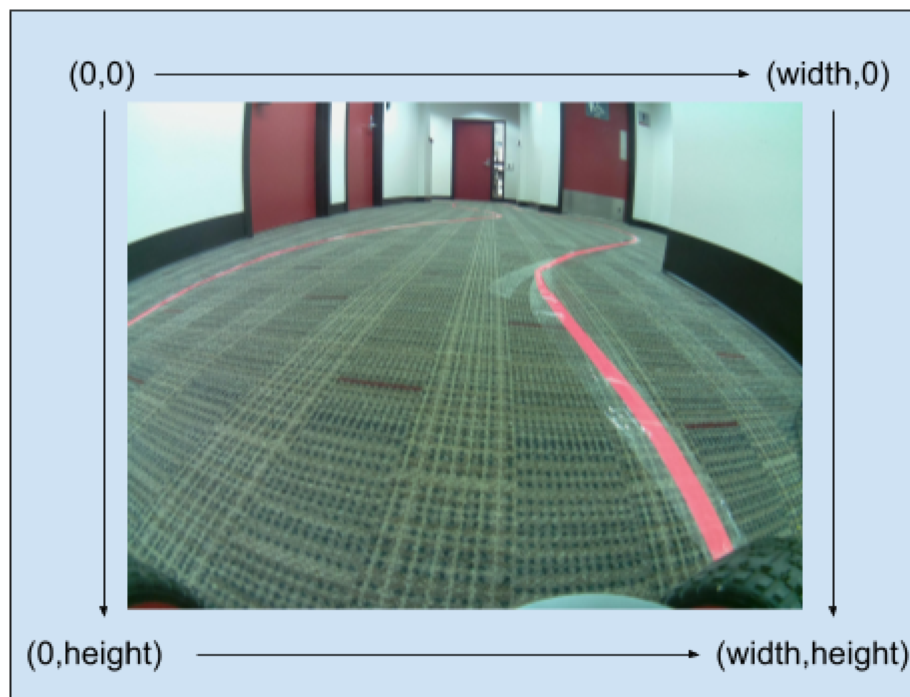


Figure 8.4: Coordinates of a Frame

The following code stored the attributes of the image as integers:

height, width, channels = frame.shape

The *height* variable is how tall the image is in pixels and width is how wide the image is in pixels. In our case *height* is 500 pixels and *width* is 600 pixels.

The horizontal middle of the image can be calculated using:

middle = width / 2

The following code splits the image in two and crops the image:

frame1init = frame[int(height/2.2):int(height/1.7), 0:int(int(width) / 3)]

*frame2init = frame[int(height/2.2):int(height/1.7), int(2 * int(width) / 3):int(width)]*

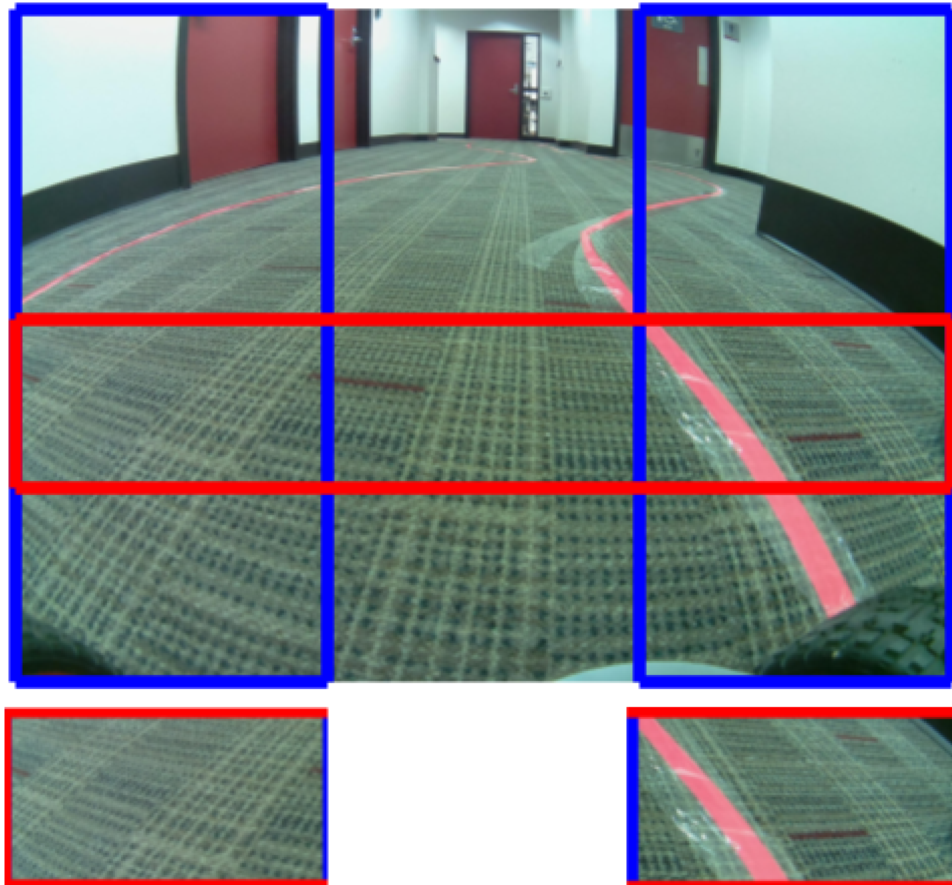


Figure 8.5: Image Cropping Visualized

Figure 8.5 above shows a visualization of the different cropping areas for the camera input. The blue lines represent crops determined through the X-axis and the red lines represent the crops determined through the Y-axis. The final results of the crops is shown below the image.

frame1init crops the original image into the leftmost third, this is what "*0:int(int(width) / 3)*" means where 0 signals the leftmost side of the image as the beginning of the crop and "width/3" signals the end of the crop as a third of the image starting from the left.

frame2init crops the original image into the rightmost third. This is the leftmost blue box in Figure 8.5.

"*int(2 * int(width) / 3):int(width)*" means where "*2*width/3*" signals the left side of the last third of the image as the beginning of the crop and "int(width)" signals the end of the crop as the rightmost side of the image. This is the rightmost blue box in Figure 8.5.

The area to crop in-between "*height/2.2*" and "*height/1.7*" was determined through testing. This area gives a comprehensive view of the lanes while cutting out most of the noise. Having a small area to focus on is important as it decreases the likelihood of getting false positives in lane detection. This is the red box in Figure 8.5.

8.6 Lane Detection

The primary code of the lane following is to stay in the middle of the two lanes. The two sides that were split earlier are processed separately. The first step of the lane following is finding the location of the lanes in relation to the car. This is done by applying the color filter to the two images (*frame1init* and *frame2init*). The libraries used in this part of the code are OpenCV (cv2) , Numpy (numpy), and Adafruit Servokit (adafruit_servokit).

The color filter is applied using the following functions:

frame1 = colorThreshholdFilter.apply(frame1init, [86, 95, 153], [100, 253, 216])

frame2 = colorThreshholdFilter.apply(frame2init, [86, 95, 153], [100, 253, 216])

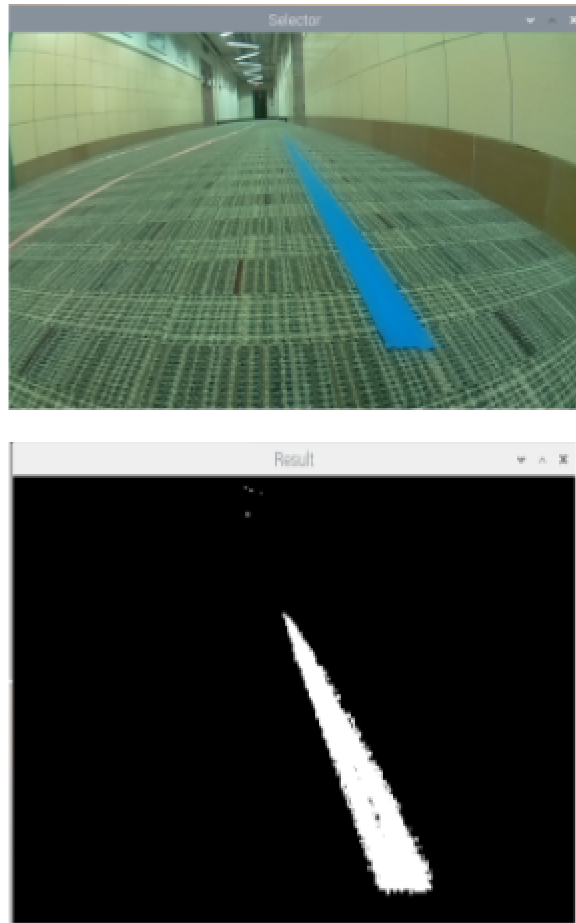


Figure 8.6: Color Filter Results of a Single Lane

frame1 and *frame2* are black and white versions of *frame1init* and *frame2init*. Every pixel within the given RGB values is made perfectly white and every other pixel is made perfectly black. From here knowing the locations of the white pixels will give us the locations of the lanes. This can be seen on Figure 8.6 where the color filter is applied to a single lane.

The following code returns the x and y coordinates of every white pixel within an image as an array:

```
np.argwhere(frame1 == 255)
```

np.argwhere(frame2 == 255)

The y coordinates of the lanes are not actually needed so we can just get the horizontal location of the pixels using the following function:

[coordinate[1] for coordinate in np.argwhere(frame1 == 255)]

[coordinate[1] for coordinate in np.argwhere(frame1 == 255)]

This is still an array so we use the following code to get the lane positions as a single integer:

leftlane = np.mean([coordinate[1] for coordinate in np.argwhere(frame1 == 255)])

*rightlane = (2 * int(width) / 3) + np.mean([coordinate[1] for coordinate in np.argwhere(frame2 == 255)])*

By adding "*2*width/3*" to the coordinates of the pixels we make it align with the original image format. This is because the earlier crop 0 coordinate on fram2 would be "*2*width/3*" on the initial frame.

The mean of the x coordinates of the image on the left is determined to be the position of the left lane and the mean of the x coordinates of the image on the right are determined to be the position of the right lane. Figure 8.7 show a simplified visualization of the Image Processing and Lane Detection

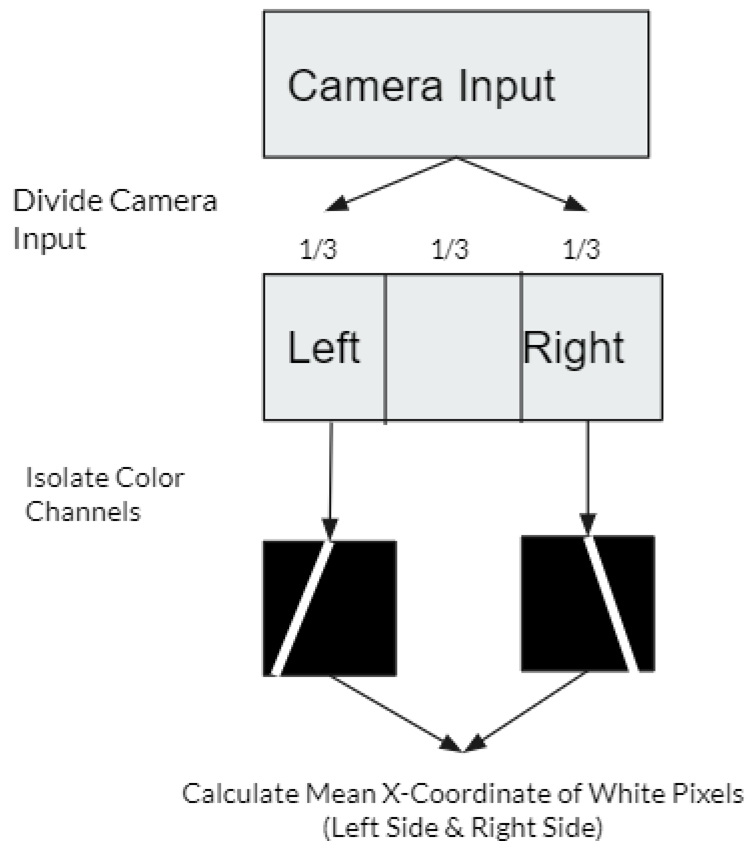


Figure 8.7: Lane Detection Flow Diagram

8.7 Lane Following

As the primary goal is the car staying in the middle of the lane it is important to know the distance of each lane to the middle of the car. Getting to the middle of the lane is done by attempting to equalize the car's distance to both lanes.

First, we must take the difference between the middle of the camera input and the position of the lane within the image to determine how far away the car is to a lane at any given time. This can be done using the following code:

offsetl = (middle - leftlane)

offsetr = (rightlane - middle)

Note that if a lane is not detected at all the distance is assumed to be as far as the camera can see. The following code does that:

```
if np.isnan(offsetr):
    offsetr = middle
if np.isnan(offsetl):
    offsetl = middle
```

For example, given the two final cropped images in Figure 8.5 the *if np.isnan(offsetl)* code would trigger. This is because the cropped area on the lane does not contain a lane. The resulting angle would be a sharp left.

The difference (distance) of the left lane is then subtracted from the difference (distance) of the right lane. This is called the offset. This determines where the car should be going to stay in the middle of the lane. The code is:

```
offset = offset - offset
```

The values is still in pixels so we turn it into percentages using:

```
per offset = offset / width
```

The sign of a number determines which side to turn and the magnitude of a number determines the sharpness of the turn. The transformation from the pixel to pixel equation into a percentage so the lane following code is important so the code is not resolution specific. What each angle corresponds to can be found in the earlier Figure 8.2.

Finally, the angle is set using the following code:

```
angleset=90+(180*peroffset)
```

The “+90” is there since an angle of 90 means that the steering is looking right ahead. There is still some supplementary functionality, however. Depending on the design of the car, having a

manual lower and the upper bound is important as to not damage the steering or get stuck. Figure 8.8 shows an example. The following code achieves sets the bounds:

```
if angleset < 40:
    angleset = 40
if angleset > 140:
    angleset = 140
```

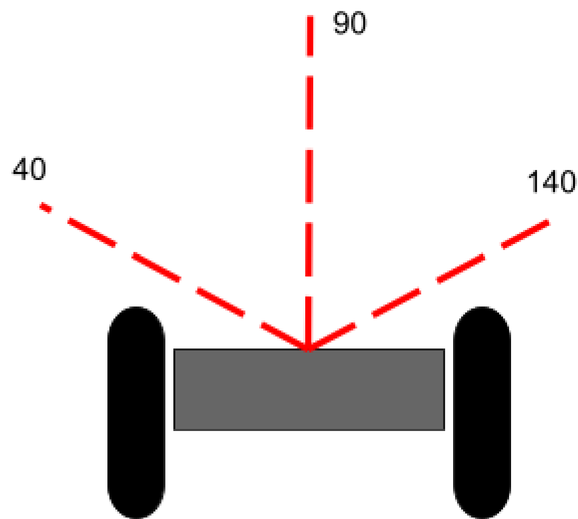


Figure 8.8: Steering Angle with Hard Boundaries

8.8 Sensor Reading in Raspberry Pi

The Arduino (Elegoo Mega2560) and the Raspberry Pi are connected through a serial USB-B to USB-A connection at a baud rate of 9600. This connection allows for a single data packet to be read in on the Raspberry Pi with every sensor reading included. This means less individual data transmissions have to be sent, resulting in less time waiting for a serial read from the buffer on the Raspberry Pi. This read is achieved by using Python's built in Serial library.

The initialization is completed by:

ser = serial.Serial("/dev/ttyACM0", 9600)

Subsequent reads of the data are completed by calling the readline() function:

read_ser=str(ser.readline())

The resulting byte string is encoded to UTF-8 before being broken up into smaller segments. An example string prior to being broken up might look like this:

temperature:92.1|imu-x:1.22|imu-y:0.2|imu-z:2.231|battery:99|humidity:23.8

Since this format is predictably formatted, we just perform a split on the string at each vertical pipe character '|':

readings_dict = read_ser.split('|')

Lastly, each reading in the dictionary is iterated over to make the sensor title and reading a key/value pair:

```
output_dict = {}
for reading in readings_dict:
    kv = reading.split(':')
    output_dict[kv[0]] = kv[1]
```

This dictionary can now be sent over to the server hosting the user dashboard for rendering, with each value only being updated if a reading is found from the Arduino. This ensures that values are not hardcoded in the event of a sensor being damaged, or even just a student wanting to swap out which sensors are in use for a given car.

8.9 Ultrasonic Image Calibration

As obstacle avoidance is a fusion of ultrasonics sensors and the camera it is important to know the pixels in the image where each ultrasonic sensor corresponds. This is to be able to

compute driving angles with both the lane and the obstacles in mind. We need to know the location of the pixels for every given cm in the ultrasonic reading.

A practical calibration test needs to be performed once the ultrasonics are installed. The test must take place on an empty flat surface at least 1 meter long and 50 cm wide. The car is placed on one end of the surface looking inward. A small object that is still tall enough to be picked up by the ultrasonic sensors is placed in front of the car. The object is moved around carefully making sure that the only thing triggering the ultrasonics is the object itself. A sketch of this process can be seen in Figure 8.9.

The object is clicked on giving the pixel coordinates of the click. This is useful for finding the minimum and maximum of the pickup cone of the ultrasonic. The final result of moving the object around is a y=mx+b equation where giving the ultrasonic distance as x will be the object coordinates as y.

