WPI

Modular Package for Autonomous Driving V2

A *Major Qualifying Project*

Submitted to the Faculty of Worcester Polytechnic Institute

In fulfillment of the requirements for the degree in

Bachelor of Science in

Computer Science and Electrical Engineering

By

Bryan Lima (CS), Anthony LoPresti (RBE), Thomas Riviere (ECE),

Lindberg Simpson (CS), and William Yang (CS)

Advised by

Professor Kaveh Pahlavan (ECE/CS) and Professor Pradeep Radhakrishnan (ME/RBE)

May 2, 2022

Abstract

The Modular Package for Autonomous Driving (mPAD) allows any user to implement autonomous driving to various scaled vehicles with the appropriate hardware components. The need for this project comes from a lack of flexible and affordable educational tools as well as a limited supply of autonomous driving software packages available for non-commercial use. The technical design objectives considered were to develop a robust, modular, and intuitive package to be used in an academic setting by engineering students with minimal prior software development, electrical engineering and hardware design knowledge. mPAD V1 included a package that enabled self-driving capabilities through lane detection, and an online dashboard was used to display sensor readings and control the speed and steering of the car. However, the platform crashed frequently and it was difficult to operate due to insufficient documentation.

To address the drawbacks of mPAD V1, our team improved the performance of the backend by updating the sensor package, reducing network traffic and revamping the communication system between the car, server, and client. We also improved the dashboard by including a tutorial walkthrough, hardware component testing, a manual driving mode, power level indicators, and the ability to change various advanced settings related to the car. Finally, we developed a comprehensive setup guide for our system that also includes troubleshooting suggestions.

We have advanced mPAD to be capable of ensuring that the remote controlled car is capable of performing at least 5 laps without any manipulation. It is modular as the system is composed of entirely off-the-shelf components and is intuitive as the system is easy to set up and operate with little background knowledge.

Acknowledgements

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

Professor Pradeep Radhakrishnan, WPI Department of Mechanical and Materials Engineering
Professor Kaveh Pahlavan, WPI Department of Electrical and Computer Engineering
Ms. Barbara Furhman, Administrative Assistant, WPI Department of Mechanical and Materials Engineering
Cam Tu Le, Teaching Assistant, WPI Mechanical and Materials Engineering Department

2021-2022 MQP Team: Developing a Fully 3D Printed Vehicle for Autonomous Delivery

Ben Amado
Joseph Calcasola
Anthony LoPresti
Matthew Maloney
Kwesi Sakyi

2020 - 2021 MQP Team: Modular Package for Autonomous Driving (mPAD)

Enzo Giglio de Azevedo
Antonio Jeanlys
Chris Mercer
Julien Mugabo
Eric Reardon
Taylan Sel

Authorship

Section	**Writer(s)**
1. Introduction	Anthony
2. mPAD V1 Summary	Anthony, Bryan, Thomas, William
3. Background	Anthony, Thomas, William
4. Initial Package Review and Analysis	Anthony, Bryan, Lindberg, Thomas, William
5. Initial Software Analysis and Development	Lindberg, William
6. Dashboard Changes	Bryan
7. Updated Sensor Package	Anthony, Lindberg, Thomas
8. USB Camera Implementation and Autonomous Driving Testing	William
9. Object Detection Testing	Anthony, Thomas, William
10. Discussion	Anthony, Bryan, Lindberg, Thomas, William
11. Conclusions	Anthony, Bryan, Lindberg, Thomas, William

Table of Contents

1. Introduction

Automation has been a key factor of technology growth and innovation throughout history. In today's day and age, a vehicle is the fastest and most convenient way to get from point A to B. The introduction of this technology to vehicles is the obvious next step in the world of automation. Major manufacturers have already begun to implement some of these features. Lane departure warning and adaptive cruise control are becoming more and more common in vehicles every year. Additionally, fully autonomous vehicles are under development by many major car manufacturers. Tesla, who has one of the more advanced forms of autonomous vehicle technology on the road today, has a ‘Full Self Driving’ feature that allows the vehicle to drive practically fully autonomously on its own already. While these innovations are impressive, this technology is far from complete.

Autonomous vehicles in 2022 are expensive and not readily available to the average consumer. Even if someone is able to obtain an autonomous vehicle, information on how the system actually works is often not available to the person as it is intended only for use instead of development. The need for this project comes from a lack of flexible and affordable educational tools as well as a limited supply of autonomous driving software packages available for non-commercial use. With this in mind, the Modular Package for Autonomous Driving Version 2 (mPAD V2) was created to be an educational tool that can be used to implement autonomous driving on a Remote Controlled (RC) sized vehicle.

There were three main concerns when considering the design of mPAD V2. First we wanted the software package to be robust and scalable. This would be demonstrated by a vehicle consistently completing multiple laps around the track. Additionally, the package should be able to function properly on RC vehicles of all different shapes and sizes.

Second, the package needs to be modular so users still have flexibility for their own creativity and modifications. In fact, a variety of hardware and sensors that can enhance the package are not essential for basic self-driving functionality. As such, users should be able to add or remove various parts of the package based on their needs and interests.

Finally, the package needs to be intuitive and easy for an inexperienced user to set up and run. A detailed setup and troubleshooting guide should be included that details the first time setup process and to help the user identify and resolve common issues. In order to increase the

ease-of-use of the system, an intuitive dashboard that allows a user to interface with the vehicle should also be included.

During the WPI's 2020-2021 academic year, Giglio de Azevedo et al. created the first iteration of mPAD (henceforth referred to as mPAD V1), which includes a package that enables self-driving capabilities through lane detection, and an online dashboard that is used to display sensor readings and control the speed and steering of the car (Giglio de Azevedo et al., 2021). Unfortunately, our team determined that mPAD V1 did not meet the criteria outlined above. As such, we did work to improve the reliability of the system and accuracy of the self-driving code to improve the robustness of the system; introduced a tutorial, various testing features features, manual driving, and the ability to change some advanced software settings to improve the intuitiveness; and implemented additional cameras and sensors to aid in object detection capabilities while also improving the modularity of the system.

The report begins by giving some background information in the form of a literature review. After the necessary background information is given we move into our initial preliminary design and analysis chapter where we carefully analyze the current state of mPAD and develop our project objectives. Following that we move into our testing and results portion of the report beginning with our dashboard updates. After dashboard changes we discuss the changes made to the sensor package and self-driving package in mPAD V2 as well as some object detection testing results. Finally we move into discussions, conclusion, and recommendations for the project moving forward.