For example for our project the equation for the min and max pixels of ultrasonic number 2 were:

us2min=[0.61142,155.6]

us2max=[1,99999,181.4]

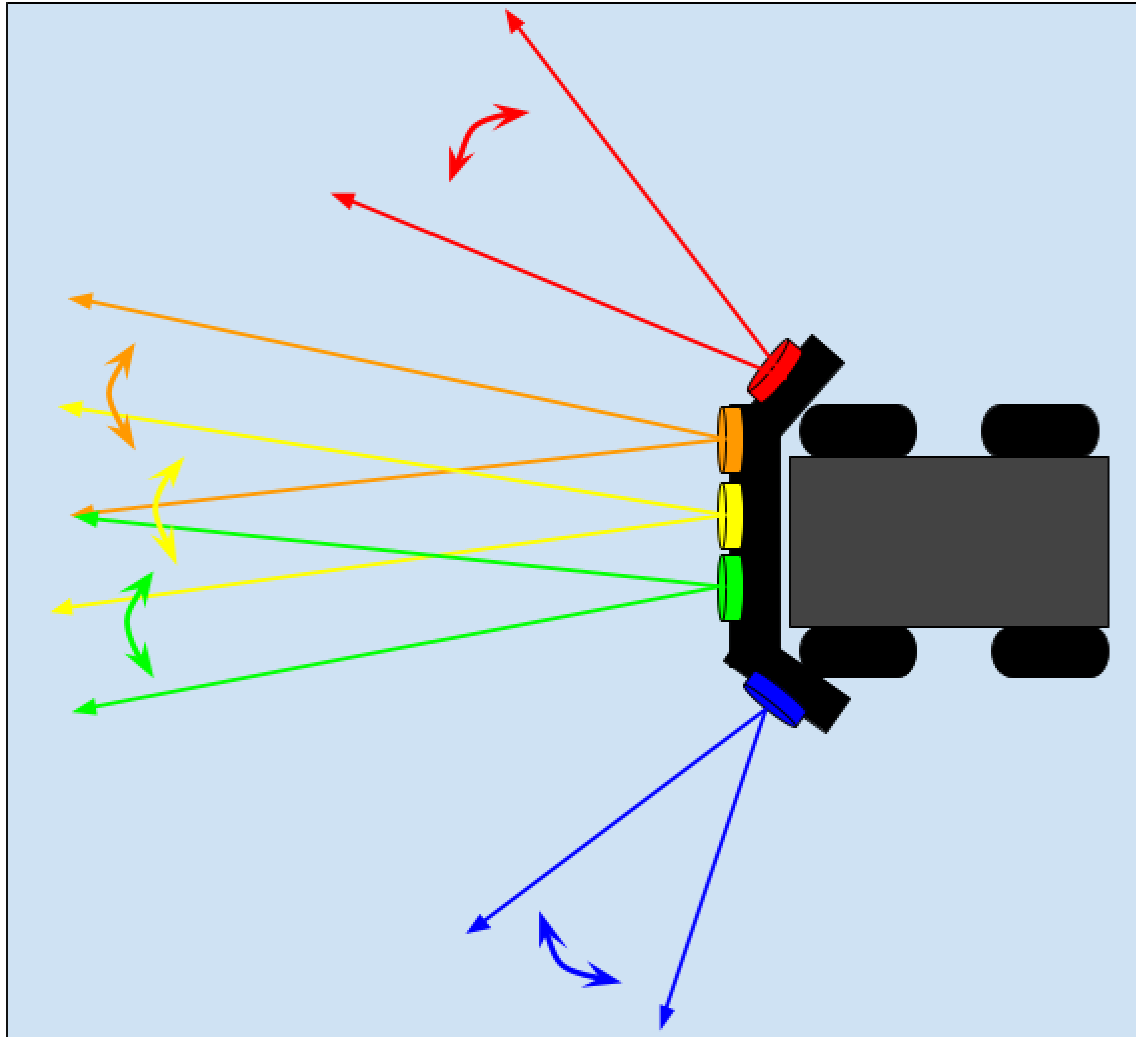


Figure 8.9: Sketch of Ultrasonic Sensor Pickup Cones and Calibration Test

8.10 Obstacle Avoidance

Obstacle avoidance builds upon the lane following. This means that obstacle avoidance will still follow lanes. This is important as this feature alone changes obstacle avoidance from simply comparing the ultrasonic values to a feature where each given ultrasonic value has to correspond to a location within the camera input. This is also why ultrasonic calibration is important.

For our given car, we decided that 60 cm for the front ultrasonic sensors was a good pickup range for obstacles. Due to the inherent horizontal inaccuracy of the ultrasonic sensor and longer range would not provide useful data, or worse, provide false positives. 40 cm was picked for the bumper side ultrasonics as due to their angle and further readings would be outside the lane. The following code take place in between the “if np.isnan(rightlane):” and offset=(middle-leftlane):

```
if u2<60:
    leftlane =max(leftlane,int(us2max[0]*u2+us2max[1]))
```

Functions like these supersede the lanes if the obstacle detected is closer to the middle of the screen. “int(us2max[0]*u2+us2max[1])” here gives us the rightmost value of the obstacle considering the ultrasonic inaccuracy. If this value is higher than the existing leftlane value from the lane detection then (as a higher value is more to the right thus closer to the middle when looking at the left lane) it is deemed as the left lane and lane following continues on as before. This functionality is applied to all of the ultrasonic besides the middle one. One difference is that in the case of the rightward ultrasonics instead of looking at the higher value we look at the lower value:

```
if u4<60:
    rightlane = min(rightlane,int(us4min[0]*u4+us4min[1]))
```

There are also two cases where the code ignores everything and hard sets the offset. These are when an obstacle is too close to the side bumper ultrasonic sensors:

```
if u1<25:
    leftlane =middle
if u5<25:
```

```
rightlane = middle
```

This exists as a failsafe in case the ultrasonics do not pick things up beforehand. There is always some latency so a less extreme response might still make contact. The rest of the time the side bumper ultrasonic sensors function like the others:

```
if u1<40:
    leftlane =max(leftlane,int(us1max[0]*u1+us1max[1]))
```

There is also some tiebreaker functionality that is implemented before the lane selection happens. This is important as usually more than one ultrasonics can pick up an obstacle. The functionality is in the following code:

```
if u1<40 and u2<60 and u4<60 and u5>40:
    u4=100

if u1>40 and u2<60 and u4<60 and u5<40:
    u2=100

if u1<40 and u4 <60:
    u4=100

if u5<40 and u2<60:
    u2=100

if u2<60 and u4<60:
    if u2<u4:
```

```
        u4=100
    else:
      u2=100
```

In these cases, setting an ultrasonic value to 100 means that it will be ignored by the rest of the code. There is also a final use for the ultrasonic sensor and that is breaking. It is just an if-else function that checks to see if any ultrasonic sensor readings are below 15 cm and stops the car.

8.11 Conclusion

In this section, we shared specific details of how the self-driving algorithm works, including the past iterations that led us to our end result. The general flow can be broken down as follows: First, the camera input is read and captured as a single frame. This frame is then run through a color filter in order to isolate the specific color that matches the 'road's' lanes. From here, the algorithm looks at the bottom half of the image to determine which side of the screen has the least lane color in it. Because this side is assumed to be furthest to the car, the servo is set to steer toward it. By looping this process hundreds of times a minute, we achieve autonomous driving in a way that is minimally taxing on the Raspberry Pi's CPU. However, driving alone is only a small part of this project. To tie it into the greater system, we have enabled control of this self-driving algorithm from a web dashboard. This dashboard is covered in the next chapter in detail.

9. Dashboard Implementation

This section will discuss the goals and implementation of MPAD's web application. The main goal was to create an intuitive driving dashboard to allow remote access to an RC Car. The dashboard was intended to provide a user with real-time driving controls and a display for sensor data. Through this platform, students can simultaneously remotely control their cars over the WPI network, without any prior Computer Science knowledge or skills. All features are designed for ease-of-use, but the web application explains what each tool does on the landing page for thoroughness. And although the web application is publicly accessible, only a car configured with the team's self-driving module will be able to communicate and register to our application. To implement this in the ME 2300 course, the CS and ECE teams have pre-configured all of this for the students. The rest of this chapter will discuss the dashboard's implementation, architecture, and features in detail.

9.1 AI Design

Modularity extends beyond the physical components of the car. To be considered fully modular, the various software components need to be abstracted out to a level that works independently of the other parts. To achieve this, the software was written in separate sections that can quickly interface with each other without concern about the underlying implementation. For instance, the web server will send a start/stop command to the car, but the car's self-driving program is what decides what happens with the controls. This means our team could develop each part in parallel.

9.2 MPAD Web Application

The team developed a web application to provide real time controls for students to easily manage their RC car. After assembling the sensor package, the student can simply navigate to the MPAD Web Application's unique URL on any web browser to view our documentation and access their car's driving dashboard. When an RC Car's Raspberry Pi is powered on, it will automatically register to our server and begin streaming camera and sensor data. After approximately one minute and thirty seconds of startup time, a student is able to see this registration occur on the main page of the web application. By clicking on their car, they are able to monitor and export sensor data, watch the camera feed, and control their car's driving in real time.

9.2.1 Technology

The team's main goal for this web application was to make it intuitive for the user and simple on the backend. This project is to be used in a 2000's level Mechanical Engineering course, so the student's should not need any prior training or explanation to consume their time. All instructions and controls should be straightforward on the user interface. Additionally, because this is an ongoing project, the web application code needs to be simple enough for additional development and maintenance in any future MQP. To achieve this, we developed a python application using the Flask web framework[30]. For reference, Flask is a simple but extensible web framework for Python applications. The MPAD Web Application uses the Flask framework to define our application and interpret web requests. The frontend was designed using HTML, javascript, and vanilla CSS for simplicity. Finally, we made sure the frontend libraries are open source and well-documented. The team only used two: ChartJS[31] for the dynamic

graphs that log sensor data and SocketIO[32] for the websocket and streaming services. ChartJS is an open-source javascript

9.2.2 Functionality

Figure 9.1 shows a picture of what a user sees after a student has turned on their car and selected it from the landing page.

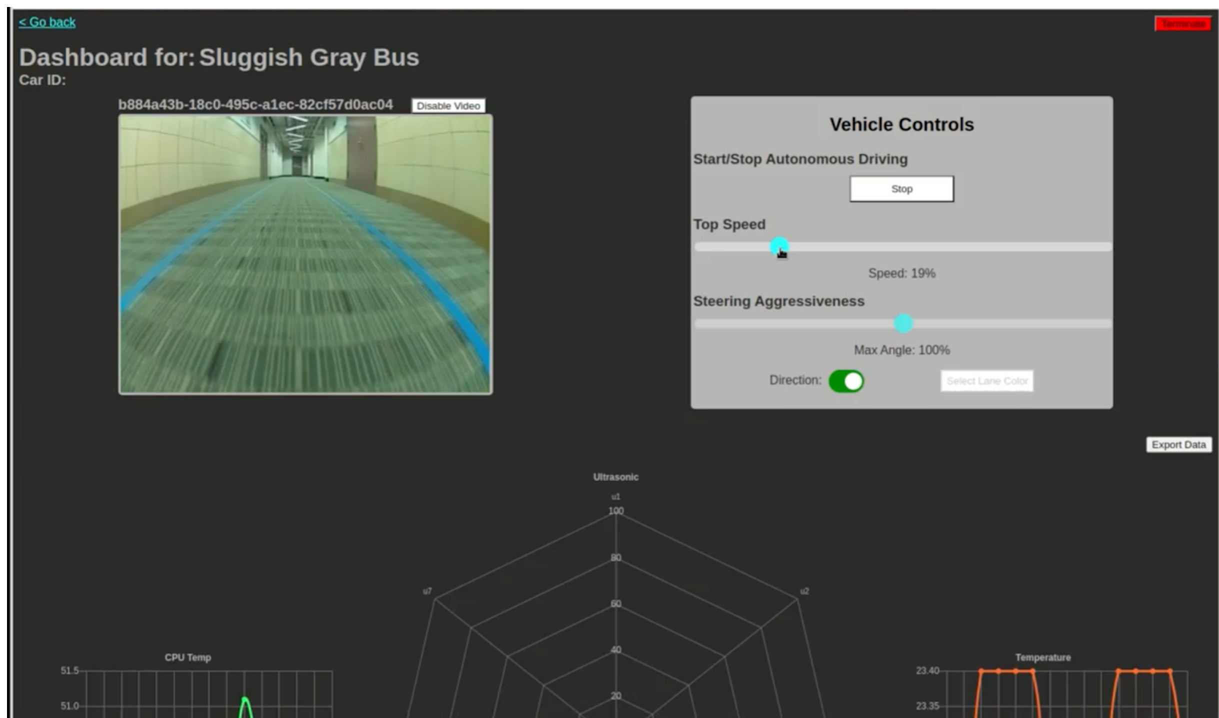


Figure 9.1: Driving dashboard on MPAD's web application

9.2.3 Vehicle Controls

As listed and described on the landing page, the following features are all configurable for the user on the front end, as shown in Figure 9.2:

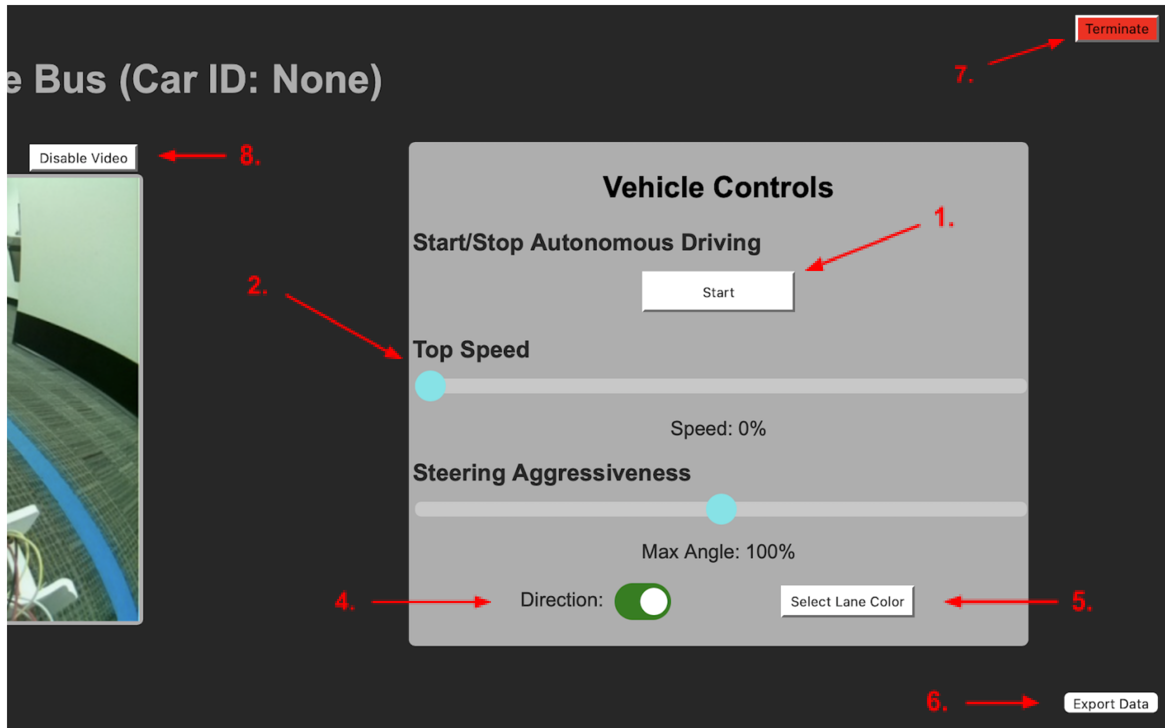


Figure 9.2: Control panel on web dashboard, annotated for each feature

1. ‘Start/Stop’ - the command that signals a vehicle to start/stop autonomously driving.
2. 'Top Speed' - change your vehicle's speed from 0-100 percent of full throttle. The team placed a warning on this option, as driving at high speeds can cause mechanical damage if the car crashes.
3. 'Steering Aggressiveness' - a slider to change the car’s steering aggressiveness from 50-200 percent of normal steering. This is to help accommodate cars that consistently over/understeer.
4. 'Direction' - a toggle button to change the direction a car will drive upon ‘Start’. This feature is necessary to accommodate motor placement.
5. 'Lane Color Selection' - a tool that allows a team to set the color of the lanes that the car needs to follow. View the following section for more information.

6. ‘Export Data’ - downloads a .csv file containing all collected sensor data to the user's browser.
7. ‘Terminate’ - shutdown the Raspberry Pi and terminate web connection.
8. 'Disable/Enable Video - a feature that toggles the video stream. The team added this feature since disabling streaming can improve the car’s driving performance. Because we are using a free version of Heroku for our server, multiple video feeds can create a backup of requests.

In order for the web application to function as a remote controller, each car is constantly listening for requests being made to them and sending sensor data to the server. All of the customizable features above are created with defaults when a car is registered to the server. To keep track of each car’s individual configurations, a unique id (UUID) is also generated and used as a key to access, edit, and send this data. When a user makes a change to any of the above features, the dashboard will send an appropriate API request to the server with the car’s id. The server will then retrieve the current configurations of the car, apply the requested change, and send the updated information to the car through the open WebSocket, which is created on startup. The car’s driving code utilizes a Streamer class that maintains all of the variables inside of the driving code which is subject to change by user input. So when the car receives the request from the server, the car settings can be adjusted even while the car is driving.

9.2.4 Color Selection Tool

The Color Selection Tool is accessible from a car’s dashboard. A key consideration of this project was how someone will have to set up the tracks for their car. The team only set a few requirements for track configurations, such as width and sharpness of turns, but wanted to leave

everything else up to the students. Therefore, MPAD should allow a vehicle to successfully drive a course of any color. In order to achieve this, the team developed a ‘Lane Selection Tool’ designed for a user to dynamically select the color of the lane and be able to view how the car sees it. To use this tool, a user must simply click on different parts of the lane in the frame on the left. Upon each click, the right frame will begin to highlight the lane in white. A user can be certain that the lane colors are set properly when there are no large gaps and there are clear white lanes in the right frame. Figure 9.3 below shows an example of this tool for lane selection. A live video demonstration has been provided in Appendix Q.

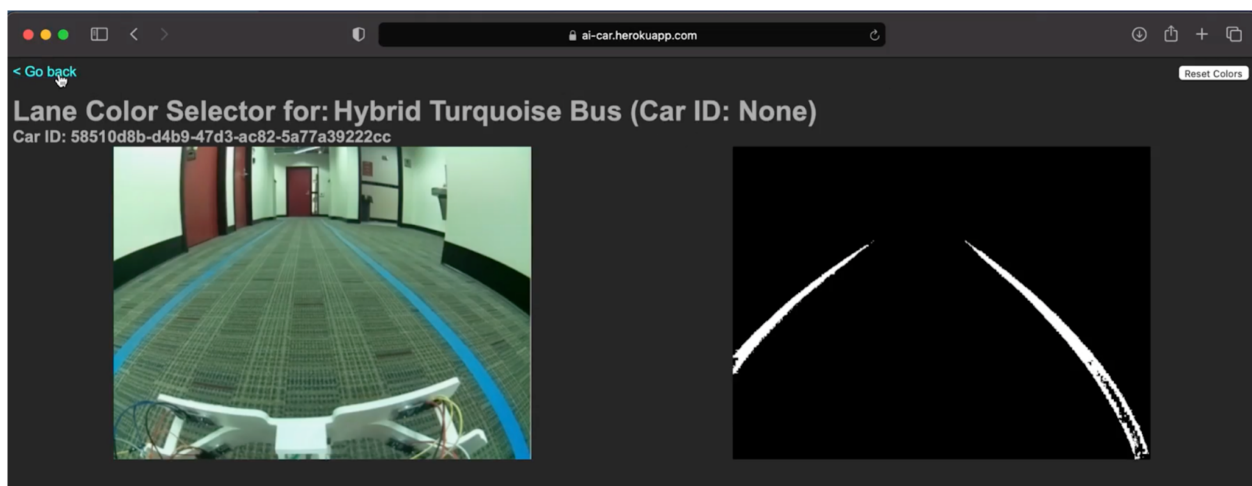


Figure 9.3: Lane color selection tool on the web dashboard

The color selection tool on the MPAD Web Application utilizes the same code logic as the initial Python color-finder script (Appendix D). However, the major difference is that the team had to rewrite parts of this tool in Javascript and integrate the response with the driving code. To understand why this tool is important, it is necessary to first know how the car actually follows lanes. The camera first reads the data in the HSV format (Hue, Saturation, Value), which is a means of breaking each pixel’s color into a set of three integers. The self-driving code utilizes two arrays of these HSV values, which we refer to as ’color channels’. One array

represents the lower color bound while the other represents the higher color bound. Using OpenCV, the camera is able to isolate only colors within these two ranges and ignore everything else. All pixels with color outside this range are translated to black (0) and all pixels with a color inside this range are made white (1). When a car is initialized, the lower channel is set to the maximum ([255, 255, 255]) and the higher channel is set to the minimum ([0, 0, 0]), meaning that the camera will process no image. When the user clicks on the lane, it will adjust a value in the lower channel array if the hue, saturation, or value is lower than what it previously was. Similarly, it will adjust a value in the higher channel array if the hue, saturation, or value of the selected pixel is greater than what was previously in the array. As the user clicks, they will see the lane start to form in white as the camera inputs these new HSV values. Because of this model, it is very important to select lane colors anywhere on the track where there is different lighting. Additionally, there is a button to reset the color channels if the user selects a pixel that they did not intend to. Any selection that is not on the lane will create noise and highlight background elements that will disrupt the driving algorithm since the color channels are output to the car and used in the driving code. The lane colors need to be set before the car starts driving and cannot be changed while the car is in motion.

9.3 Sensor Module Integration

Figure 9.4 below is a slightly simpler abstraction from Figure 7.13 in Section 7.10 to show the individual components and data flow within MPAD. Each color in the figure represents a different section in MPAD, as shown in the Key to the right.

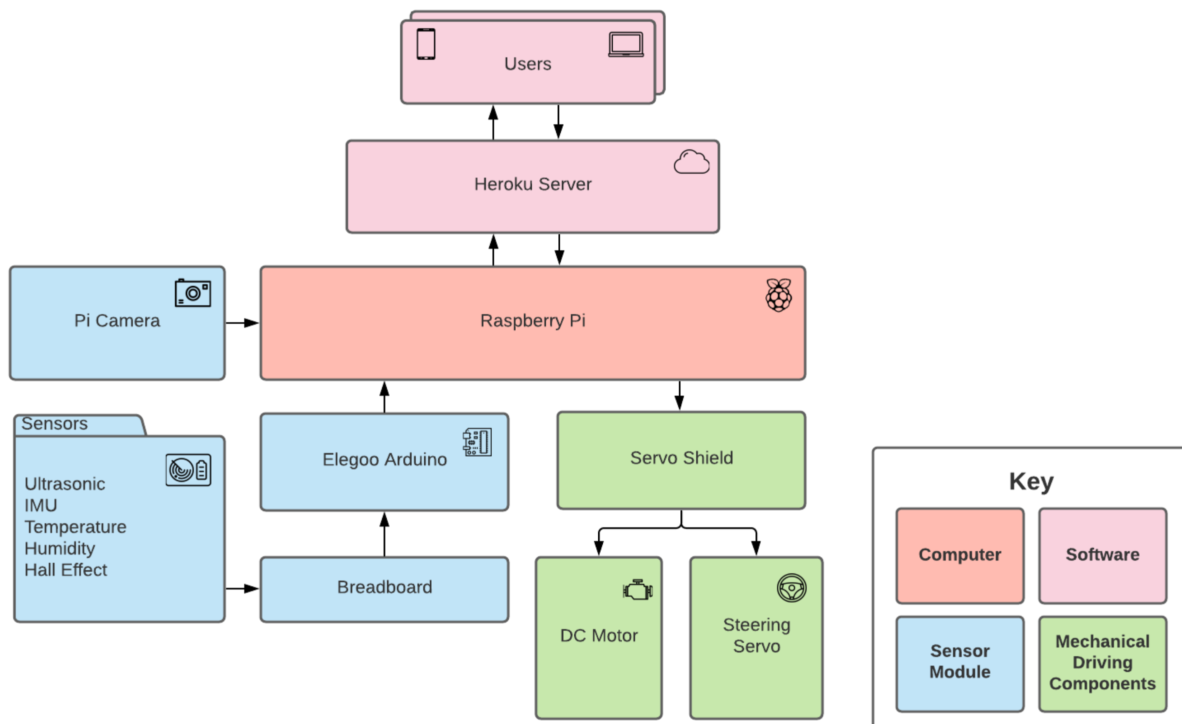


Figure 9.4: Abstracted system block diagram

Starting with the sensor module (in blue), the sensor data and camera feed are processed on the Arduino and forwarded to the Raspberry Pi, which then sends that information to the Users (in pink) on the web dashboard through the Heroku Server[33]. The Raspberry Pi can then receive driving requests from the Users and send it to the mechanical driving components (in green). For more information on how sensor readings are passed between the Raspberry Pi and Ardunio, view Section 9.6. In order for the Raspberry Pi to send this data to the server, the team developed a python script that bundles all of the data into a JSON format. JSON is one of the most popular data interchange formats. Once the data is parsed and packed accordingly, the Raspberry Pi actually sends a HTTP POST request to our server. Because this is not a production-level application, we do not have any security or passwords on these requests. This is highly recommended as future work to avoid any interference by someone trying to interfere with or overload the server with requests.

9.4 Application Architecture

The MPAD Web Application was specifically built to be lightweight. Figure 9.5 below shows the web application architecture with brief functional descriptions of the components involved.

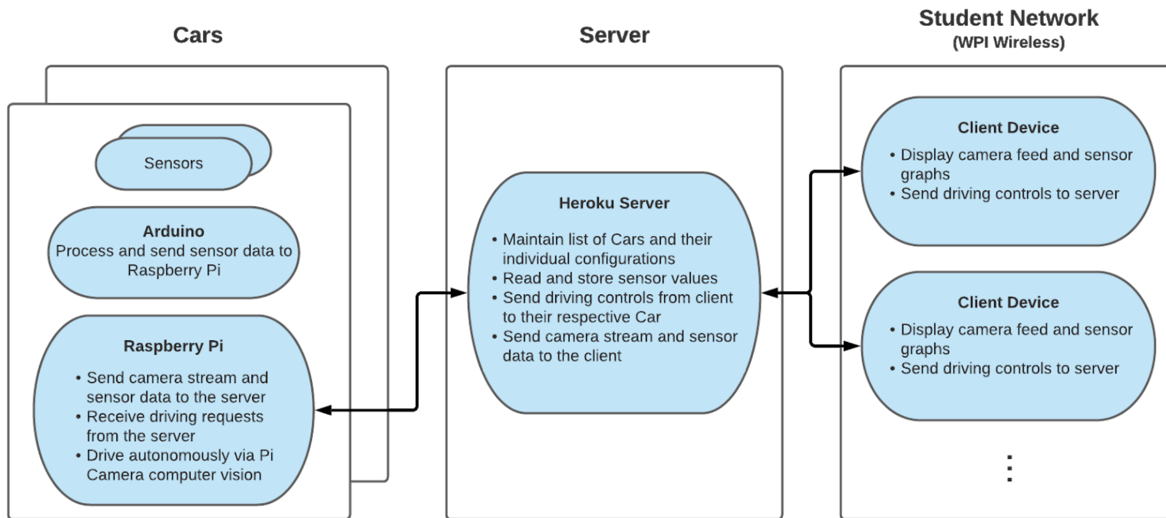


Figure 9.5: Web application system architecture

The layered rectangles on the left represent the actual RC Cars. Our application can handle any amount of cars but is limited by the server's request handling capabilities. Within the car, there are numerous sensors that are wired through a breadboard to an Arduino. The Arduino is responsible for processing all of the sensor output and sending the data to the Raspberry Pi through a wired connection. The Raspberry Pi is responsible for all of the car's driving and processing, which is why the team recommends using Raspberry Pi 4 (8GB RAM). Through testing, the team has seen significant differences in driving performance on Raspberry Pi 3's because it only has 1GB RAM. As latency increases, the more the car oscillates. At high speeds, this can cause the car to drive off the track. This is why the team recommends a Raspberry Pi with at least 4GB of RAM. In addition to running the driving code, the Raspberry Pi is

responsible for sending the sensor data and camera feed to the server and receiving driving requests from a user.

The team decided to use Heroku as a server. There are a lot of options for free servers, but we chose Heroku because it is simple to use, flexible, lightweight for small applications, and reliable. The major drawback with Heroku is the performance of its free version. Like all other cloud platforms, payment is necessary to scale an application for production. The free version still works for this project but is not able to handle more than a few cars at a time. The major role of the server in this project is to maintain all of the cars and their respective data and communicate back and forth between the cars and users. All communication and data go through Heroku.

For storage, the team decided to use Redis, which is a very useful Heroku add-on. For this project, it was necessary to have a storage system that could store large amounts of data and respond to requests as quickly as possible. Databases were considered but were not practical considering the number of requests that were going to be made and the type of data to store. Redis is an ideal solution since it is an in-memory key-value data store (similar to a cache) that is much faster than an external database alternative. Once a car is registered, the server stores all its information in Redis and updates it appropriately as changes are made. This is how lane color channels and sensor data will persist for as long as the car is connected. There is also a button on a car's dashboard to export all of the collected sensor data to a CSV file.

Finally, the rectangle on the right represents the Student network from which students can access the web application. The team has also pre-configured all of the Raspberry Pi's to be connected to the WPI-Wireless for the fastest speed and so that someone can remote into it easily if there is ever an issue. As shown inside the student network, there can be many client devices.

This way, multiple students can all watch their car's camera feed and sensor logging. The Web Application can be accessed through any web browser and works on mobile devices.

9.5 Startup Scripts

One of the most important parts of the self-driving module is for it to be as easy to use as possible. As mentioned in the Sensor Module Integration above, the team decided to use a Raspberry Pi as the vehicle's computer. Therefore, throughout the self-driving and obstacle avoidance development process, the team would Remote Desktop into the Raspberry Pi in order to write and execute code to test the car's driving. Remote Desktoping is a technology where a user can virtually access another computer and view its Desktop on your own personal device. However, in order to do this, the IP Address of the remote device needs to be known. For a Raspberry Pi, the easiest way is to attach it to a monitor using a HDMI/mini-HDMI and a power cable. The user also needs a keyboard and a mouse to run the following command in the terminal: "hostname -I". In order for this to work, the user first has to set up the Raspberry Pi to a wifi network or hotspot. Then the user will be able to Remote Desktop into the Raspberry Pi if they connect to the same network. To avoid this lengthy and troublesome procedure, the team pre-configured all of the Raspberry Pi's to the WPI-WIRELESS network and developed a startup script that automatically enrolls to the MPAD Web Application's server. This way, a student will never have to Remote Desktop into a Raspberry Pi, which can be a confusing and tricky process for someone who has not done it before.

The startup script was written in Bash, which is the default login shell and command language for most Linux distributions, including Raspbian (the Raspberry Pi Operating System). Raspberry Pi's have a script management tool which can be accessed through a tool called cron.

Cron is a time-based job scheduler in Unix-like computer operating systems, so the team was able to utilize this scheduler to execute the startup script and sensor-reading python script as soon as the Raspberry Pi turns on.

The sensor reading script is the less complex of the two. It simply captures the serial data present in the /dev/ttyACM0 buffer and puts the most recent reading into a text file in the /etc/selfdriving-rc directory. This was because a text file can be accessed by multiple processes at once compared to just one when reading from a USB device like the Arduino. The script was written in Python and includes various sanity checks like ensuring the device is connected with the appropriate serial specifications.

The second script is the master startup script that handles the car's controls. Upon startup, it will establish a connection to the server to get the car's status. From here, it continually polls the server every 500ms to check if any control updates are necessary. Once it detects that the car should be driving, it starts the self-driving algorithm as a child process. This approach of using a parent/child process eliminates any potential errors that may crop up while running the self-driving algorithm. For instance, if the car's algorithm is not working properly, the user can start and stop the driving to 'restart' the process. In addition, the parent process can monitor the child process and determine if a forceful termination is necessary, such as the event of the algorithm using an unusually high amount of memory. This architecture ensures the reduction of fatal errors and separates functions into easy to understand programs.

9.6 Conclusion

In this chapter the team covered the technical foundation that enables the dashboard to interface with the car. This begins with the car's startup procedure that both checks the car's

system, and connects it with the server. The server then acts as the gateway between the car and the web interface page. This page pulls in data in real time from the car's sensor array, while providing a means for users to control the car's self-driving algorithm. This system utilizes a wide array of tools including Flask, Heroku, bash scripting, Chart.js, Socket.io, and general web development tools (HTML, CSS, JS). These tools were used in tandem to provide the user an intuitive connection with the car. In the next section, we will detail how the dashboard and the self-driving system were tested to ensure accurate results and reliable performance for the end users.

10. Technical Testing

In this section, we discuss the sensor accuracy testing of each sensor in our system. We were able to calculate the percentage error for each sensor. This section also covers the user guide created for implementing the sensor package and AI features on the RC car. Lastly, we discuss the use of our sensor and AI package in the Introduction to Engineering Design (ME2300) course at WPI.

10.1 Inertial Measuring Unit (IMU) Accuracy Testing

To make the IMU accuracy tests, we first had to solder header connector male pins to the IMU to make sure that the connection was stable. Without the solder pins, we would not be able to get accurate results from the IMU readings. For these IMU testings, we tested three values. These values were the x, y, and z-axes. Both the x-axis and the z-axis would go from 180° to -180°. The y-axis goes from the -90° to 90°. During our testings, we would align the IMU with a straight surface and rotate the IMU 90° on a single axis at a time (x,y, and z-axes). For every rotation, data points were gathered to make sure that each axis would have accurate values throughout the range of values. The orientations of the IMU repeated themselves to make sure that the data points gathered were accurate. Figure 10.1 shows the axes of rotation that were tested and provide an idea of what the initial point actually was.

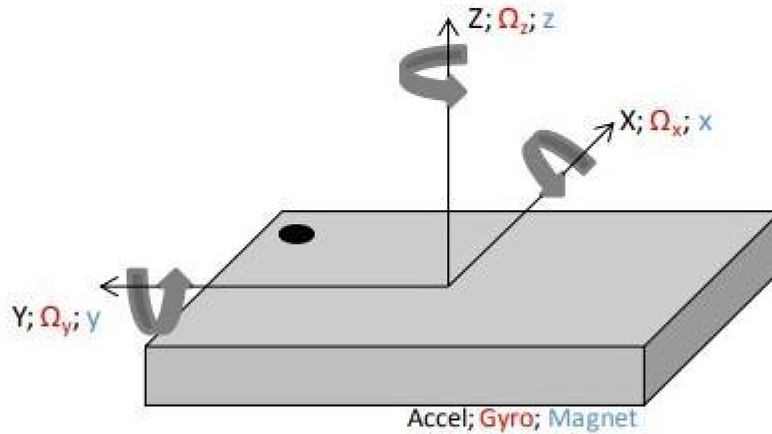


Figure 10.1: IMU Orientation for testing

In Appendix A, there are the actual results of the IMU testings in Tables 15.1, 15.2, and 15.3.

Table 10.1: Percentage Error of the IMU

	Percentage Error		
	X-axis	Y-axis	Z-axis
Average	0.98%	2.93%	2.69%
Maximum	3.33%	6.67%	6.67%

Table 10.2: Difference in Degrees of the IMU

	Difference in Degrees		
	X-axis	Y-axis	Z-axis
Average	1.07	2.63	2.77
Maximum	3	6	6

In Tables 10.1 and 10.2 we can see the results of the accuracy testings. Table 10.1 shows the average and maximum percentage error of each of the axes. Table 10.2 shows the actual degree difference for the IMU. For our specific project, this percentage error and the differences

in degrees are acceptable for our project. The following Figure 10.2, 10.3, and 10.4 summarize the results received from the IMU data points gathering.

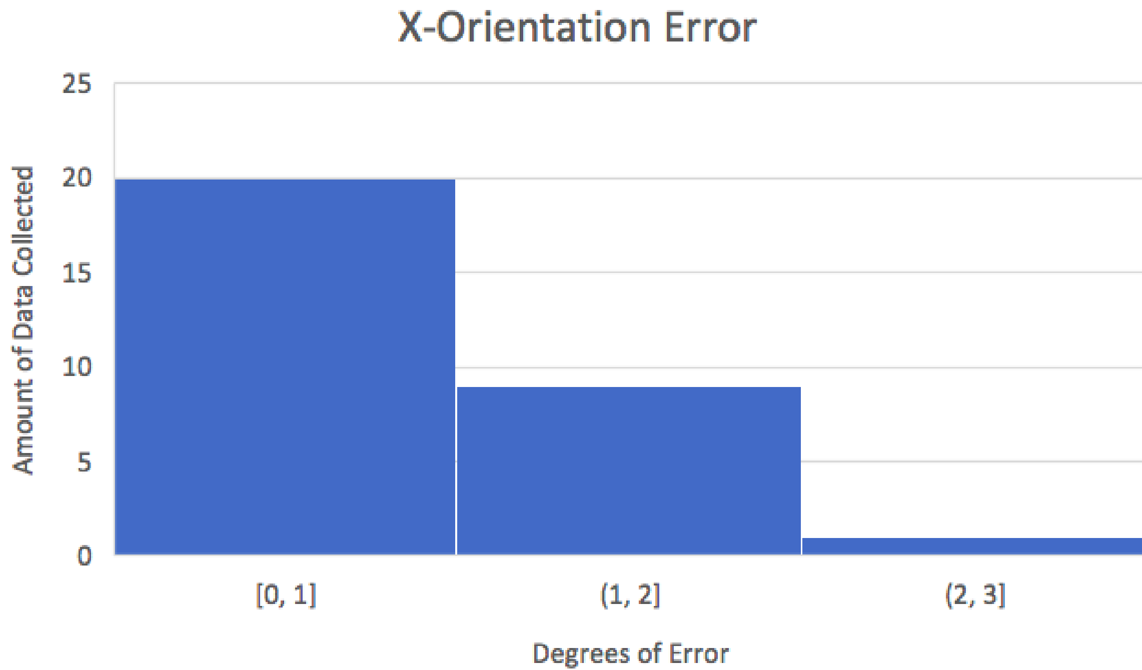


Figure 10.2: X-axis Orientation Error for the IMU

Figure 10.2 shows the degrees of error of the X-axis data points collected compared to what they were supposed to be. As seen in Figure 10.2, the X-axis of the IMU had great accuracy considering that most of its data is concentrated within the 0 to 1 degrees of error bin.