2. mPAD V1 Summary

This chapter will analyze the sensor, dashboard, and self-driving implementation of mPAD V1 by Giglio de Azevedo et al. The purpose of this analysis was to help us understand the reasoning behind the decisions made by the past two teams and to help us continue to build upon the foundations they provide and avoid any mistakes the previous team made during development. This analysis speeds up the early design process allowing us more time to implement new features.

2.1 Sensor Implementation

Figure 2.1 shows where the sensors would be situated on the car. As you can see two temperature sensors were used to monitor the temperature of the motor and battery, an IMU was connected to the PCB board for the acceleration and orientation, a hall effect sensor was hooked up to the motor to measure the RPM, and seven ultrasonics were used to measure distance between the car and obstacles. The purpose of each sensor and why it was chosen will be discussed in this section. Also included in this diagram are the location of the camera which was used for the self-driving implementation and the PCB board which was used to read the data from the sensors. Sensor cost and specifications can be found in Appendix Section 13.1 of this report.

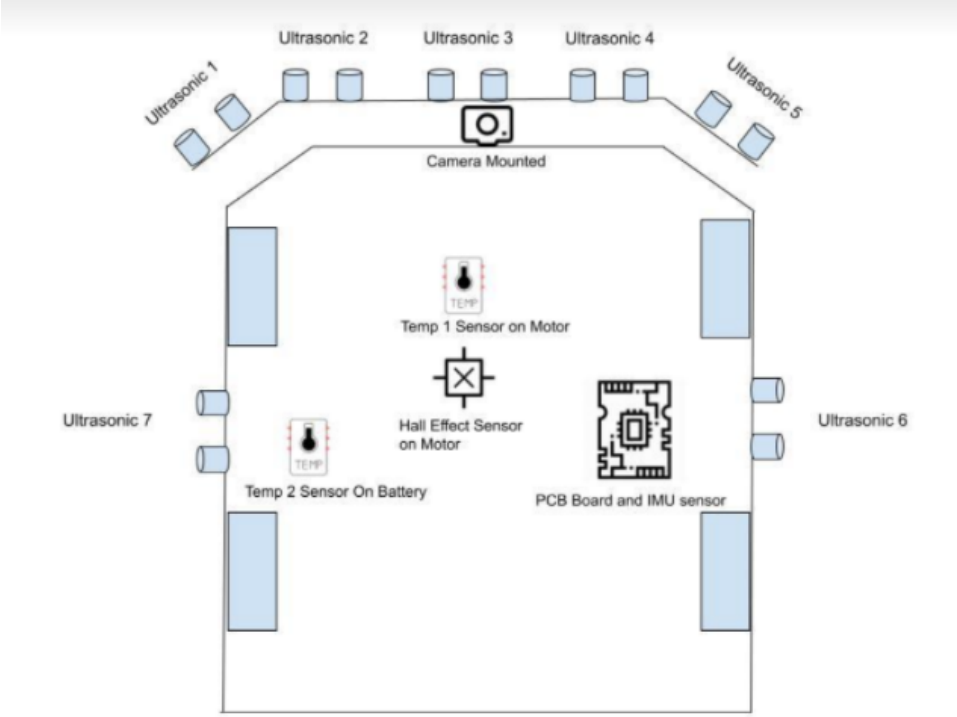

Figure 2.1: Sensor Locations mPAD V1

(Giglio de Azevedo et al., 2021)

2.1.1 ELEGOO Mega 2560

When designing the sensor package the first design decision made by last year's team was to use a separate board for the sensors to reserve more processing power on the Raspberry Pi for autonomous driving. The secondary board is the ELEGOO Mega 2560, chosen because of its price and compatibility with Arduino IDE, as shown in Figure 2.2.

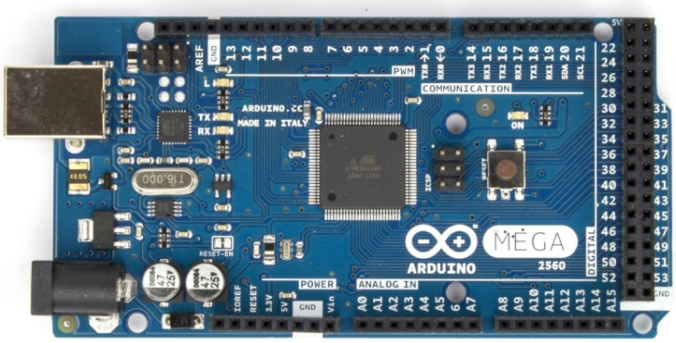

Figure 2.2: ELEGOO Mega 2560

reproduced as if from link

2.1.2 HC-SR04 Ultrasonic Sensors

Ultrasonic sensors are used for object detection. In last year's implementation, a total of five sensors were used. All of the sensors are installed on the front of the car using a custom 3D printed bumper. Additionally, the past team suggested adding an additional two ultrasonic sensors on each side of the RC car. The reason so many sensors are used is that their detection range is only within a cone of 30°. Three of the sensors are side by side on the front bumper and the final two are on each end of the bumper oriented at a 45° angle to allow a broader detection cone in front of the vehicle. The ultrasonic sensor chosen was the HC-SR04, which can measure a range of 2cm to 400cm. This sensor - an example of which can be seen in Figure 2.3 - was selected because of its low price and because of the team's previous experience with the sensor.

Figure 2.3: HC-SR04

reproduced as if from link

2.1.3 DHT11 Temperature Sensors

To monitor the temperature of the RC car the previous team chose to use the DHT11 temperature sensor, as shown in Figure 2.4. This component is extremely cheap in price, and in turn is not the most accurate. The team decided accuracy was not a desired trait because the temperature sensor is there mainly to keep the temperature of the RC car under a certain threshold. In total, three of these sensors are used on the car. One is near the motor and the other two are near the batteries.

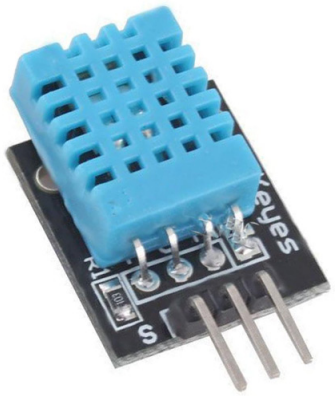

Figure 2.4: DHT11

reproduced as if from link

2.1.4 BNO055 IMU

The IMU gives the current position and orientation of the RC. This information allows the car to drive autonomously up and down an incline. Figure 2.5 shows the BNO055, which was chosen as the IMU for the project. This specific IMU model was chosen due to its ease of use and great compatibility with DHT11 temperature sensors having the same manufacturing company in Adafruit.

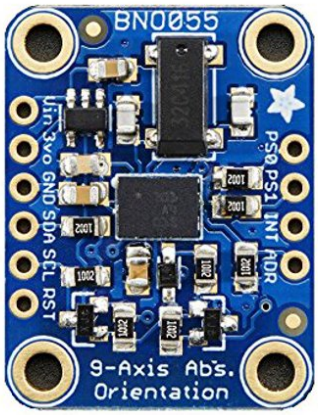

Figure 2.5: BNO055

reproduced as if from link

2.1.5 KY-003 Hall Effect Sensor

The hall effect sensor measures the RPM of the Motor. The RPM is measured by attaching a magnet to a rotating gear and placing the sensor in the path of the rotating magnet.

RPM is determined based on how frequently the magnet is detected as it rotates proportionally to the speed of the motor. The hall effect sensor used is the KY-003 - pictured in Figure 2.6 - chosen for its low cost and accurate readings.

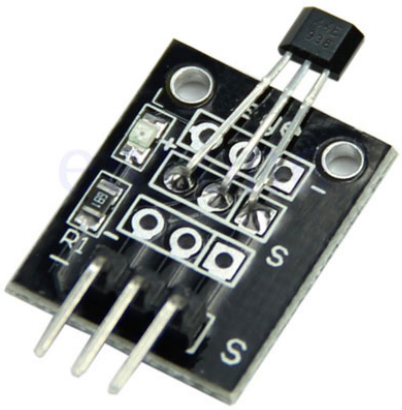

Figure 2.6: KY-003

reproduced as if from link

2.2 Dashboard Implementation

Giglio de Azevedo et al. set some goals and requirements for their dashboard implementation. The team wanted the data collected from all the sensors to be displayed in well-labeled graphs within 500ms of reading from the car's sensor array. These readings were to be polled every 1000ms or less. Car controls such as starting and stopping autonomous driving, adjusting the throttle, and adjusting the steering aggressiveness were set as requirements for the dashboard. They also wanted to display a live camera feed of at least 10 frames/second with a latency of no greater than 500ms. Users should be able to click the camera's live feed for the lane color picker, then the user clicks as many times as they want on the camera feed with each click adding the color of the selected pixel to the car's lane color range. An example of what the mPAD V1 dashboard looks like can be seen below in Figure 2.7.

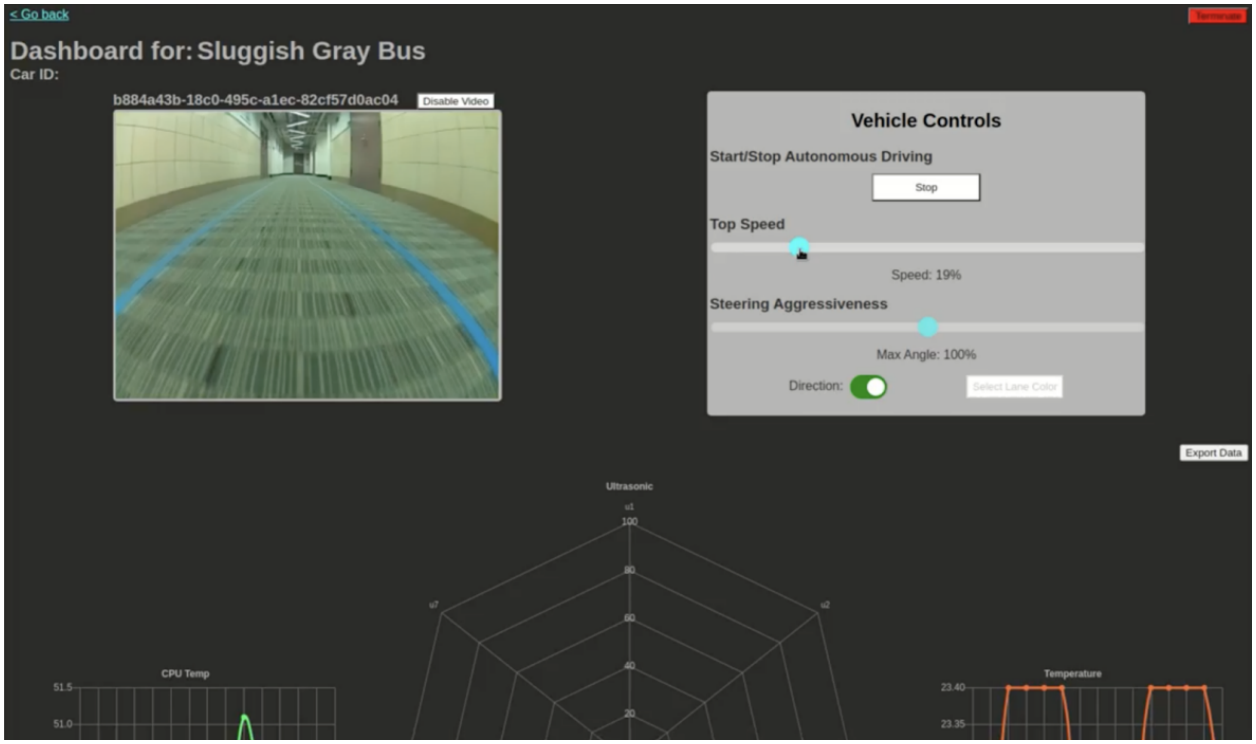

Figure 2.7: mPAD V1 Web Dashboard

(Giglio de Azevedo et al., 2021)

2.2.1 Technologies

In order to stream to and from a Raspberry Pi 4, Giglio de Azevedo et al. utilized SocketIO, which enables streaming content of any kind, including video and sensor data. Through this, a socket can be instantiated on a web page, giving users real-time views of the car's camera and sensor data. JavaScript was utilized to display the video stream and render the detailed sensor data graphs. Last year's team also wrote the underlying server framework in Python since SocketIO is a Python library and utilized the Flask framework.