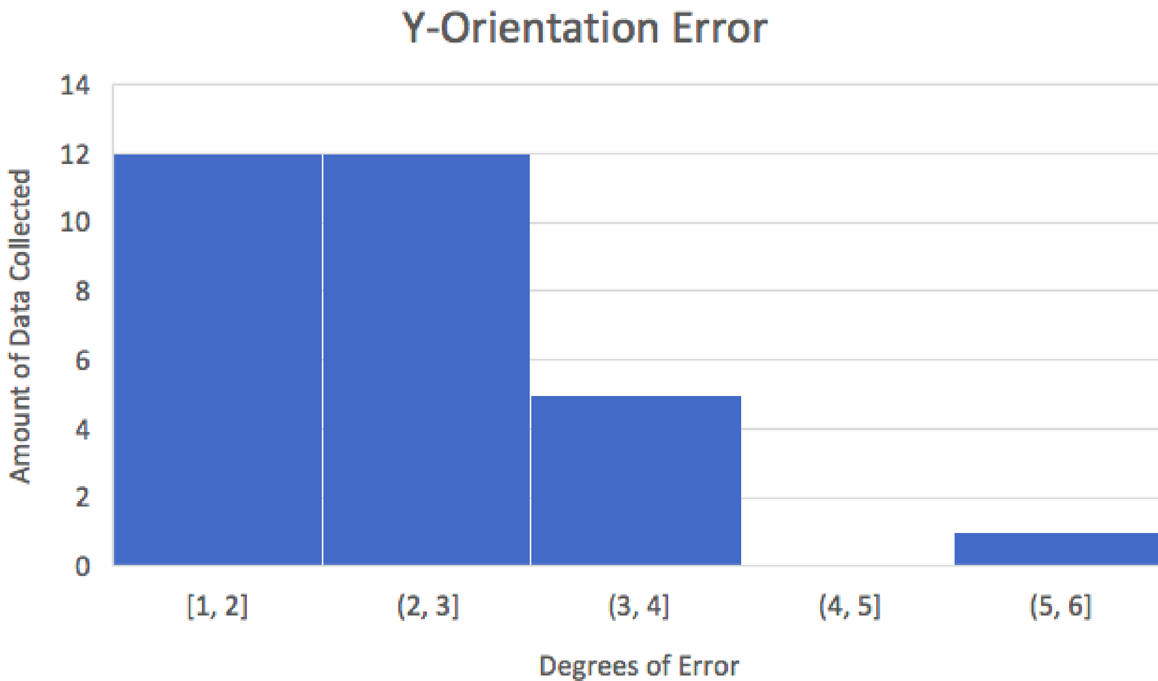


Figure 10.3: Y-axis Orientation Error of the IMU

Figure 10.3 shows the degrees of error of the Y-axis data points collected compared to what they were supposed to be. Comparing Figure 10.3 to Figure 10.2, it is possible to notice that the Y-axis is not as accurate as the X-axis. However, it still has a good accuracy with most points having less than 3 degrees of error. For this specific project, having this level of accuracy is completely fine given that what matters is surpassing a threshold and that the 3 degrees of difference would most likely not make a difference.

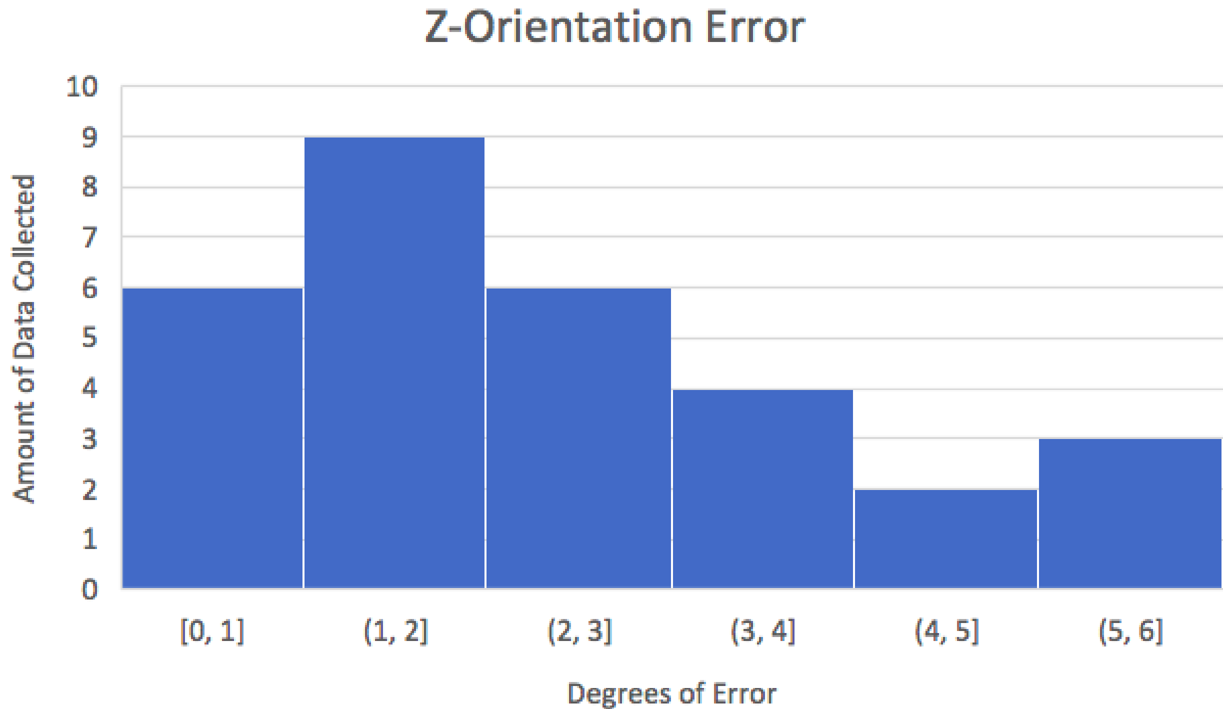


Figure 10.4: Z-axis Orientation Error of the IMU

Figure 10.4 shows the degrees of error of the Z-axis data points collected compared to what they were supposed to be. Compared to the other two axes, the Z-axis had the worst overall accuracy. The accuracy was spread out between most of the data bins having its peak in the 1 to 2 degrees of error bin. Overall, the 5 to 6 degrees of error data points collected is not ideal, but it is still manageable given the application of the IMU in this project.

10.2 Ultrasonic Sensor Accuracy Testing

To test the accuracy of the ultrasonic sensor, we placed an object in front of the sensor at various distances and compared the readings with a tape measure. The Ultrasonic sensor was accurate in detecting closer objects, but was less effective when the object was further away. This is due to it's range which is 2cm to 400cm. During the testing, we found that they were less effective with objects at an angle. The angles between 30 and 60 degrees were blind spots out of

90 degrees. As seen in the Table 10.3 below, the ultrasonic sensor was close in values to the tape measure, however the object was directly in front of the sensor. The results from the sensor testing showed that the average error was 0.39% and the maximum error was 0.89%. Overall, the ultrasonic accuracy testing proved that the ultrasonic sensors have great accuracy whenever an object is directly in front of it.

Table 10.3: Ultrasonic Sensor Accuracy Testing Readings

Tape Measure (cm)	Ultrasonic Sensor Reading (cm)	Error Percentage
10	10	0.00%
20	20	0.00%
30	30	0.00%
40	40	0.00%
60	59.8	0.33%
80	79.52	0.60%
100	99.77	0.23%
120	119.12	0.73%
140	139.06	0.67%
160	159.4	0.37%
180	178.72	0.71%
200	198.84	0.58%
220	218.17	0.83%
240	239.21	0.33%
260	259.31	0.27%
280	279.8	0.07%
300	298.3	0.57%
320	318.32	0.53%
340	339.75	0.07%
360	358.12	0.52%
380	376.6	0.89%
400	398.62	0.34%

10.3 Hall Effect Sensor Accuracy Testing

To complete the Hall Effect Sensor, test three separate gear systems were used. These gears were held within a gearbox. By activating the motor, the goal of the test was to measure each RPM of the individual gears in the system. To get these readings, a small magnet was placed adjacent to the gear teeth. The magnet provided the magnetic pulse to be read and charted. The RPM diagram of the configuration can be seen below in Figure 10.5. As for the physical testing of the gear and sensor, we started by calculating the RPM of each gear tested. We counted how many rotations the gears would make in a minute. Those values can be seen in Table 10.4. The sensor and code worked very well and had a low percentage error. The highest error percentage calculated was 2.34% on the first trial.

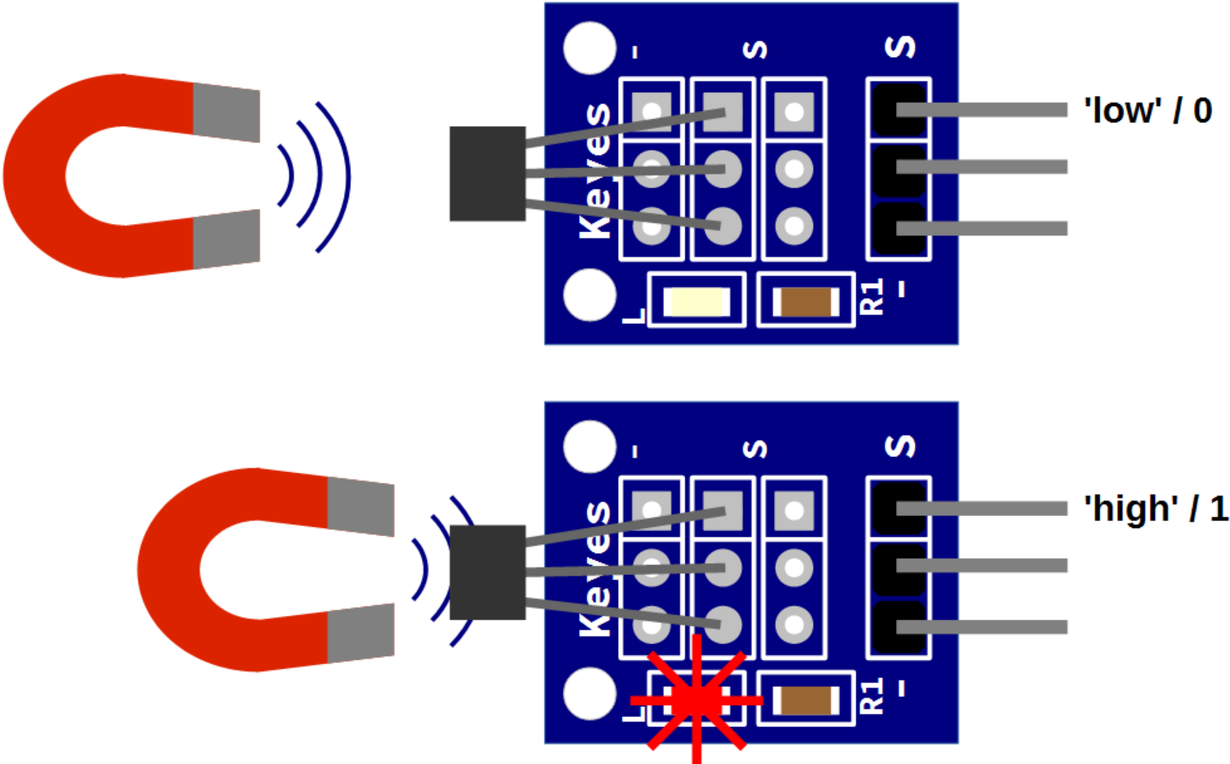


Figure 10.5: RPM Diagram of Hall Effect Configuration

Table 10.4: Accuracy Testing Data for Hall-Effect Sensor

Expected (Counted)	Calculated (Sensor)	Precentage Error
17.5	17.09	2.34%
17.5	17.81	1.77%
17.5	17.62	0.69%
3.35	3.39	1.19%
3.35	3.3	1.49%
3.35	3.29	1.79%
16.65	16.63	0.12%
16.65	16.57	0.48%
16.65	16.52	0.78%
84	84.75	0.89%
84	84.51	0.61%
84	85.47	1.75%
86	86.21	0.24%
86	85.59	0.48%
86	87.08	1.26%

10.4 Temperature Sensor Testing

The temperature sensor testing was made differently than the other sensors. The temperature sensor function in MPAD is to make sure that no part of the RC car and package is overheating. Once the temperature sensor is heated up it takes a long time to cooldown. Due to this characteristic of this sensor, it is hard to actually gather accuracy data from it. The temperature sensor was tested by bringing the sensor near heat sources (for example a space heater), and checking to see if the values of temperature would actually change and notice the difference. Once the temperature sensor was removed from the surroundings of the heat source, the temperature readings would decrease at a slow pace. This characteristic is what made it hard to test the temperature sensor, since sudden changes in temperature might not actually be reflected by the temperature sensor. However, in our application the temperature sensor could

still prove to be useful, because the only purpose for the temperature sensor is to notify that a part is overheating.

10.5 Project Implementation

In order to implement and test our modular package, we soldered multiple PCB boards for the ME students. This board has the IMU soldered on it, Pins for inputs and outputs and lastly resistors for the temperature sensor. An example for the sensor layout on the PCB board can be seen in Figure 10.6. A PCB board was chosen over a breadboard to save space on the car and their ease of use. Each student in the Introduction to Engineering Design (ME2300) course received seven ultrasonic sensors, three temperature sensors, an IMU sensor and hall effect sensor. Each soldered board was tested for its functionality and accuracy.

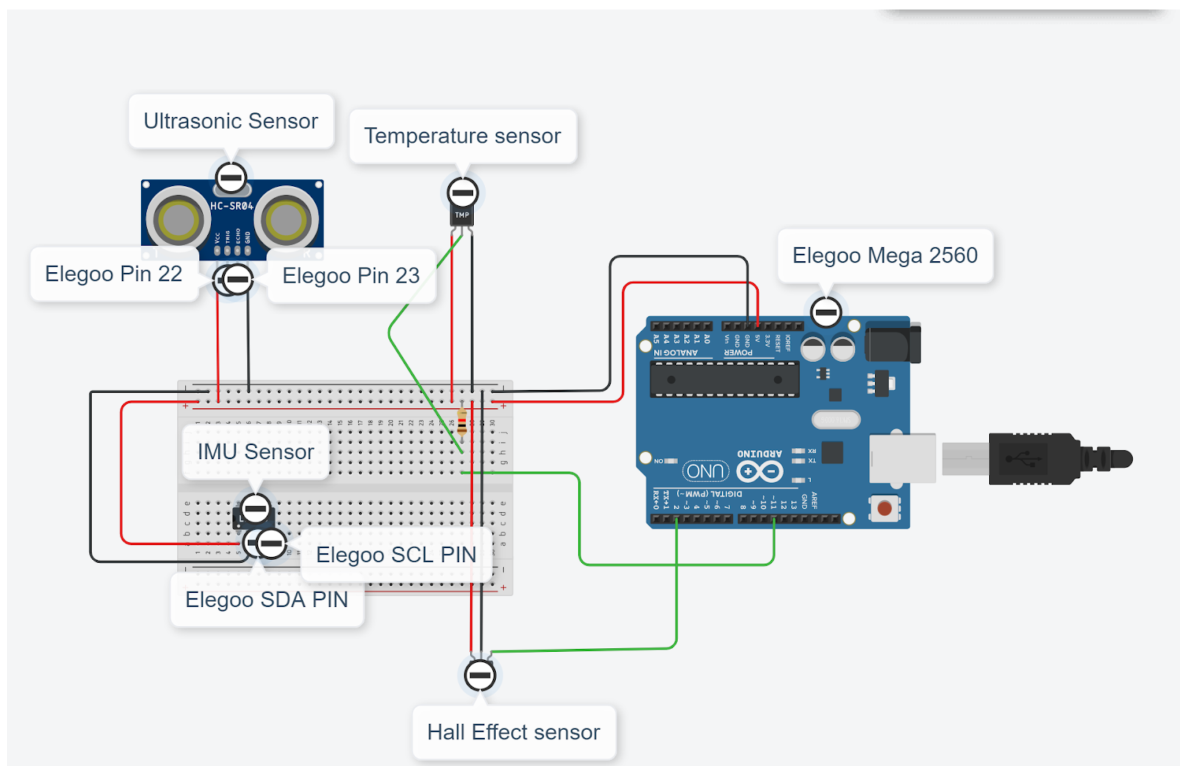


Figure 10.6: PCB board sensor Layout

10.6 User Guide

We created a user guide with the overall process of assembling the sensor and AI package. The user starts with a list of hardware required for the MPAD. The guide then goes into the hardware set up for components such as the camera and servo driver. Next, the guide covers the AI and UI system and steps for connecting the car to the dashboard. Lastly,the guide provides schematic and images for wiring the sensors and downloading the sensor code. The full guide can be found in Appendix B.

With this guide, students in the Introduction to Engineering Design (ME2300) course can implement the sensor and AI package on their designed RC car as long as they have a 3-pronged servo connection. One of the main goals of our project was to have a modular system that can be implemented on any RC car without training. Our guide provides all the information such that no prior experience is necessary in rebuilding our sensor package.

10.7 Autonomous Driving Results

Given the car's mechanical dimensions and size, we tested the car on multiple tracks with the full modular package. The car's track was set up using tape for the lane following. The car was tested for its speed on the track and it's ability to maneuver the obstacles and stay in lane. The sections below describe the testing in detail.

10.7.1 RC Car for Testing

The RC car used for testing is 27cm wide, 41cm long, and 18 cm high. It is a 3-D printed RC car made from mostly plastic parts with some aluminium elements. It has rubber PRO1190-013 wheels, a JUSTOCK 3650SD G2 Brushless Motor, and a SAV-SC-1256TG Steering Servo. The RC car is rear wheel drive only and is capable of reversing direction. In terms of steering, each wheel is capable of turning 46 degrees in both directions. This gives our RC test car a turning radius of roughly 90 degrees. The turning radius measurements can be found in Figure 10.7. Images of the car can be found in Figure 10.8 and Figure 10.9.