The dashboard application is reserved so that only a car configured with the team's self-driving module will be able to communicate and register to the application. Once the Raspberry Pi is powered on the RC car will automatically register to our server and begin streaming camera and sensor data. From our experience, we have noticed that it takes about 90 seconds for the car to appear as an option in the dropdown menu. The front-end of the website was designed using HTML, JavaScript, and vanilla CSS. The front-end libraries utilized by last

year's team were all open source and well documented. They utilized ChartJS for the sensor data charts and SocketIO for the live video feed stream.

When a car connects to and is registered in the application, a unique ID (UUID) is generated to keep track of each car's individual configuration. When the user makes a change, an API request is sent from the dashboard to the server with the car's UUID. The server then makes the change and sends the change to the car through WebSocket. The user can make changes to the car's settings while it is autonomously driving since a Streamer class was used in the implementation.

2.2.2 Lane Selection Process

There is also the Lane Selection Tool on the dashboard which allows the user to choose what color will serve as the car's street borders. When there are no large gaps and there are clear white lanes in the right frame, the user can be certain that they have properly set the lane boundaries. The camera reads the data in HSV format (Hue, Saturation, Value), a set of three integers representing a pixel's color. Two arrays known as color channels are created for the HSV values, one array starts as the maximum ([255, 255, 255]), the other starts as the minimum ([0, 0, 0]). This creates a black 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, and vice versa for the higher channel array. As the user clicks, they will see the lane start to form in white as the camera inputs these new HSV values.

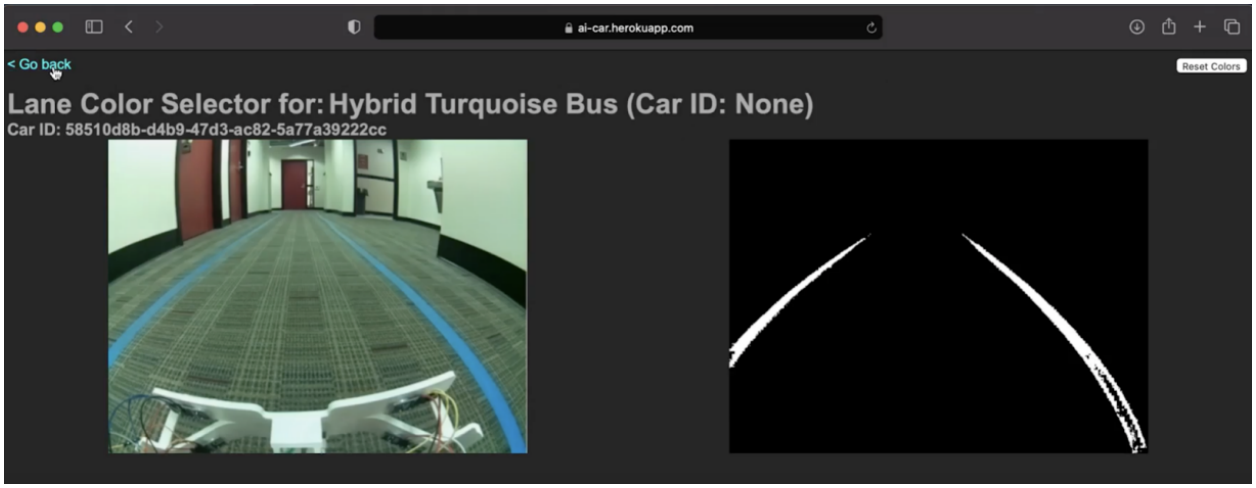

Figure 2.8: mPAD V1 Lane Selection Page
(Giglio de Azevedo et al., 2021)

It is very important to select lane colors anywhere on the track where there is different lighting. As can be seen in the top right of Figure 2.8, 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. Lane colors need to be set before the car starts driving and cannot be changed while the car is driving.

2.2.3 Structure of mPAD V1 Web Application

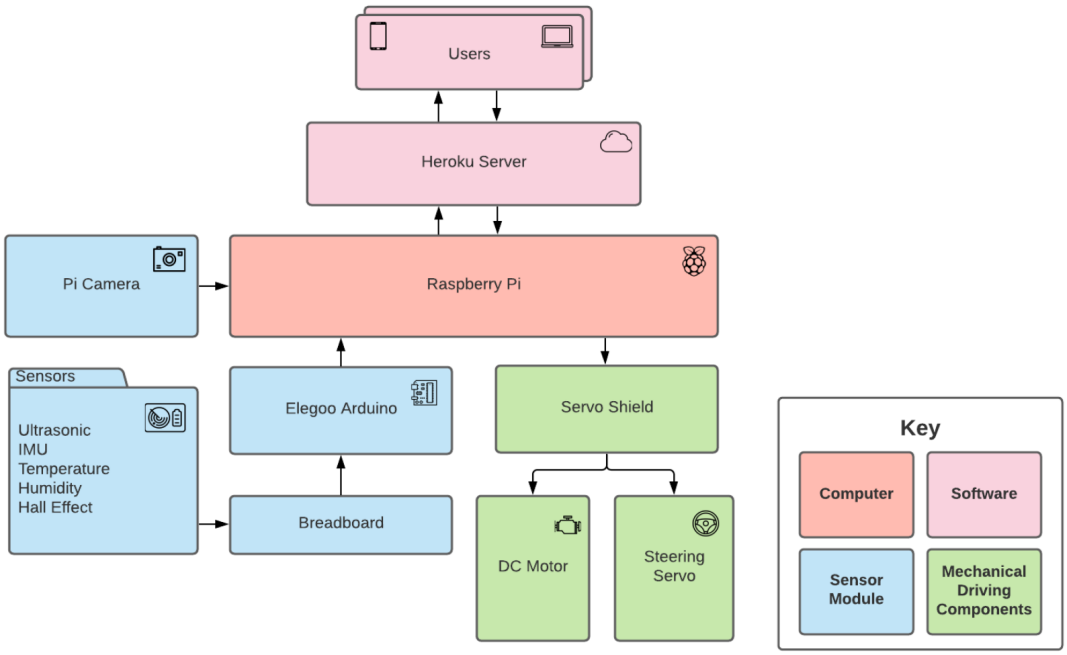

Figure 2.9: Block Diagram of mPAD V1 Layout

(Giglio de Azevedo et al., 2021)

Figure 2.9 showcases how the individual components of mPAD V1 communicated with one another across the system. The key on the right side shows what the different color groupings in the diagram represent. The sensor data and camera feed are processed on the Arduino and sent to the Raspberry Pi, which then sends that information to the Users (in pink) on the web dashboard through the Heroku Server. A Python script bundles all of the data into a JSON format. HTTP POST Requests currently do not have any security or passwords required.

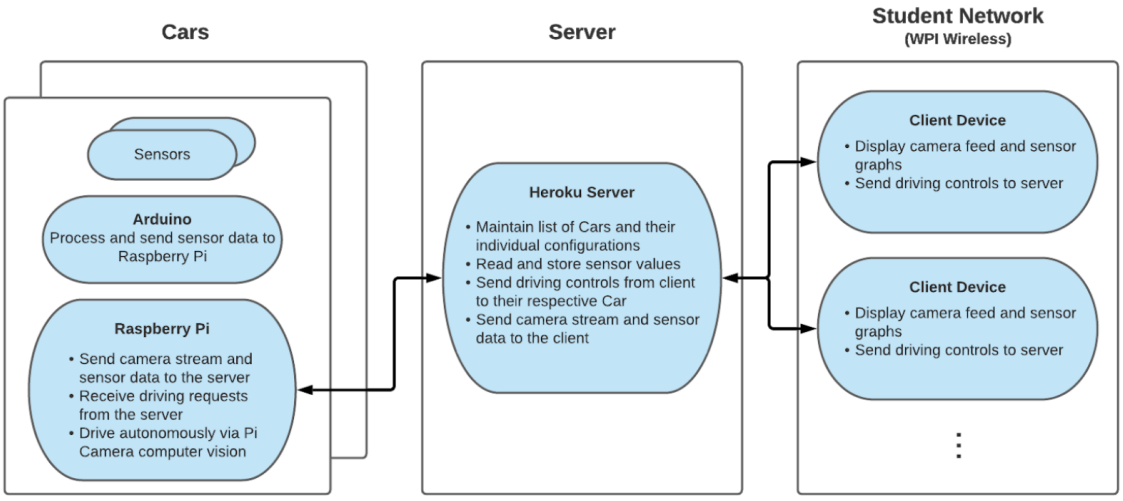

Figure 2.10: mPAD V1 Web Application System Architecture
(Giglio de Azevedo et al., 2021)

Figure 2.10 has a brief description of what occurs throughout mPAD V1's web application. The dashboard application is currently limited by the server's request handling capabilities. Heroku is used as a server and all communication and data go through it, but one of the major drawbacks is with the free version's performance. Redis, a Heroku add-on, is used for storage, it is an in-memory key-value data store (similar to a cache) that is much faster than an external database alternative.

The startup scripts for the Raspberry Pis were written in Bash and all of them are already pre-configured for WPI's Network. Two scripts are executed on startup, the sensor-reading script and the master startup script for the car's controls. The sensor reading script 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. The master startup script establishes 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. By opening both the serial feed on the Pi and the web dashboard, we can see the flow of data from the Arduino.

Dashboard Controls:

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 are 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 2.3 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 the 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.

2.3 Self-Driving Implementation

2.3.1 Lane Following

The previous MQP team used a variety of different Python libraries in order to facilitate the self-driving aspect of each RC car. Namely, they used the Adafruit_servokit library in order to control the motors present on the RC car, the OpenCV library to capture video input from the camera, and for the processing of said input for the self-driving logic, the Numpy Library (in conjunction with the two previously mentioned libraries) for the self-driving logic itself.

Once the OpenCV library successfully captures the video input from a Raspberry Pi Camera V2 attached to the Raspberry Pi of the car, the frames of the input are processed such that only two small portions of the original input are visible to the program. Thus, the program is able to have a clear view of the lane without any potential noise (e.g. objects colored similarly to the lanes located in irrelevant parts of the frame), thus decreasing the likelihood of getting a false positive in lane detection. Figure 2.11 below shows an example of how the crop forms the view of the system. The lane detection logic itself is relatively simple. Namely, a color filter is applied to the cropped frames which turns pixels with the correct RGB values to turn white, with

everything else turning black. The program then finds the horizontal value of the white pixels, and adjusts their position in code to reflect that of their real world positions. It then determines the position of each lane by finding the mean of the horizontal positions of white pixels in each frame.

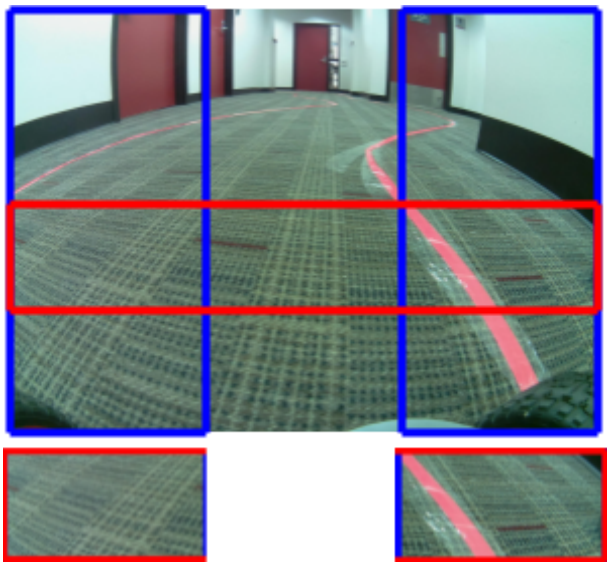

Figure 2.11: Camera view of the Raspberry Pi
(Giglio de Azevedo et al., 2021)

After the lanes are detected, the self-driving code attempts to drive the car while keeping it within the bounds of the lines. It achieves this by equalizing the car's distance to the position of both lanes, which was determined by the lane detection logic. After the distance between the car and both sides of the lane is determined, the program finds the percentage value of the difference between the distances of the car from the left and right sides of the lane. It then uses this percentage difference to set the angle of the wheels, with an upper and lower bound to prevent accidentally damaging the car. It achieves this through using the Python library for the Adafruit Servo Shield that is attached to the car.

2.3.2 Obstacle Avoidance

In addition to self-driving capabilities, the previous year's MQP team also gave the RC cars basic obstacle avoidance. This is achieved through an array of ultrasonic sensors which are

controlled by an Arduino microcontroller, which is in turn connected to the Raspberry Pi 4 used to control the RC car's self-driving capabilities.

In order for obstacle avoidance to work properly, it is important to know what pixels in the Raspberry Pi's camera input correspond with what part of the ultrasonic sensor array. Thus, the RC car will know where to drive with both the lane and any potential obstacles in mind. As such, the previous MQP team built a tool to calibrate the ultrasonic sensors that was present in the source code on cars but for some unexplained reason was left out of the dashboard.