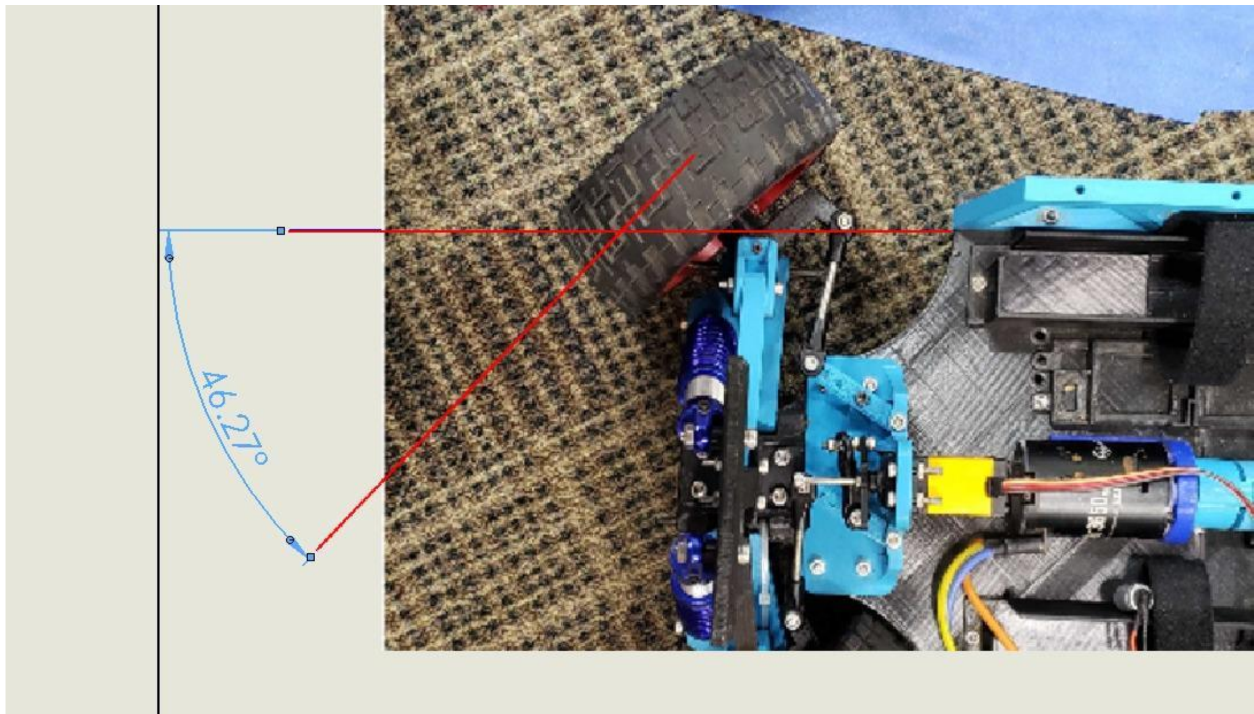


Figure 10.7: Turning Radius Calculations

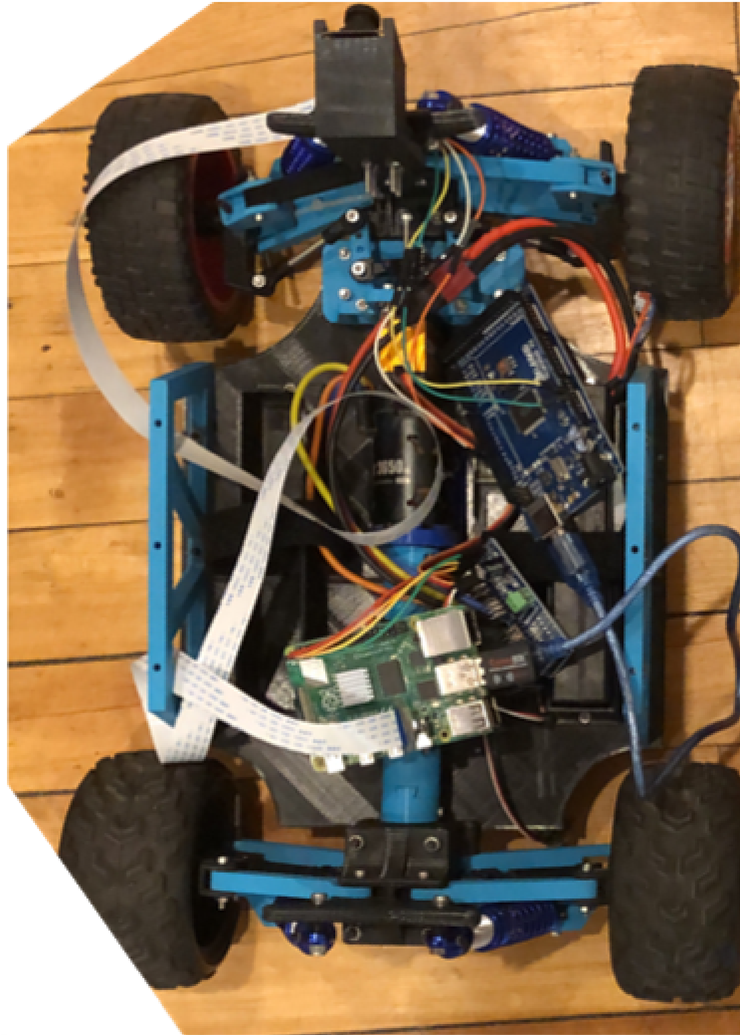


Figure 10.8: Birdseye Image of RC Car

Figure 10.9: Sideways View of RC Car

10.7.2 Lane Following

A track was set up in the bottom floor Higgins Labs using tape. A rough sketch of the track and the surrounding room can be found in Figure 10.10. The track is roughly twice the width of the car, meaning in our case the track was around 50 cm wide. Some areas of the track however were almost up to three widths of a car at 75cm wide. Figure H is an image of the car on the track. Just doing lane following the RC car with the package was able to complete the track reliably up to a speed of roughly 1.1 meters per second. It took roughly 32 seconds for the car to complete a single loop. Different speeds were tested by changing the value of *kit.continuous_servo[1].throttle = X* through a remote desktop connection. Higher speeds were tested but the car would go off the track in the sharp turn found in the upper right corner of

Figure 10.10. Latency is always a factor but most of the time the driving code can make up for the little inaccuracies. As the speed of the car increases, the latency stays the same but an opportunity to make up for it decreases. Sharp turns also have less room for error so during high speed testing they tend to be the point of failure. The track was completed both ways resulting in the same performance. Performance in this context is how fast the car can complete a lap, how little it oscillates in a straightaway, and generally how much the car stays in the middle of the lanes.

A video of the car autonomously navigating the course can be seen in the following links, each video is a different example of the lane following in the track in Higgins Laboratories. These videos look similar to Figure 10.11. Watching the videos will give a better sense of the speed and performance of the lane following. The first two videos are a top down view of the lane following completing a loop at 0.5 and 0.4 throttle respectively. The third video is a closer look at the turning from a ground perspective.

Video 1:

https://drive.google.com/file/d/1jV-99D5fjMUC5ARiA5n7WP-pCuKKI0ix/view?usp=sharing

Video 2:

https://drive.google.com/file/d/1juQ0-trf4w6Yx3iPMgRqoQPBqu43ks65/view?usp=sharing

Video 3:

https://drive.google.com/file/d/13cqevBN7nMKHYZQfraD9xoayIaacCHI0/view?usp=sharing

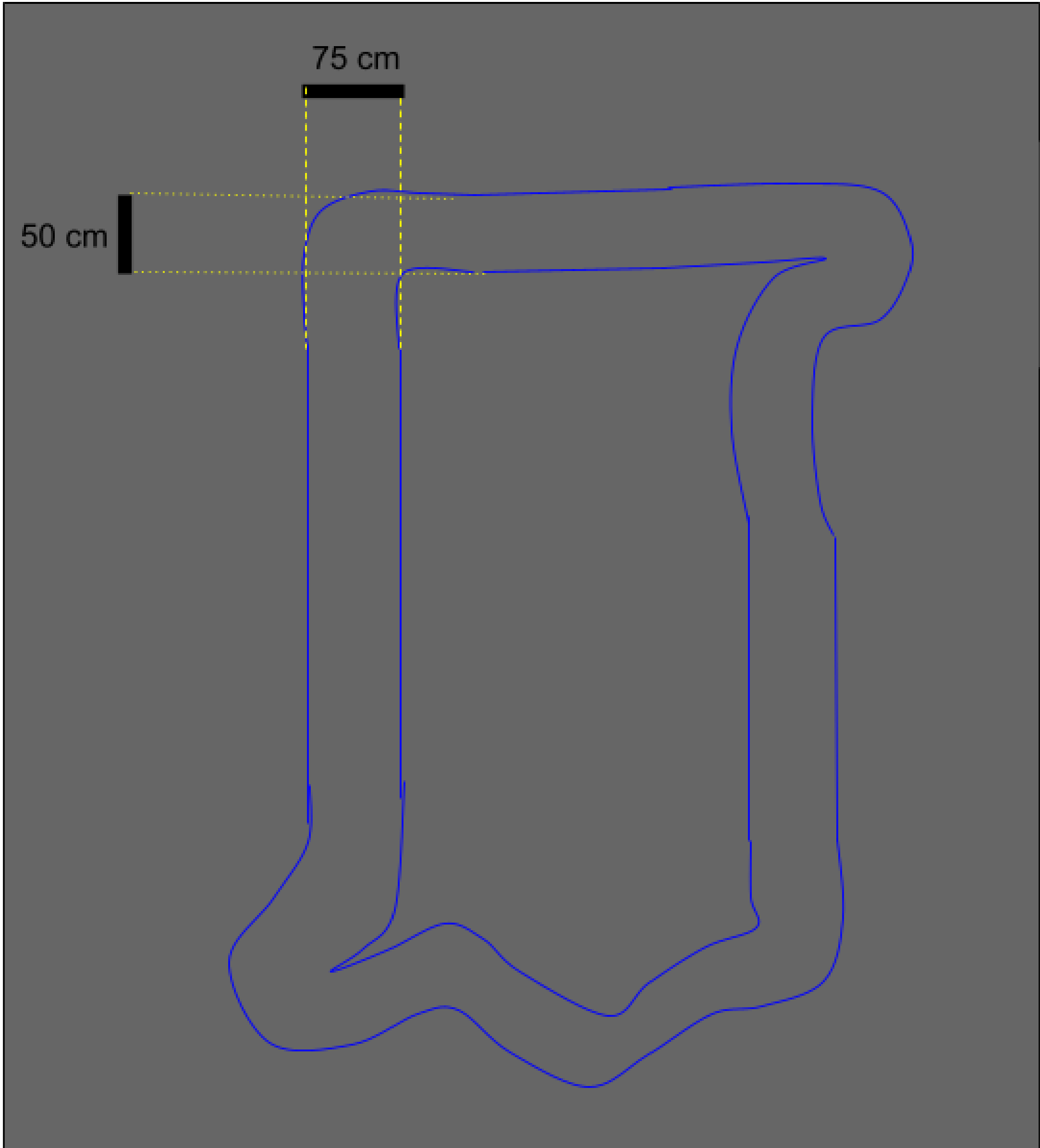


Figure 10.10: Track in Higgins

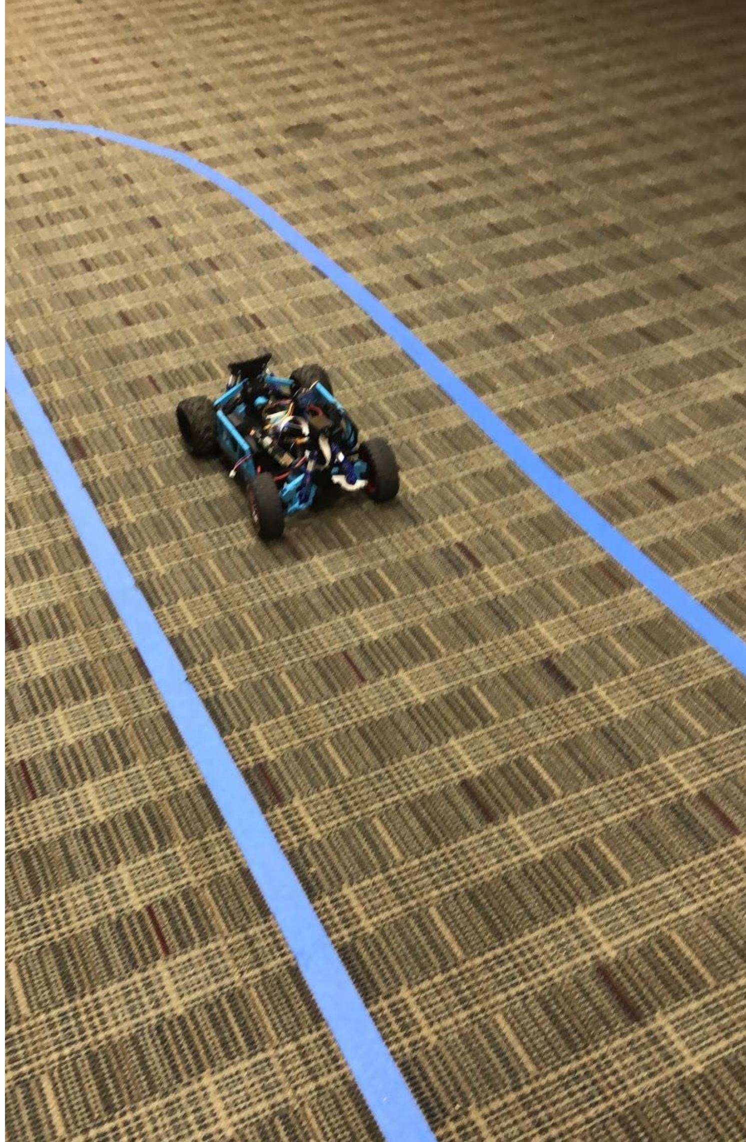


Figure 10.11: Car in Track

Time functions were used in the code to determine runtime of each loop. The average time over thousands of loops was calculated as 35 ms. The runtime would increase if there was a large number of white pixels detected post color filter. The camera records at 30 fps. So, at minimum, the loop has to wait 33.3 ms for the image to be taken. This means the driving code itself was fairly efficient. Practically, with the 35 ms runtime, the driving essentially runs at 28.5 fps.

10.8 Obstacle Avoidance

The obstacle avoidance was tested on the same track as the lane following (Figure 10.10). The bumper was attached and the ultrasonics were installed. Initially, the car was tested without any obstacle to test if the lane following still functions properly. The results were inline with our earlier lane following tests. There were no false positives on the empty track.

The obstacle avoidance itself was tested by having the car complete the track continuously while increasing the size of the obstacle and how much of the lane they blocked. Testing showed that as long as the obstacle had a surface to bounce off, the size of the obstacle was not as important as the total road blocked. The car was able to avoid obstacles and still stay within the lane as long as at least 60% of the road was unblocked. Sharper turns were required on the bigger blocks so not every car might have the steering capabilities to avoid it. Testing can be seen in Figure 10.12 or watched in:

https://drive.google.com/file/d/15bLv6veQlNPVU4T7B9WpzCzTLLPKx6Ex/view?usp=sharing

Figure 10.12: Obstacle Avoidance Testing

10.9 Dashboard Sensor Integration

The dashboard is the hub for all of the sensor data to be shown in a user friendly way. Across multiple graphs, sensor data is arranged and updated within 500ms of it being read on the Arduino. The testing process is relatively simple. By opening both the serial feed on the Pi and

the web dashboard, we can see the flow of data from the Arduino. Tests were conducted with the full array of sensors plugged in. After waiting 2 minutes, the car was connected to over SSH:

ssh pi@<car-ip>

Then, the serial stream opened via the tail command. The -f flag follows the data as it comes in:

tail -f /dev/ttyACM0

Finally, the dashboard is opened alongside. When a new serial stream comes in, the graphs are checked for accuracy. For example, the following sensor string:

|humidity:37.00|temperature:27.90|heat index:27.39||Distance 1:86.16|Distance 2:3.20|Distance 3:4.98

Should result in the following entries on the respective car's graph, as shown in Table 10.6.

Table 10.5: Sample sensor readings from a serial string

Sensor Name	Reading
Humidity	37.00
Temperature	27.00
Heat Index	23.97
Ultrasonic 1	86.16
Ultrasonic 2	3.20
Ultrasonic 3	4.98

Once the entries were confirmed to be present, we could run the test for 10 more loops until we could ensure each sensor was polled correctly. See Appendix P for a video demonstration of this live feed.

Each of the sensors are presented on a standard line graph, with the exception of the ultrasonics which are shown on a web graph. See Figure 10.13 for a breakdown of what each sensor's graph looks like.

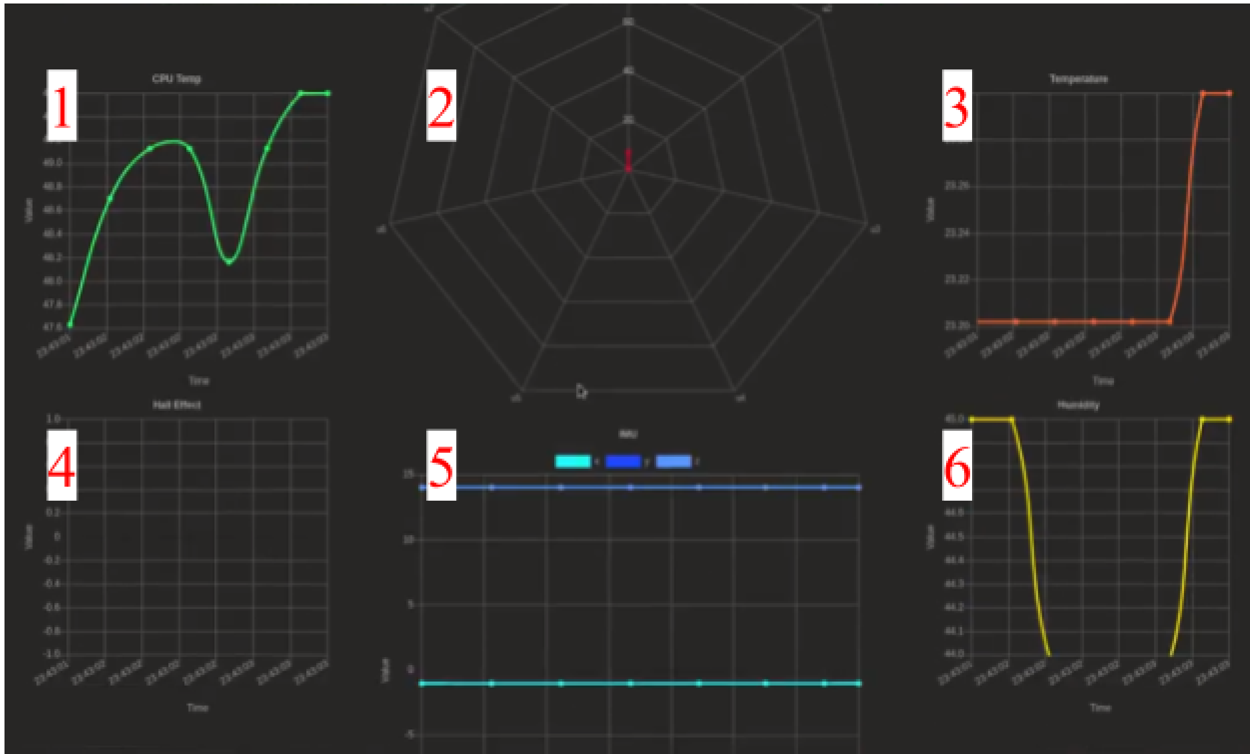


Figure 10.13: 1) CPU Temp reading 2) Ultrasonic readings. Each ultrasonic is angled in a different direction. 3) Ambient temperature 4) Hall Effect 5) IMU data, with a line for each X, Y, and Z axis. 6) Humidity

11. Financial Analysis

In this financial analysis chapter, a financial viability check for opening a business based on the MPAD's concept to fit into any RC car. The financial analysis goes over the costs of building a single unit of MPAD, the projection of the investment needed to open the business, and the process of pricing the MPAD based on the current market. The first step for this financial analysis was to figure out what would be fixed and variable costs would be taken into account. The target client of the MPAD would be Academic institutions and any person who would like to automate their own RC cars.