Once the ultrasonic sensors are installed and calibrated, the extra information obtained from them is used in addition to the information provided by the camera in order to enable the car to avoid any detected obstacles. Namely, if the ultrasonics detect an object that is closer to the car than the lane is, it will consider the object as the new lane until it is no longer detected by the ultrasonic sensor array. This logic, in addition to the existing lane-following logic described in Section 2.3.1, will allow the RC car to maneuver around the obstacle.

3. Background

The following section provides a background understanding of the various systems and principles that the team researched to develop this project. It also takes into account the recommendations from Giglio de Azevedo et al. about potential technologies that could be incorporated into the Autonomous Car. The topics discussed in this chapter include Alternative Self-Driving Systems for RC Cars and Sensors for Object Detection.

3.1 Alternative Self-Driving Systems for RC Cars

There are many platforms that can be applied to an RC car to provide autonomous driving capabilities. Some examples of these platforms are DonkeyCar, AWS DeepRacer, and the OSOYOO Robot Car ("Donkeycar: a python self driving library," n.d.; "AWS DeepRacer," 2022; "OSOYOO Robot car kit Lesson 1: Basic Robot car," 2018). While all these kits offer different functionality, one common trait among them is that they were developed with machine learning approaches in mind. Machine learning is robust and potentially makes adding complex behaviors to an RC car less logically complex.

However, a major shortcoming of machine-learning based approaches to an autonomous platform is the length of time it takes to train a car to the point where it is able to reliably drive autonomously. Using DonkeyCar as an example, users must record 10-20 efficient laps around their intended track before training a model for upwards of 5 hours. Additionally, the model trained is specific to the car and track that the data was recorded on, and as such the entire process must be repeated should the users wish to use a different car or track. As such, the length of time it would take to provide enough data for and train a model capable of supporting a modular system is extremely time prohibitive, making a logic-based computational approach like mPAD more favorable.

The shortcomings of a machine learning model approach to self-driving were made evident during WPI's 2019-2020 academic year. A MQP team utilizing the DonkeyCar system ran into several issues related to their self-driving system that they could not overcome by the end of their project (Kim et al., 2020). Specifically, Kim et al. point out models being unique to their hardware and quality training data being difficult to obtain, among other things, as issues to be addressed in any future work with their project. However, many of the proposed changes -

such as standardizing hardware across cars - would decrease the modularity of the system and thus are undesirable.

3.2 Obstacle Detection

When testing the ultrasonic sensors from the previous year's team, it was found that they only provide measurements from a max distance of 50cm and at a sampling rate of 1Hz. Theoretically speaking, if the car is going any speed higher than 0.5m/s (or 1.12 mph), it will just have enough time to detect an obstacle, but will not have enough time to avoid it. You could compare 1mile-per-hour to a slow walking speed, which could be considered reasonable for a small scaled RC car. However, this hinders the future development of the car on any larger scaled vehicles. Thus, the team searched for a better alternative for obstacle detection.

3.2.1 RPLiDAR

The first alternative that was studied was the Slamtec RPLiDAR, a Light Detection and Ranging (LiDAR) sensor designed for the Raspberry Pi, that can provide a 360° mapping of the surrounding 2D environment (Huang, n.d.). If used in conjunction with an IMU sensor, it could even provide a 3D mapping of the environment in front of the car, allowing the car to see objects or even change in terrain.

Figure 3.1 Slamtec RPLiDAR

reproduced as if from link

The RPLiDAR, shown in Figure 3.1, has an average sampling rate of 5.5 full 360° scans per second and a max range of 12m, which is five times faster and 24 times longer-range than our current ultrasonic setup. In theory, this should mean that the RPLiDAR is capable of

detecting obstacles up to speeds of 66m/s (147mph). The module is split up into two systems, the laser and the motor. The laser requires a standard power supply of +5V DC and 100mA, the motor +5V DC, 500mA startup current and 300mA working current. Additionally, the module uses Universal Asynchronous Receiver-Transmission (UART) to communicate with the Raspberry Pi. Compared to the PixyCam and HuskyLens, this component is the largest in size and probably the most difficult to implement mechanically, without getting in the way of other components. Due to the nature of its measurement recording function, the team speculates that it must be placed on top of the car, and will most likely need to be tilted at an angle to get a better reading of the surface in front of the car.

Initial research and testing was done to find the feasibility of using the RPLIDAR in mPAD. The RPLiDAR is a 2D LiDAR that can detect walls and obstacles on a single plane. It collects distance in mm, angle in degrees, it checks whether a point belongs to a new scan, and the quality of the current measurement. When it comes to obstacle avoidance, the advantage it has over the ultrasonic sensors is that it can detect obstacles in a much wider range and can help avoid collisions with walls much more effectively. It can also detect objects as small as 1mm (specification says .05mm but during testing this did not prove to be the case). It is not sufficient for low hanging obstacles or short obstacles on the ground since it can only detect in a 2D plane - as shown in Figure 3.2. It makes 1400 measurements per second so in total it will take about 700 microseconds to process the data. The Arduino may be a bit slow to process this amount of data each loop. We concluded that the RPLIDAR system would be an improvement to the current ultrasonic system in place. However, we did not pursue the completion of the object detection implementation using the LiDAR in the interest in completing high priority tasks.

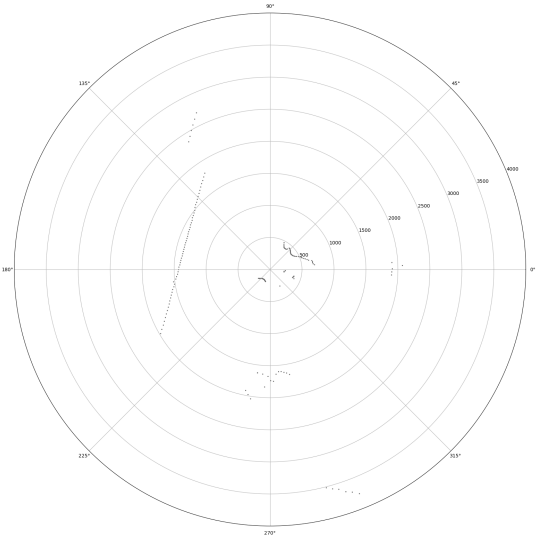

Figure 3.2: RPLiDAR displaying data on a 2D plane

3.2.2 PixyCam

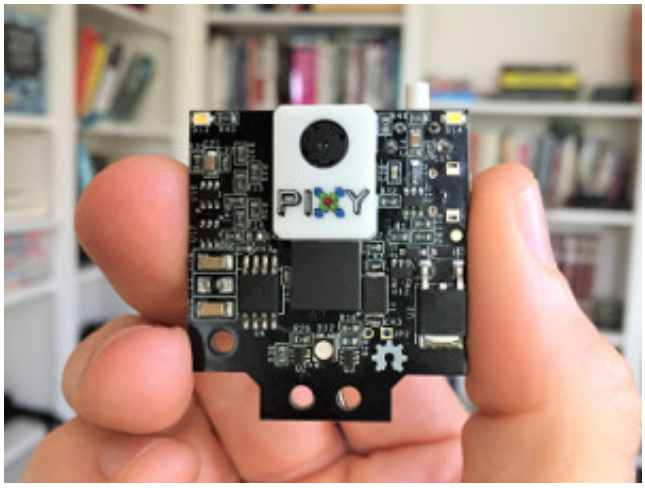

Figure 3.3: PixyCam2

reproduced as if from link

Another alternative that was considered was the PixyCam2. This module is a vision sensor with an integrated chip to process color and line data received from the lens. This allows the sensor to detect objects, track lines, and read bar codes, based on specified color values (PIXY2, n.d.). These are all functions that would improve the functionality of mPAD. The

PixyCam2 can be used with the Raspberry Pi or Arduino, and can communicate via Arduino ISPC, SPI SS, I2C, UART, analogX, analogY, and LEGO SPI. The integrated processor helps the module to send only the data that the microcontroller needs in order to minimize computations and allow the microcontroller to use processing power for other tasks. The PixyCam2 can sample at a rate of 60fps, which is much faster than both the ultrasonic sensor system and the RPLiDAR. It can be powered with +5V by USB or 6-12V from an unregulated source, and usually consumes around 140mA. However, it cannot pass video footage to the microcontroller, and will need to be used in conjunction with a USB camera for the mPAD system. The PixyCam2 can be purchased on Amazon for $59.99, which is more appropriate for our budget. Overall, the PixyCam2 is a promising option that we may consider using to improve the obstacle avoidance function of our vehicle. An example of the hardware of the PixyCam2 can be seen in Figure 3.3.

3.2.3 HuskyLens

Figure 3.4: HUSKYLENS

reproduced as if from link

The team also did research on the Gravity: HUSKYLENS AI Machine Vision sensor. This is another vision sensor module with a much more powerful integrated chip in comparison to the PixyCam2. It includes more functions to detect all different things such as color recognition, object tracking, object recognition, line tracking, tag recognition, and face recognition. The module includes a 2-inch screen to allow the user to configure the sensor without needing a computer. It can be used with many different microcontrollers including the Raspberry Pi and Arduino. The power requirements of this device is standard at +3.3V, 320mA

or +5V, 230mA. It communicates with the microcontroller via UART or I2C. Compared to all the other alternatives for obstacle avoidance, this option is the cheapest at $44.90. Just like the PixyCam2,, it cannot pass video footage to the microcontroller, and will need to be used in conjunction with a USB camera for the mPAD system. The hardware of the HUSKYLENS system is shown in Figure 3.4.

4. Initial Package Review and Analysis

During A-term we conducted a review and analysis of the work of the previous MQPs. Using cars from coursework we analyzed the electronic hardware and sensors, self-driving code, and general system design of the previous mPAD teams. The purpose of this section is to discuss some of the issues we came across during testing and our solutions to these issues.

4.1 Network Connectivity Issues

The project by Giglio de Azevedo et al. used a cloud platform called Heroku as a server to host the dashboard. This platform enabled communication and transmission of data between the dashboard and the various components on the car. Thus, the Raspberry Pi required a stable, continuous connection to WPI's wireless network to interact with the Heroku server and successfully perform autonomous driving.

However, this implementation's reliance on the internet posed several issues. Firstly, the WPI network can have "dead spots" around the campus which are zones where the Raspberry Pi would not be able to connect to the WiFi. Thus, the car would be unable to drive autonomously as the bridge for communication between the various parts would be interrupted. Secondly, there were issues with initial remote connection to the Raspberry Pi on startup. This is because remote connection to the Pi uses the SSH protocol and the Pi's IP address. However, the Raspberry Pi's IP address would change frequently. Thus, the user would have to connect the Raspberry Pi physically to a monitor and keyboard, run a command to find the IP address of the given Raspberry Pi, and then connect remotely using this IP address. This process is very inconvenient especially for novice users with limited experience using Linux's terminal. Thirdly, with Heroku hosting the dashboard, all Raspberry Pis running the code would be connected to the same server. Thus, the user may unintentionally connect to the dashboard controlling a Raspberry Pi other than the one they intended. Finally, the internet connection needs to be powerful and consistent enough to facilitate continuous data transfer between the components.

During testing, our team found the performance of the system was negatively affected by the inconsistency of the Heroku platform. Thus, the team evaluated the possibility of using local servers as an alternative to Heroku. The team discovered that the Raspberry Pis themselves were capable of launching their own servers. We were then faced with two potential solutions. Each car could be equipped with two Raspberry Pis, one which would host the server and another that

would be responsible for processing the self-driving code. The other solution would be using a single Raspberry Pi to host the server and dashboard as well as to run the self-driving code. We ultimately chose the latter to reduce the physical load on the driving system of the RC cars. We were able to launch a local server from the Raspberry Pis by running the dashboard code and hosting it on the Pi's IP address or hostname. This was accomplished through the use of a script that runs on the Pi's startup. It executes a series of command line instructions to acquire the IP address, then host the dashboard at that address. This local implementation also has the added benefit of the Pi being able to continue running the self-driving code even if internet connection is interrupted.

We then resolved the connection issues posed by the dynamic IP addresses of the Pis. We discovered that although the IP addresses tend to change on startup, the Pis' autoreg hostnames are static. Thus, we made note of all the Pis' hostnames and used them to connect remotely via SSH rather than using the IP addresses. Also, since each Pi is now hosted independently from the others using its unique IP address, this resolved the issue of the user potentially connecting to an unintended Pi. Using these local servers also reduced overall network traffic of the system compared to when the Heroku platform was being used for all cars at the same time. This improved the efficiency of communication in the parts of the system that used the network.