11.1 Fixed Costs

The fixed costs that were taken into account were: rental payments of a factory, insurance of the building, direct labor, cost of buying machines, utility service bills (such as water, electricity, gas), and interest payments. For the rental payments, warehouses near Worcester, MA were considered during this financial analysis. Since the process of building a MPAD does not require a large area, only warehouses of a smaller area were considered. After some research, three possible warehouses that would fit the requirements were found. Table 11.1 shows a breakdown of the costs and features of each option.

Table 11.1: Warehouse Choice Costs and Features Table

Warehouses	30 Glennie St	243 Stafford St	49 Gardner St
Size (SF)	3020	12569	6068
Extra Space	-	10791	6068
Drive-in Bays	1	1	2
Parking Spaces	50	-	11
Rate (SF/year)	$15.00	$5.00	$7.00
Water	City	City	City
Sewer	City	City	City
Heating	-	Gas	Gas
Yearly Rental Payment	$45,300.00	$62,845.00	$42,476.00

From Table 11.1, it is noticeable that the warehouse at 49 Gardner St seems to be the cheapest, while still covering the requirements for MPADs production. With the warehouse selected, the next step was looking into the insurance. A general insurance for a small business costs between $500 to $1500 per year, with 42% costing between $500 and $1000[34]. With that in mind, the insurance cost considered for this financial analysis was $1000. After insurance was settled, the following step was to look into how many workers the business would need. Since this project was made with six people, we could have six employees for the company, which would be a reasonable number. For the hourly rate of the employees, the average salary for a normal employee should be around $15 per hour and for one manager, it would be around $25 per hour. If workdays are eight hours long and if a year has a total of 252 working days (same as 2020), the total cost per year would follow the equation below.

Employee Cost = $15 * 5 * 8 * 252 + $20 * 8 * 252 = $191,520.00 per year

For the implementation of the MPAD, some 3D-printed parts are needed to make full use of the modular package. The main 3D printed component that is needed is the bumper to hold the

ultrasonic sensors. In the financial analysis consideration, it was decided that if the customer wanted, the company would produce different parts such as chassis and suspensions for them. Given that, the company would need to have multiple 3D printing machines. The prices for 3D printers can vary quite a bit. After some extensive research, it was decided that the 3D printer of choice should print using carbon fiber and the best cost-benefit 3D printer was the Ultimaker S5 Pro[35]. This specific 3D printer costs $9,100 per unit. To find out how many actual printers would actually be used an ARENA model was made. Figure 11.1 and Figure 11.2 shows the ARENA model that was used to get some useful information on the numbers for the financial analysis.

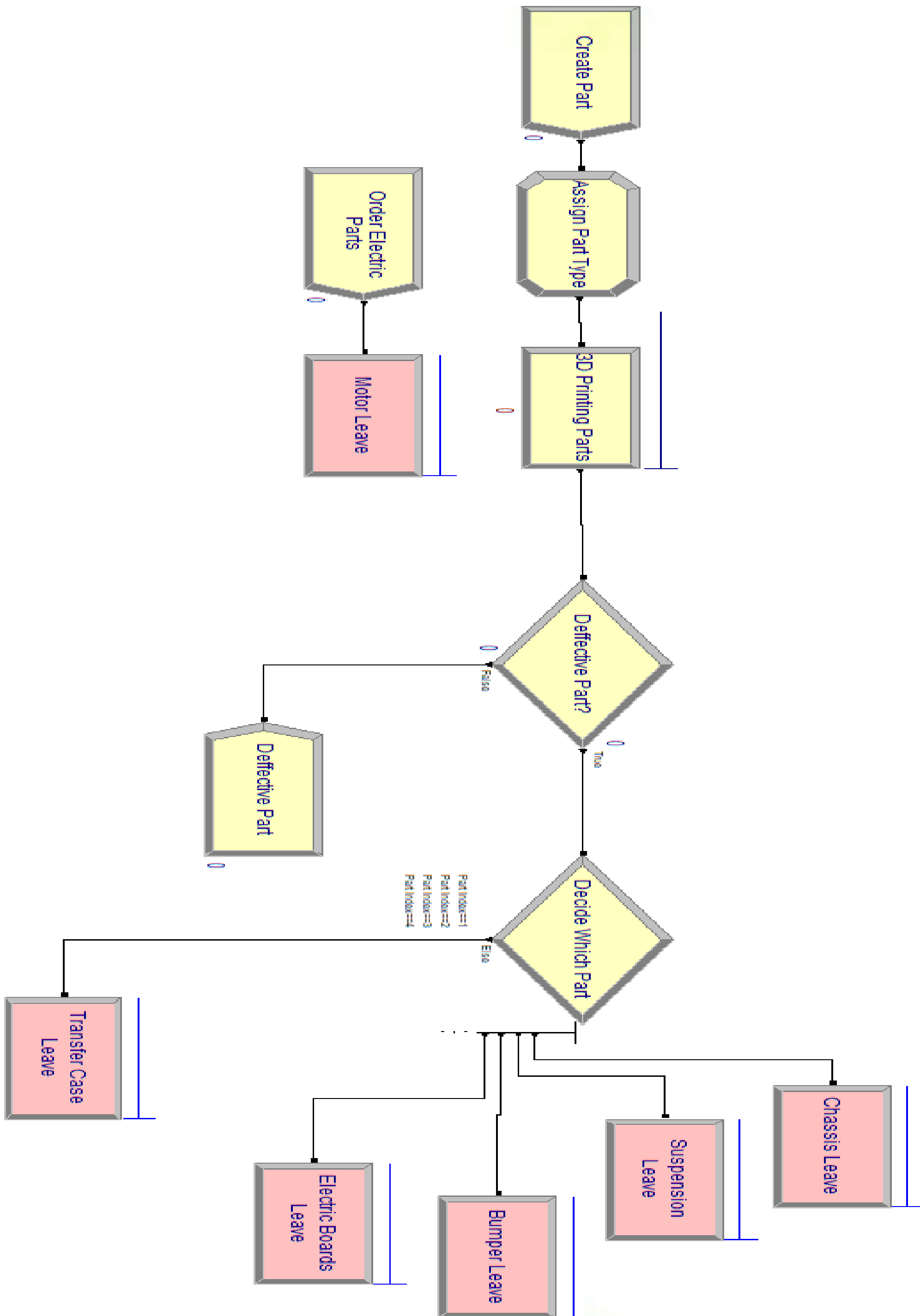


Figure 11.1: First Part of ARENA Model for Financial Analysis

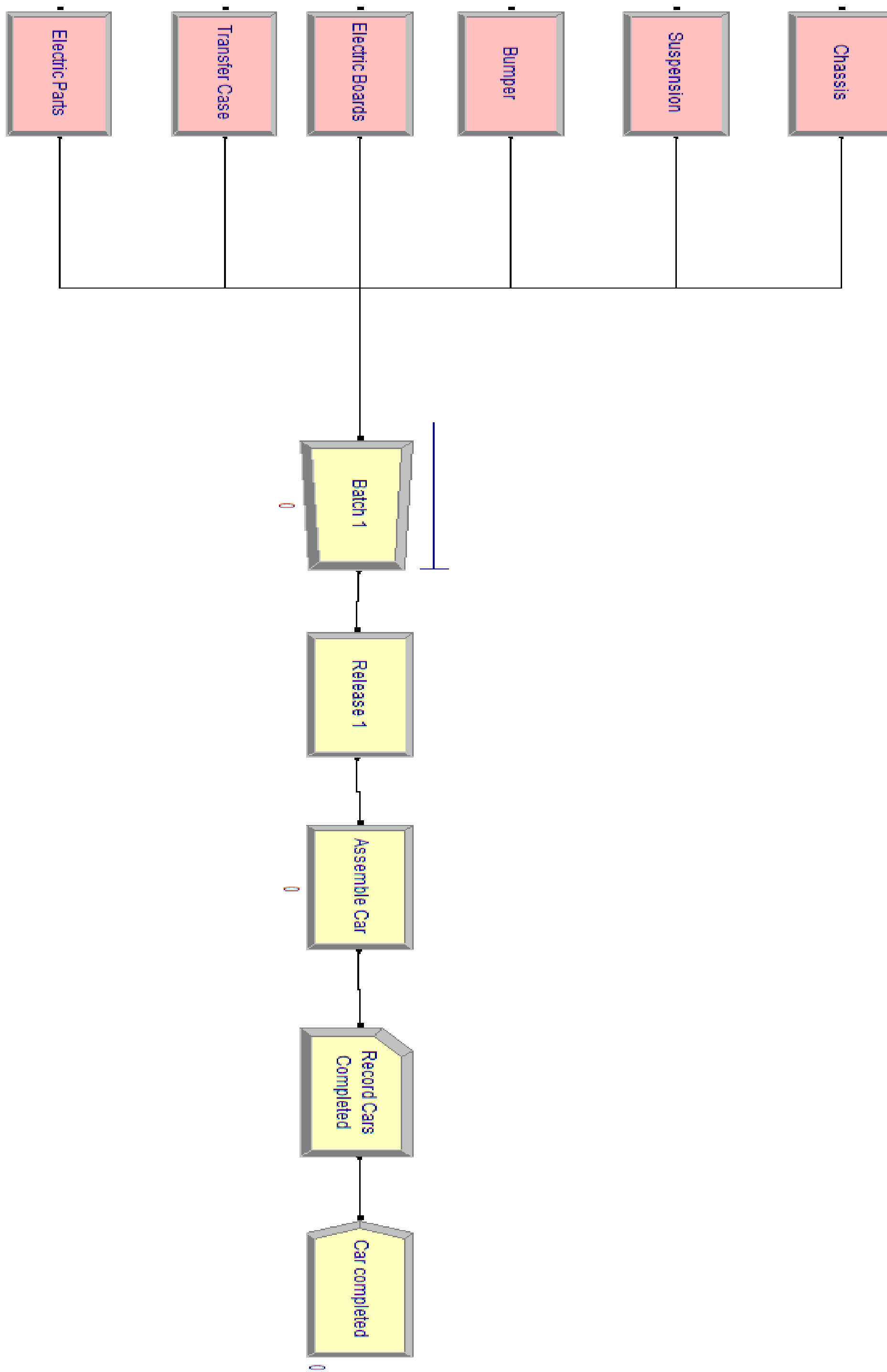


Figure 11.2: Second Part of ARENA Model for Financial Analysis

The Arena Model can be divided into two main sections, the first is the manufacturing and ordering of parts, while the second is the assembly of the parts. The entities that are flowing through the system are the 3D printable parts and ordered parts. The first section is responsible for assigning what type of part will be processed through the model. Each different part requires a different 3D printing time. Once a defective part is done being 3D printed, the model will check if the part was defective or not. Defective probabilities vary a lot depending on the sizes and complexities of the parts. If it was defective, the 3D printer would need to reprint the whole part again. After the defective parts have been checked, they are sorted into appropriate bins, and once there are enough entities to build a full package they move to the second section of the model. In the second section, the parts are batched and assembled, and eventually shipped out, in other words, they exit the system. All the numbers for the 3D printing part times and the assembly time of the MPADs were made based on the real-time data gathered of ME students who worked on printing parts and building RC cars.

After testing, it was found that the optimal number of 3D printers for this application was 5. Knowing that the 5 machines will be used, it is possible to calculate total machine cost of 5 * 9,100 = $45,500. To cover up for such a big amount before the business starts running, a loan can be borrowed. To try and avoid paying extra interest, the debt should be paid in the least amount of time possible. With that in mind, a three year limit with a 5.95% interest rate was assumed. Another loan that could be borrowed is a loan that covers all of the fixed costs to help start the business. This would probably require a longer time to be paid and it would be ideal to search for a lower interest rate.

The next step for the fixed costs would be to consider utility service bills. The Ultimaker S5 Pro consumes 0.05 kWh per hour. While doing our calculations we will consider that the 3D

printers were only actively working during the 8 hours of the workday. With that, the annual electricity consumption of the 3D printers is derived from the following equation: 0.05 * 5* 8 * 252 = 504 kWh. The electricity bill rate in Massachusetts is 22.57 cents per kWh[36]. So, the electric bill for the 3D printers would equal 504 * 0.2257 = $113.75 per year. Of course, this would not be the entirety of the cost of electricity, however, it is extremely hard to predict how much electricity a small company would consume and how much it would pay annually.

11.2 Variable Costs

When it comes to the production of MPAD, there are a few values that we have to take into account. Some of them are the raw materials needed, the packaging costs, the shipping or distribution costs, and the maintenance costs. The first main consideration is to make a bill of materials to find the unit cost of each MPAD unit that will be produced. The second consideration would be to utilize Amazon Seller Central. The cost to use Amazon Marketplace would be $39.99 a month plus some additional fees[37]. Amazon would take care of almost all the shipping, packaging, labeling , and customer services aspect of the business. It is hard to tell how many additional fees MPAD would require, but it can be assumed that on an average, it would be somewhere around 15% of the selling price. The last part of variable costs that needs to be taken into account is the cost to repair 3D Printers. 3D printers tend to have a lifespan of 5-10 years, depending on how well it is maintained and its frequency of use. Since the machines would be constantly active, it is reasonable to assume that the lifespan of the 3D printers would be 5 years. A repair for a 3D Printer should cost about 30% of the original price of the 3D printer price. So, to repair the 5 3D printers, the following annual cost equation would be used: 9,100 * 0.3 = $2,730.00 per year. Notice that since the repair would happen every 5 years and there are

actually 5 machines used the fives would cancel. This value could also vary depending on the production rate of the company.

11.3 Financial Analysis Results

After analysing different aspects of the costs involved in starting a business, it was possible to create a bill of materials for a single unit of MPAD. As seen in Table 11.2 below, the total cost for creating a single unit of the MPAD was $179.14.

Table 11.2: Bill of Materials of MPAD

Sensor and AI Part	Quantity	Unit Cost	Combined Cost
Temperature Sensor	3	$3.89	$11.67
Ultrasonic Sensor	7	$1.68	$11.76
Inertial Measurement Unit	1	$34.95	$34.95
Hall Effect Sensor	1	$14.99	$14.99
Camera	1	$15.87	$15.87
Raspberry Pi 4	1	$35.00	$35.00
Jumper Wires	1	$9.88	$9.88
Elegoo Mega 2560	1	$15.99	$15.99

Micro SD	1	$24.03	$24.03
Half Size Breadboard	1	$5.00	$5.00
Total Cost	-	-	$179.14

After running the ARENA Model, it was possible to find the production rate of MPAD. The production rate with five 3D printer machines was found to be 26 MPADs per day. With 252 work days in a year, the total production of MPADs per year is equal to 26 * 252 = 6,552 units. However, we have to consider that some mistakes might happen, so let's assume that a total of 6,500 units are produced in a year. The following assumptions breakdown and the Profit and Loss Statements are used to identify what the revenue and gross profit will be.

Assumptions

MPADs Sold	6500
Price / MPAD	$260.00
Cost / MPAD	$179.14
Rent of Warehouse	$42,476.00
Insurance of Building	$1,000.00
Direct Labor	$191,520.00

Electricity for Machines	$113.75
Maintenance Costs	$2,730.00

Profit & Loss Statement

Revenue	$1,690,000.00
Variable Costs	$1,418,390.00
Gross Profit	$271,610.00
Fixed Costs	$237,839.75
Operating Profit	$33,770.25

In this assumption breakdown, the gross profit represents the difference between the revenue and the variable costs. The revenue is the multiplication between our units sold and the price of the MPAD. The price of MPAD was chosen based on the cost to produce a single unit, however, it does not mean that this will actually be the actual selling price. The selling price is decided after performing a sensitivity analysis. The sensitivity analysis for this case is shown in Table 11.3. The number that actually matters for the financial analysis is the operating profit. The operating profit would equal the gross profit minus the fixed cost, which would give the actual profit the business had.

Table 11.3: Sensitivity Analysis

				Units Sold		
	$33,770.25	5000	6000	6500	7000	7500
Selling Price	$280.00	$55,980.25	$114,840.25	$144,270.25	$173,700.25	$203,130.25
	$270.00	$13,480.25	$63,840.25	$89,020.25	$114,200.25	$139,380.25
	$260.00	($29,019.75)	$12,840.25	$33,770.25	$54,700.25	$75,630.25
	$250.00	($71,519.75)	($38,159.75)	($21,479.75)	($4,799.75)	$11,880.25
	$240.00	($114,019.75)	($89,159.75)	($76,729.75)	($64,299.75)	($51,869.75)

To perform the sensitivity analysis, the What-if-Analysis feature of Excel was used. The sensitivity analysis helps to decide the price at which MPAD should be sold. As the selling price increases, the probability that there would be less units sold increases. The same idea is utilized when the price is decreased. On average, the correct amount sold in relation with the prices is seen through the main diagonal across the sensitivity analysis. With that in mind, it is possible to see that the best selling price would be to sell MPAD at $270 per unit, which on average would allow 6,000 units of MPAD to be sold. This would result in an operating profit of $63,840.25, which is greater than the one presented in our assumptions.

As mentioned in the Methods (Chapter 6), a loan will be used to cover the machine's cost. To figure out how much each annual payment would be, the use of time value of money analysis was needed. Table 11.4 below shows the calculations made for the annual payment that would need to be done.

Table 11.4: Time Value of Money Analysis Table

i	N	PMT	PV	FV	Solve for	
5.95%	3		$45,500.00		PMT	($17,006.25)
4%	15		$237,839.75		PMT	($21,391.57)

In this table, it is possible to see that two different loans were considered. The first one had a 5.95% interest rate, to be paid in a 3 year span and the total quantity borrowed was $45,500. The Present Value quantity represents the cost of buying the machines. To calculate the payment the PMT function in excel was used. The total payment per year generated for this loan was $17,006.25. The second loan calculation was made using an interest rate of 4%, with a 15 years life span and a present value of $237,839.75. With that, the annual payment needed is $21,391.57.

After doing all the previous calculations and deciding the production rate and sell price of the MPAD, it is good practice to conduct a break even analysis to find out when the business would start to actually make money. Figure 11.3 shows the Break Even chart of the MPAD product.

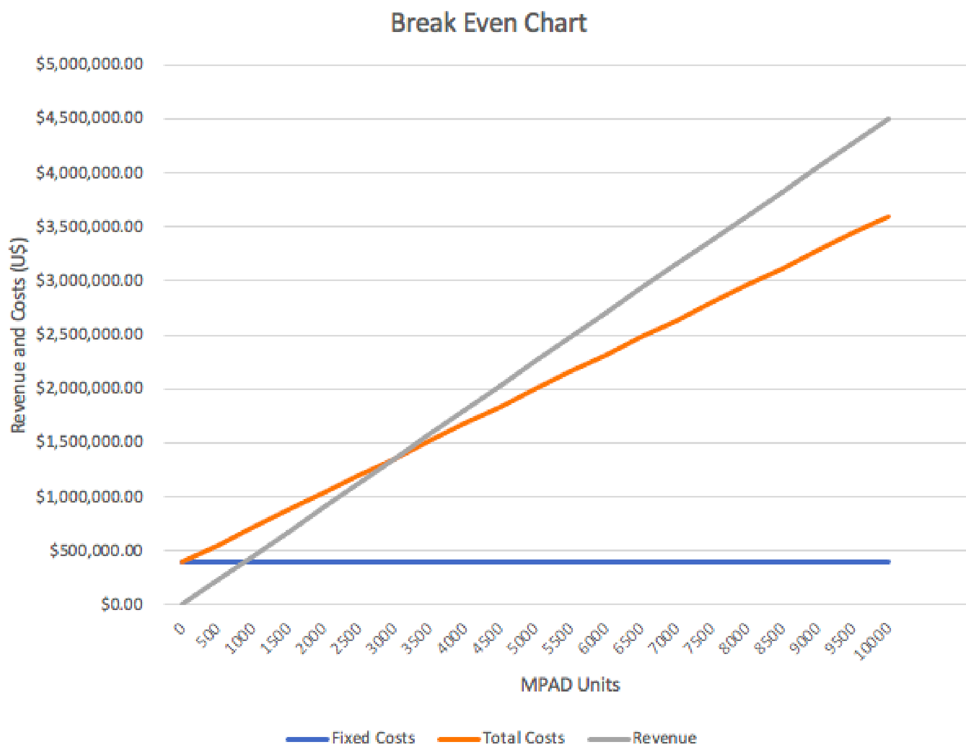


Figure 11.3: Break Even Chart

From the break even chart, it is possible to see that once about 2,000 units of MPAD are sold, the business would break even and would eventually start making an actual profit. To find how long it would take to break even, just divide the 2,000 units by 26, which is the production rate per day. Given that, it is possible to see that it would take about 77 days of the business operating to actually start making a profit.

11.4 Conclusion

With all of these data gathered and the results achieved, it is possible to reach a conclusion that creating a new company for this would be an extenuating task and more variables would need to be taken into account. Some other variables that could be looked into are marketing, inventory costs, and risk factors. However, given all the analysis undertaken in this project, it would be possible to start a company with MPAD as its product. This financial analysis establishes a good basis for the first step one would take to open their own company. The next chapter contains the discussion of the integration of this project into the Introduction to Engineering Design (ME2300) course.

12. Discussion

The discussion section will dive into a more in-depth description of the team's results and findings. To prepare for the course implementation, the team was first able to make important project changes and overcome obstacles through our own testing. To replicate what students would be doing in the course, the team re-replicated the entire development process from scratch. Then, after implementing this project into WPI's ME 2300 course, taught by Professor Pradeep Radhakrishnan, the team was able to gather feedback from students and the course staff.

12.1 MPAD Guides

The first section of course feedback was on the MPAD guide that students had to use to construct their own MPAD for their RC cars. There were two main guides that were given to the ME2300 students, which were the Arduino IDE Setup Guide and the Sensors and AI Setup Guide. The link for these guides can also be found under the Appendices B and C. The Arduino IDE Setup Guide went over how to download the Arduino IDE and sensor code and how to effectively upload the code into the Elegoo Mega 2560. It contained step by step instructions with images to support and allow students to go through the whole process by themselves. The Sensors and AI Setup guide was the main responsible for all the hardware setup needed to fully utilize MPAD. The guide goes over the main components of the sensor package as well as how to implement each of the sensors described in this paper. This guide also covers the AI and UI systems overview and the limitations of MPAD.

After the completion of the web application, lane following and obstacle avoidance code, and sensor package streaming, the students were given the final formatted SD cards and Raspberry Pis. As before, a major goal of this project was to design a system such that

Mechanical Engineering students are able to implement it without any prior ECE or CS technical knowledge. The course feedback from students gives us valuable insight as to where our team overestimated their abilities or under described certain procedures.

These guides were the main form of contact that we had with Introduction to Engineering Design (ME2300) students. After we provided the guides, they gave us their feedback and allowed us to improve the guides, so that eventually it covered all the steps on how to set up MPAD. Most students had trouble with some Electrical Engineering aspects of the guide, but clarifications were made and most of them were able to successfully implement the sensor package into their own RC cars. The experience of writing guides for an audience that had limited knowledge about the subjects that were covered was one the most challenging aspects of the integration of MPAD into the ME2300 course. However, we believe that the final drafts of the guide were able to reach our goal of providing information about MPAD and how to set it up.

12.2 MPAD Dashboard

The second section of course feedback was on the usability and functionality of the web dashboard. Through initial testing, it became clear that the students could quickly pick up the dashboard and learn the functionality of the controls. The issues mostly came up with loading the dashboard across various devices for the first time. For instance, initial testing with a teaching assistant revealed that "https://" at the beginning of the url is necessary for the dashboard to work. If not, the camera live feed will not be displayed, since the websocket requires an encrypted connection to the server. Given more time, the dashboard could be tested across multiple operating systems, browser vendors, and browser plugins such as Adblock. Although the dashboard works the same across browsers, the team recommends Google Chrome, as it has been tested the most. Aside from these tests, it did seem the basic system worked as intended.

Because the dashboard is first and foremost a means of checking the car's status, design did not need to be the main focus. In fact, we found the lack of front-end web frameworks easier to work with due to the app's simplicity. That said, there is room for making the dashboard's design more refined to guide users in the right direction. If many more options were to be added to the dashboard, it may be beneficial to make a serious attempt at formatting the site consistently. This could mean a set color scheme, button format, etc.

12.3 MPAD Modularity

The third section of course feedback was on the mechanical modularity of MPAD. The steering for the car is done using the front servo. Most of the servos used in testing were 3-wire servos as it is the most prevalent type and do not require specialized ESC's. This means other types of servo connections such as 5-wire servos were ignored. Our servo shield is only capable of accepting 3 wires. While experienced hobbyists and engineers could find an adapter and still continue to use MPAD it is too much to ask for the average user. Allowing more types of connections is important for modularity moving forward.

13. Conclusion

In this project, MPAD was created to be a modular package that allows RC cars to autonomously drive through tracks. The final package contained features such as lane following, obstacle avoidance, sensor data gathering and display, and interactive dashboard. There were relevant improvements compared to the previous MQP where the package does not require any training and is not track specific, increasing the efficiency, practicality, and value of the final product. The added dashboard provides a lane color selection and allows for any user to easily control the car's speed, steering sensibility, and start/stop commands. Alongside that, it also displays collected sensors data in real time that allows users to check the RC cars' performance. This project was able to complete its main educational goal with the help of two guides that allowed the ME2300 students to learn concepts of electrical engineering and computer science by setting up their own MPADs. All of the hard work that was put into the project had its proof of concept when the students were able to utilize MPAD on their own RC cars. This MPAD implementation was also a proof of its modularity aspect, where different RC cars were all able to utilize MPAD to add autonomous driving features to their design. Given all of that, MPAD was a successful project and was able to exceed the team's initial expectations and meet all the goals outlined in Chapter 4 of this paper.

The team now looks forward to what new features, ideas, and possible technologies can be implemented in MPAD to improve its performance and allow for the creation of a product with even more value. In the remaining part of this chapter, the team will go over what we believe are possible improvements that can be done to our current system and what we would have worked towards if we had extra time to work on this project. In the last section of this

chapter, each of the team members provided a reflection of how their time at Worcester Polytechnic Institute allowed them to be prepared and contribute to tackle this project.

13.1 Future Work

This section details potential areas that have room for improvement / expansion in the years to come. These are not bugs, but rather features that could build upon the framework that has already been built. This project has been developed to not only be modular at a high level, but also for each component. This means future improvements can be implemented with less friction.

13.1.1 Lidar/Radar Replacing Ultrasonics

For the obstacle avoidance mechanism in the sensor package, we recommend using either Lidar or Radar. In this project, we choose ultrasonic sensors because of their low price and easy implementation for the ME students. However, the ultrasonic sensors proved to have blind spots. The Lidar sensor uses a laser light and has better accuracy and range. The Radar sensor is also another option we recommend because of their accuracy in sensing motion and figuring out the object's position and shape. The Radar and Lidar can be expensive but they can provide the RC car with an improved obstacle avoidance system alongside the camera.

13.1.2 Sensors Alerts

The current sensors on the car are able to give measurements on the various aspects of the car, however, It would be beneficial to have the sensors trigger other mechanisms in order to troubleshoot the car. For example, if the temperature sensor next to the motor detects a temperature above a certain value, a fan would be triggered to cool it down.

Another aspect would be to detect the battery level, the current Lipo battery does not show when the battery is running low. Instead, it would be beneficial to implement a circuit to notify the user on the car and dashboard if the battery is running low.

13.1.3 Cloud Computing

Currently, all of the processing is being done onboard the RC Car on the Raspberry Pi. In order for the car to drive successfully at high speeds, latency needs to be as low as possible. Therefore, utilizing cloud computing would be a great way to reduce latency by having computing power and data storage on demand without consuming onboard system resources.

13.1.4 5G

5G is the most recent standard instituted for cell network communication. Using high power, narrow band radios, 5G allows for a large number of devices to connect to a radio with high throughput. In other words, 5G networks can connect more RC cars with a faster and more intensive connection. Compared to the current WiFi system, the 5G network would also enable cars to be used outside regardless of WiFi connectivity.

13.1.5 Local Hotspot

Instead of a preconfigured WiFi network, it is theoretically possible to have a WiFi network set up on a portable access point. This WiFi network would not need to be connected to the wider internet to function so long as the control server was also on the same network. Using this set up, the cars could be raced in spaces without normal wifi networks, or even where cell service is unavailable.

13.1.6 Algorithm Switching

An additional feature that the team was looking into during development was allowing the user to select the driving algorithm for their car to use. This could even be a part of a Computer Science course, where teams can work to develop their own driving algorithm, upload it to the server, and then race with different algorithms to compare performance.

13.1.7 Scalable Server

Currently the Heroku server is set to a fixed amount of RAM and disk space. If too many cars are connected concurrently the video stream performance degrades more and more before becoming unusable. By configuring the server to scale up its specifications as load increases, the same performance of a single car would scale to 50 or 100 cars.

13.2 Reflections

In spending nearly a full year on this project, our team has come to learn many crucial educational lessons from the work conducted. Below, there is an outline detailing which classes each team member found relevant to the project, and which areas of the project provided academic growth.

13.2.1 Enzo Giglio de Azevedo

Participating in the Modular Package for Autonomous Driving project was probably the moment that I felt the most like an engineer. I enjoyed the idea of working with a group of people that had different specializations even among the ECE department. This made us depend on our other team members and have them depend on us. This mutual relation creates tight team bonds and allows us to advance and never be content unless we reach our assigned goals. I was

in an interesting position in the team, since I was the only double major, meaning I had to cover two completely different aspects of the project. I believe that this fact actually allowed me to take a lot of ownership of the project and see the bigger picture of the project. From the IE side I mainly worked on deciding what would be our design decomposition by creating an Axiomatic Design for the project and making a Financial Analysis to discuss the viability of creating a marketable product out of this MQP. From the ECE side, I worked mainly in the sensors development and integration with the dashboard and AI side. Additionally, I was also the main responsible for creating the timelines and ensuring deadlines for specific parts of the project as well as keeping the team working as a unit, which could be related to the function of a project manager.

Multiple WPI courses and experiences provided me with the appropriate background to complete this MQP. From the IE side, I would say that courses like BUS 3020 where we go over Axiomatic Design and Lean principles were crucial in my work for this project. For the financial analysis, I used a lot of knowledge that I learned from the OIE 2850 course where we talk about engineering economics and time value of money. I learned how to work with ARENA in the WPI's Simulations course (OIE 3460), which was extremely important when gathering data for the financial analysis. Moving on the ECE side, ECE 2019 and ECE 2049 were the courses that helped me the most during this project. ECE 2019 allowed me to learn how to work and implement sensors in a project and ECE 2049 taught me how to work with microcontrollers and improved my knowledge on C/C++ language. ECE 2799 was also a great learning experience since in the project for that class I actually worked with servos, motors, sensors, and microcontrollers and it had a lot of concepts that were similar to this project were we did business analysis and had to improve the value of our product while on a budget. I also did not

only learn things in courses, but also throughout my time at WPI. Things such as improving my communication skills by participating in clubs and being involved on campus definitely played a big role in my successful integration to this team. Another highlight that allowed me to grow a lot was the IQP experience of writing a paper and working on a long term project definitely made a difference in my overall experience of this MQP. In general, I am glad to have worked with this amazing team and I hope that the project will continue in good hands and see multiple improvements through the years.

13.2.2 Antonio Jeanlys

Testing all the sensors, implementing, and designing a fully modular sensor package took many things that I have learned in the past four years and brought it together. The great thing about WPI is that it's project based curriculum prepares you immensely for working in groups. I was able to work within my expertise while trusting my group to be there for assistance and handle their part of the workload. This dynamic is something I learned during my time in ECE 2799 , where we compiled gantt charts and built a parking assist sensor using many of the sensors that were implemented in this MQP. I was also able to put theory to practice from some of my coding classes such as ECE 2049, CS 1004, CS2301 and ECE 2029. Working in the Arduino IDE software development platform provided a good way to understand what those classes offered in real life conditions. It was also interesting to take things I learned as an underclassmen in classes such as ECE 2010 and ECE 2019 and easily apply and comprehend certain topics. Topics such as voltage dividers , ensuring that sensors were functioning correctly and all receiving ample amounts of current and the correct voltage showed me that I have retained the knowledge that I have learned. Overall, this project allowed me to explore aspects of

both my electrical engineering and computer engineering degree. It was a great experience to prepare me for the challenges that I may face in the work field.

13.2.3 Chris Mercer

This project nicely tied in other aspects of ECE that I was not comfortable with despite having taken ECE and Computer Architecture classes in years prior. The reason for this was that the classes did not develop a product from the design stage all the way through to user testing. By developing the self driving module from the ground up, I got to learn about design decisions and the impact they make on a product's development cycle.

That said, there was a great deal of classes that prepared me to take on the challenges presented by the system. Operating Systems (CS3013) was useful for understanding low level memory buffers and commands for interacting with the serial data coming in from the Arduino. Webware (CS4241) gave the background necessary for writing the dashboard that controls the car and writing the backend server components that connect cars to clients. Lastly, Algorithms (CS2223) let me work on the code for autonomous driving.

This past work in the Computer Science department was not the only part of my education that proved useful. ID2050 and the subsequent IQP work completed in my junior year made soliciting requirements and planning the project a lot easier. There is no replacement for genuine work experience, and this project nicely incorporated budgeting and collaboration challenges similar to what could be expected in a workplace.

13.2.4 Julien Mugabo

Developing a Modular Package for Autonomous driving provided me with an overall understanding on how to build a system from the initial research phase all the way to the design

and implementation. In this project, I was able to work in a cross functional team with students from the ME department and CS department while bringing my knowledge from ECE. I got to understand some of the mechanisms of a car such as how the motor and steering works in different conditions and how it can be improved to better handle the track and obstacles. From the CS side, I got to see how they implemented their dashboard system and lane following code.

In this project I used my knowledge from my electrical courses to implement the sensor package. The sensors,circuits and systems(ECE 2019) were helpful in understanding the required sensors and their circuits. The embedded computing in design course (ECE 2049) knowledge was useful in programming the sensors on the arduino board. Lastly, the Electrical and Computer Engineering Design(ECE 2799) prepared me for this project with experience in designing an electrical system. Overall, I have gained technical experience that can be useful in autonomous vehicles and I also gained skills that will be applied in my career.

13.2.5 Eric Reardon

This project nicely builds upon the skills that I have learned over the past four years. Developing an autonomous driving package was of a high interest to me from the beginning. Many of the classes I have taken at WPI, including Intro To Artificial Intelligence (CS 4341) and Machine Learning (CS 4342) reference Autonomous Driving through neural networks and machine learning. Therefore, I was excited to explore opportunities for self-driving and how the old driving software, DonkeyCar, worked. Although we strictly used real-time Computer Vision for driving, I got to do some really interesting research on how some industry leaders, such as Tesla, are innovating the space. I also did a project on the ethics of Autonomously Driving Vehicles in Social Implications of Information Processing (CS 3043) which was very relevant to the background of this project.

Additionally, the Computer Science aspect of this project was very open-ended, allowing us to work in numerous spaces at once. Aside from Computer Vision and self-driving, I got to build a web application. My experience in Webware (CS 4241) and Software Engineering (CS 3733) greatly helped in this area. Webware was a great foundation in CSS, Javascript, and HTML, which are all essential for front-end development and design. Additionally, I had some experience in Software Engineering writing API's and handling data transfers between server, client, and external structures. And as always, Object-Oriented Design Concepts (2102), Algorithms (CS 2223), and Object-Oriented Design and Analysis (CS 4233) were very helpful in writing clean and efficient code.

Finally, IQP and ID 2050 were great courses to prepare and introduce me to the time management and writing that was necessary for this MQP. WPI also does a great job incorporating small group projects in the majority of courses. Learning how to effectively work and communicate in a team environment is crucial for MQP and especially important for the future work I will be doing after graduation.

13.2.6 Taylan Sel

Developing MPAD was interesting as it allowed me combine my personal interest when it came to Raspberry Pi's and Computer Vision with what I learned in WPI. I had used a Raspberry Pi along with Microsoft Azure Custom Vision in an earlier internship and then my ECE 2799 class. The same internship increased my comfortability with Linux. Although I taught Python and C++ to myself I did learn C in WPI which provided a nice foundation for the rest of my coding education. Classes like CS 2301, ECE 2049, and ECE 3849 built this foundation. Although our circuit did not necessarily need to be analyzed or calculated, classes like ECE 2010 and ECE 2019 came in handy. While the content of the class is not quite relevant to this project,

ECE 2305 had a project requiring an Arduino which came in useful. WPI got me accustomed to fast paced projects, this allowed me to develop and test quickly.

MPAD allowed me to refine my Python and C++ skill. It also made me comfortable with different libraries and sensors I had never used before. Working on an open ended project where we had the freedom to explore our own solutions was a breath of fresh air and nurtured my investigative skills.

14. References

1) P. Coppola, “Autonomous Vehicles and Future Mobility,” Science Direct, 2019. [Online]. Available: https://www-sciencedirect-com.ezpxy-web-p-u01.wpi.edu/book/9780128176962/autonomous-vehicles-and-future-mobility. [Accessed: 07-Mar-2021].
2) Donkey® Car. [Online]. Available: https://www.donkeycar.com/. [Accessed: 07-Mar-2021].
3) “SunFounder PiCar-S Kit V2.0 for Raspberry Pi,” SunFounder. [Online]. Available: https://www.sunfounder.com/products/raspberrypi-sensor-car. [Accessed: 07-Mar-2021].
4) D. Tian, “DeepPiCar-Part 1: How to Build a Deep Learning, Self Driving Robotic Car on a Shoestring Budget,” Medium, 08-May-2019. [Online]. Available: https://towardsdatascience.com/deeppicar-part-1-102e03c83f2c. [Accessed: 07-Mar-2021].
5) S. McBride, “Why 2021 Will Be The Year Self-Driving Cars Go Mainstream,” Forbes, 06-Jan-2021. [Online]. Available: https://www.forbes.com/sites/stephenmcbride1/2021/01/06/why-2021-will-be-the-year-self-driving-cars-go-mainstream/?sh=4ebe6fd74e24.
6) T. Kim, A. Marge, J. Randon, and K. Wood, “Modular Self-Driving and Sensor Packages for R/C Cars,” Google Docs, 18-May-2018. [Online]. Available: https://docs.google.com/document/d/16pEk7z4ABvYf4ebSDLYsDc6Tw24wG6V0e63b5pQLxsw/edit. [Accessed: 07-May-2021].
7) *Donkey® Car*. [Online]. Available: https://www.donkeycar.com/. [Accessed: 07-May-2021].
8) “ELEGOO MEGA 2560 R3 Board with USB Cable Compatible with Arduino IDE,” *ELEGOO Official*. [Online]. Available: https://www.elegoo.com/collections/controller-boards/products/elegoo-mega-2560-r3-board?variant=32365709492272. [Accessed: 07-May-2021].
9) Raspberry Pi, “Raspberry Pi OS,” *Raspberry Pi*. [Online]. Available: https://www.raspberrypi.org/software/. [Accessed: 07-May-2021].
10) N. P. Suh, “Axiomatic Design Theory for Systems,” *Research in Engineering Design*. [Online]. Available: https://link.springer.com/article/10.1007%2Fs001639870001. [Accessed: 07-May-2021].
11) *Shibboleth Authentication Request*. [Online]. Available: https://www-sciencedirect-com.ezpxy-web-p-u01.wpi.edu/book/9780128176962/autonomous-vehicles-and-future-mobility. [Accessed: 07-May-2021].
12) *Shibboleth Authentication Request*. [Online]. Available:

https://link-springer-com.ezpxy-web-p-u01.wpi.edu/chapter/10.1007/978-3-319-55372-6_11. [Accessed: 07-May-2021].

13) A. 15 and K. Burke, "How Does a Self-Driving Car See?: NVIDIA Blog," *The Official NVIDIA Blog*, 16-Apr-2019. [Online]. Available: https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/#:~:text=The%20three%20primary%20autonomous%20vehicle,as%20their%20three%2Ddimensional%20shape. [Accessed: 07-May-2021].
14) "Train an autopilot with Keras," *Train an autopilot. - Donkey Car*. [Online]. Available: https://docs.donkeycar.com/guide/train_autopilot/. [Accessed: 07-May-2021].
15) D. Arrachequesne, "Introduction," *Socket.IO*, 30-Apr-2021. [Online]. Available: https://socket.io/docs/v4. [Accessed: 07-May-2021].
16) A. S. Gillis, "What is REST API (RESTful API)?," *SearchAppArchitecture*, 22-Sep-2020. [Online]. Available: https://searchapparchitecture.techtarget.com/definition/RESTful-API#:~:text=A%20RESTful%20API%20is%20an,deleting%20of%20operations%20concerning%20resources. [Accessed: 07-May-2021].
17) "Welcome to Flask," *Welcome to Flask - Flask Documentation (1.1.x)*. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/. [Accessed: 07-May-2021].
18) R. Burnett, "Understanding How Ultrasonic Sensors Work," *MaxBotix Inc.*, 04-Mar-2021. [Online]. Available: https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm. [Accessed: 07-May-2021].
19) Raspberry Pi, "Raspberry Pi 4 Model B specifications," *Raspberry Pi*. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/. [Accessed: 07-May-2021].
20) Kuhlman, Dave. "A Python Book: Beginning Python, Advanced Python, and Python Exercises". Section 1.1. Archived from the original (PDF) on 23 June 2012.
21) "About," *OpenCV*, 04-Nov-2020. [Online]. Available: https://opencv.org/about/. [Accessed: 07-May-2021].
22) Stroustrup, Bjarne (1997). "1". The C++ Programming Language (Third ed.). ISBN 0-201-88954-4. OCLC 59193992.
23) "ELEGOO Mega 2560 Basic Starter Kit Compatible with Arduino IDE," *ELEGOO Official*. [Online]. Available: https://www.elegoo.com/collections/mega-2560-starter-kits/products/elegoo-mega-2560-basic-starter-kit. [Accessed: 07-May-2021].

24) R. Burnett, “Understanding How Ultrasonic Sensors Work,” *MaxBotix Inc.*, 04-Mar-2021. [Online]. Available: https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm. [Accessed: 07-May-2021].

25) A. Industries, “Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055,” *adafruit industries blog RSS*. [Online]. Available: https://www.adafruit.com/product/2472. [Accessed: 07-May-2021].

26) ArduinoModules, “KY-003 Hall Magnetic Sensor Module,” *ArduinoModulesInfo*, 30-Aug-2020. [Online]. Available: https://arduinomodules.info/ky-003-hall-magnetic-sensor-module/. [Accessed: 07-May-2021].

27) “Ultrasonic Distance Sensor - HC-SR04,” *SEN-15569 - SparkFun Electronics*. [Online]. Available: https://www.sparkfun.com/products/15569. [Accessed: 07-May-2021].

28) Amazon, “LiPo Battery Checker - RC 1-8S Battery Tester Monitor - Low Voltage Buzzer Alarm - with LED Indicator - for Lipo Life LiMn Li-ion Battery (3 Pack),” *Amazon*. 18-May-2018.

29) B. Earl, “Adafruit PCA9685 16-Channel Servo Driver,” *Adafruit Learning System*. [Online]. Available: https://learn.adafruit.com/16-channel-pwm-servo-driver?view=all. [Accessed: 07-May-2021].

30) “Welcome to Flask,” *Welcome to Flask - Flask Documentation (1.1.x)*. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/. [Accessed: 07-May-2021].

31) “Commercial Property Insurance Cost,” *Insureon*. [Online]. Available: https://www.insureon.com/small-business-insurance/commercial-property/cost#:~:text=The%20median%20cost%20of%20commercial,outlier%20high%20and%20low%20premiums. https://ultimaker.com/3d-printers/ultimaker-s5. [Accessed: 07-May-2021].

32) “Welcome to Flask-SocketIO's documentation!,” *Flask*. [Online]. Available: https://flask-socketio.readthedocs.io/en/latest/. [Accessed: 07-May-2021].

33) “Documentation,” *Documentation | Heroku Dev Center*. [Online]. Available: https://devcenter.heroku.com/categories/reference. [Accessed: 07-May-2021].

34) “Commercial Property Insurance Cost,” *Insureon*. [Online]. Available: https://www.insureon.com/small-business-insurance/commercial-property/cost#:~:text=The%20median%20cost%20of%20commercial,outlier%20high%20and%20low%20premiums. https://ultimaker.com/3d-printers/ultimaker-s5. [Accessed: 07-May-2021].

35) W. by A. P. Power, “Average Electric Bill, Rates and Consumption in Massachusetts,”

The Power to Help, 07-Jun-2019. [Online]. Available: https://providerpower.com/power-to-help/average-electric-bill-rates-consumption-massachusetts/#:~:text=Residential%20Electricity%20Rates%20in%20Massachusetts,highest%20rate%20in%20the%20nation. [Accessed: 07-May-2021].

36) Amazon, “Pricing,” *Let's Talk Numbers*. Amazon, 2021.

37) R. Burnett, “Understanding How Ultrasonic Sensors Work,” *MaxBotix Inc.*, 04-Mar-2021. [Online]. Available: https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm. [Accessed: 07-May-2021].

15. Appendix

Appendix A: IMU Accuracy Testing Appendix

Table 15.1: Expected IMU X, Y, and Z Values

	Expected	
X	Y	Z
-90	-90	-90
90	90	90
-180	-90	-180
90	90	-90
-90	-90	90
90	90	180
-180	-90	-90
90	90	90
-90	-90	-90
90	90	90
-180	-90	-90
90	90	-180
-90	-90	90
90	90	-90
-180	-90	180
90	90	-90
-90	-90	90
90	90	-90
180	-90	-180
90	90	90
-90	-90	-90
-180	90	180
-90	-90	-90
90	90	90
-180	-90	-180
90	90	-90
-90	-90	90
-180	90	180
-90	-90	-90
90	90	90

Table 15.2: Calculated IMU X, Y, and Z Values

Calculated (Sensor)		
X	Y	Z
-91	-84	-89
88	89	88
-180	-87	-178
90	88	-88
-90	-86	90
89	88	176
-179	-87	-87
88	89	88
-91	-87	-88
88	89	90
-179	-88	-86
89	89	-179
-90	-87	91
89	86	-87
-179	-87	177
87	86	-85
-92	-87	96
89	87	-84
178	-89	-177
90	86	90
-90	-89	-85
-178	87	178
-91	-88	-84
90	87	92
-178	-89	-178
90	87	-86
-88	-88	88
-178	87	176
-91	-87	-87
90	86	87

Table 15.3: Percentage Error of IMU X, Y, and Z Values

Percentage Error		
X	Y	Z
1.11%	6.67%	1.11%
2.22%	1.11%	2.22%
0.00%	3.33%	1.11%
0.00%	2.22%	2.22%
0.00%	4.44%	0.00%
1.11%	2.22%	2.22%
0.56%	3.33%	3.33%
2.22%	1.11%	2.22%
1.11%	3.33%	2.22%
2.22%	1.11%	0.00%
0.56%	2.22%	4.44%
1.11%	1.11%	0.56%
0.00%	3.33%	1.11%
1.11%	4.44%	3.33%
0.56%	3.33%	1.67%
3.33%	4.44%	5.56%
2.22%	3.33%	6.67%
1.11%	3.33%	6.67%
1.11%	1.11%	1.67%
0.00%	4.44%	0.00%
0.00%	1.11%	5.56%
1.11%	3.33%	1.11%
1.11%	2.22%	6.67%
0.00%	3.33%	2.22%
1.11%	1.11%	1.11%
0.00%	3.33%	4.44%
2.22%	2.22%	2.22%
1.11%	3.33%	2.22%
1.11%	3.33%	3.33%
0.00%	4.44%	3.33%

Appendix B: Sensor and AI Setup Guide

Link to the Sensor and AI Setup Guide:
https://docs.google.com/document/d/1WZnvo8CedjVhxcjU5xzy_SYfhUjnrrSSaWZBByvNBJ4/edit?usp=sharing

Appendix C: Arduino IDE Setup Guide

Link to the Arduino IDE Setup Guide:
https://docs.google.com/document/d/1t1BtwT7rGrhlTMjrzaKlomSs7tRYUCNe-1HfrWLGjHY/edit?usp=sharing

Appendix D: MPAD Gantt Chart

Link to the MPAD Gantt Chart and Top Level MPAD Gantt Chart:
https://docs.google.com/spreadsheets/d/18KkGmdqnskZngXL-dsDVA-IjquH-lm7xeNi4pRMJGUM/edit?usp=sharing

Appendix E: MPAD Code Repository

Link to the MPAD code repository on Github:
https://github.com/rkprad/20-21-SelfDriveCar-AI-Controls-Sensors

Link to the Dashboard/Driving code repository on Github:
https://github.com/Mersair/selfdriving-rc

Appendix F: MPAD WPI CS Department Poster

Link to the MPAD WPI CS Department poster:
https://drive.google.com/file/d/1VdvRhI_GCZ6lrplP6APo9I78LSwDOhts/view

Appendix G: MPAD WPI CS Department Video

Link to the WPI CS Department video:
https://drive.google.com/file/d/1MNwBMWbMOQI4ZcqCnT4okGd06yYaYpmL/view

Appendix H: MPAD WPI ECE Department Video

Link to the WPI ECE Department video:
https://drive.google.com/file/d/1N7Z1XSAzNRnlpeHCbUcbp9OOtgDH7cLi/view?usp=sharing

Appendix I: MPAD WPI ECE Project Presentation

Link to the WPI ECE Department presentation:
https://docs.google.com/presentation/d/13An11yUy1TKUta3dZa75II7ywtBjqH-pqxfPUoeMPfI/edit?usp=sharing

Appendix J: MPAD WPI IE Department Elevator Pitch

Link to the WPI IE Department elevator pitch:
https://drive.google.com/drive/u/1/folders/1KCqIwOsmAnT-cx-OqbgTPfW2rmBfUnik

Appendix K: MPAD WPI IE Department Poster

Link to the WPI IE Department poster:
https://drive.google.com/file/d/1fgah648QR1g9JcTYMVMemKU3JFwlANxb/view?usp=sharing

Appendix L: MPAD WPI IE Department Video

Link to the WPI IE Department video:
https://drive.google.com/drive/u/1/folders/1KCqIwOsmAnT-cx-OqbgTPfW2rmBfUnik

Appendix M: IRB

Link to the IRB Approval Letter:
https://docs.google.com/document/d/1pwauQjVPDc-O-sn1RnXjpc_qUHksRxfRD8_ERecA__Y/edit?usp=sharing

Link to Informed Consent Letter:
https://docs.google.com/document/d/1gLYqpbVfUSM2r6yFw4iyrH31j6yDOKXsrlKK2ys45UI/edit?usp=sharing

Link to Post Sensor Guide Completion Feedback survey:
https://docs.google.com/forms/d/1xunu6J8RFtYDbp2Z1CgeosS7gJ1u_UtFmESPG_50VvE/edit?usp=sharing

Appendix N: Average Loop Times for Self Driving Algorithms

Below is the time in ms to loop through the given algorithm. Each was averaged over 100 loops with the same input frame, with the test being conducted three times for each algorithm.

Table 15.4: Loop Time for Different Driving Algorithms

	Ray-tracing	Trigonometric	Lane distance comparison
Pass 1	104ms	98ms	21ms

Pass 2	110ms	96ms	21ms
Pass 3	108ms	97ms	23ms

Appendix O: Ray Tracing Algorithm

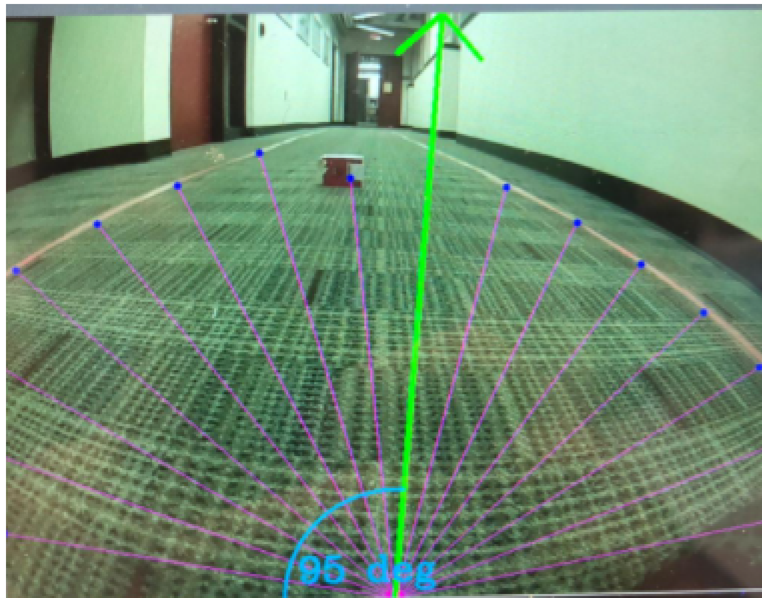


Figure 15.1: Ray tracing visual GUI screenshot

The ray tracing algorithm works by casing each ray (pictured in pink) from the car's position. Each ray continues moving along from the origin until it detects an obstacle, be it a lane marking or a road block. Then the car moves along the ray that goes the furthest from the origin. Internally, the code uses a custom line iterator that creates an array of each pixel along a given path. This pseudo code illustrates the process used:

```
frame = camera.capture()
origin = (frame.width/2, frame.height) // bottom middle of image
ray_endpoints = calculateEndpoints(frame.width, frame.height)
rays = []
ray_lengths = []
for endpoint in ray_endpoints:
      rays.append(new LineIterator(origin, endpoint.xy))
      for pixel in rays[:1]:
            if pixel == lane_color:
                  break
            else:
                  ray_lengths[endpoint] = 1 + ray_lengths[endpoint]
servo.angle = highest_value(ray_lengths)
```

Appendix P: Dashboard Sensor Data Live Updates Video

Below is a video link that demonstrates the car’s dashboard being updated in real time as sensor data is read in on the car:
https://drive.google.com/file/d/1hz6qzPXlLc8x7dPNkvBr_7ZNLYKnhYUf/view?usp=sharing

Appendix Q: Dashboard Sensor Color Selection Tool Video

Below is a video link that demonstrates the car’s lane colors being set remotely using the color selection tool on the web dashboard:
https://drive.google.com/file/d/1fyLyoJ2flh4IIxr--HBAwvCTPqXVOX-v/view?usp=sharing

Copyright Information

The work presented here is copyrighted by Enzo Giglio de Azevedo, Antonio Jeanlys, Chris Mercer, Julien Mugabo, Eric Reardon, Taylan Sel, Professor Kaveh Pahlavan, Professor Pradeep Radhakrishnan, and Professor Walter T. Towner