4.2 Camera Issues

At the beginning of the academic year, the system was extremely inconsistent and often would not start up as intended. As such, we monitored the terminal output of the system - a process that was time consuming and difficult to discover how to do due to the lack of documentation provided to the team - for any error messages that would give the team hints as to what the problem may be. The terminal quickly revealed an error message stating that OpenCV, the vision library used by the system, could not detect a camera connected to the system - despite the fact that the camera was clearly connected to the Raspberry Pi. At first we thought this was an issue related to faulty hardware, so we tested the system with components that we confirmed were working, but we encountered the same problem. The team then decided to isolate the issue and disconnect everything else from the Raspberry Pi to see if something else was interfering with the connection. Starting the system after removing all other components resulted in the camera being detected. However, as the system requires other components to be connected to the

Raspberry Pi - such as the servo shield and arduino - to achieve full functionality, this did not provide a solution. Reconnecting the servo shield and arduino after only plugging in the camera module yielded limited success, as sometimes all the components would be detected and the system would run without issue, and other times a component would not be found and the system would fail to start. More details on the team's efforts to troubleshoot and fix the issues related to the Raspberry Pi camera are discussed in Section 8.1.

Due to the problems being caused by the proprietary Raspberry Pi camera and the fact that we wished to make mPAD more modular, we decided that it was more effective to switch to using USB webcams. Doing so completely eliminated all consistency issues we were encountering due to the Raspberry Pi camera, and also made the system capable of using any USB webcam to self drive. More details on the process of implementing the USB webcam can be seen in Section 8.2.

4.3 Sensor Power Issues

While testing the car on the Heroku server, the team was having difficulty getting sensor data displayed on the dashboard. This was hindered by the camera issues mentioned before, however the team still decided to take a deeper look into the sensor package circuitry. The first step was to run test code on the Arduino IDE to see if the sensor package was outputting data to the Raspberry Pi correctly. Upon run, it was found that data was being output, however no data was being read from any sensors, all reading blank results. The team next tested each of the different sensors individually to make sure that there is no weak link in the chain. It was found here that all sensors were working properly by themselves, which means there is a problem when they are all connected together. In further investigation, it was found that the Arduino Mega +5V port is sometimes not able to output enough current to the sensor package. All of the sensors, with only one temperature sensor included, need at least 530mA, while the Arduino is only capable of putting out 500mA. Additionally, the Arduino Mega is known for having poor power output when powered by USB. The team was able to solve this by avoiding the use of the Arduino port, and instead attaching an Elegoo power module, which is powered by a 9V battery, and supplies the desired +5V and 700mA. This implementation fixes the power issue, but also adds an additional battery and weight to the vehicle. This issue was later resolved, and the power module was not needed for reasons that could not be explained by the team.

4.4 Temperature Sensor Issues

One issue that was noticed with the temperature sensors is that they do not display independent temperature measurements. When testing the DHT11 components, it was found that the temperature is only able to be read from one sensor at a time. When running the code from the previous year's team, an increase in temperature recorded by one sensor, would also be read by another sensor. It was determined that the temperature sensors only generate a collective temperature value rather than independent values that are associated with a certain area of the car. If we did want our system to record different temperatures at different parts of the car, then we will certainly have to look into upgrading the DHT11 temperature sensors. There is another sensor called the DHT22, which is a more powerful and accurate version of the DHT11, however it has a smaller sampling rate and is a little more expensive. This is another issue to be solved throughout the next term.

4.5 Sensor Dashboard Display Issues

Another issue related to the sensor package was that data from the attached sensors was not being displayed on the dashboard, despite the fact that the page had charts and graphs - as well as logic - clearly intended for showing sensor readings. However, even though we had discovered the problem, we did not yet know why it was occurring. As such, we needed to investigate where in the software package the problem originated from. There were three potential points of failure for this particular issue, which are as follows:

1. The self-driving script is failing to send sensor information to the dashboard
2. The server is failing to correctly receive and store the sensor information sent to it
3. The logic for the dashboard is failing to correctly receive or display the sensor data once it is obtained by the server.

To investigate the first potential point of failure, we simply added some debug statements to the server logic that would print a debug message and the sensor string to the console upon receipt of a sensor string. We then ran the self-driving script to see if the server console printed a message, which indicates it is receiving a message from the script. The console did not print anything and as such we knew that the self-driving script was failing to send the sensor information to the dashboard.

We then examined the second potential point of failure by utilizing Raspbian's 'curl' command and sending a sample sensor string to the server. Sending a sample sensor string that we knew was valid to the server from the terminal would show us whether it was correctly handling the string upon receipt. As a result, the debug message added during the investigation of the first point as well as the valid sensor string was printed to the console, but unfortunately the information in the string failed to display on the dashboard, which showed that there was in fact a problem related to the dashboard correctly receiving or displaying the sensor data.

The first problem was easily solved by modifying the driving logic to include code to send a POST request containing the sensor string to the server. However, the second problem was a bit tricker to solve. After a significant period of time testing and reviewing the logic for displaying the sensor data on the dashboard, we realized that the problem did not stem from an issue with the code. For some reason, even though the logic for both sending the data from the server to the dashboard and displaying said data were correct, the dashboard was not correctly receiving the sensor string from the server. Eventually, we noticed that whenever we would stop the driving logic on the Raspberry Pi, the application would not actually stop. This discovery led us to realize that the system was being started twice by the Raspberry Pi on boot- once via cron and the other using Raspbian's included systemctl feature. The two instances of the system would interfere with each other and caused its communication with the server to be inconsistent, which in turn caused the sensor data to fail to display. Disabling the cron job and relying solely on systemctl fixed the issue, thus enabling the system to correctly display sensor data on the dashboard without any problems.

4.6 Car Hardware Issues

During preliminary testing of mPAD we used RC cars built by students in an ME course here at WPI as seen in Figure 4.1. Initially, testing went seamlessly as we received the vehicles in good condition already set up to run and test the system. However, quickly we found that the vehicles were not exactly built to hold up. Failure mostly began with 3D printed components after extended use. Parts on various vehicles were either completely stripping or shearing as seen in Figure 4.2. Early on in this project, progress was hindered by these mechanical troubles as the team had to wait until repairs were completed before changes could be tested. The solution to

this problem was to purchase commercial, off-the-shelf RC cars that we knew would hold up much better under consistent wear and tear. These vehicles are shown in Figure 4.3

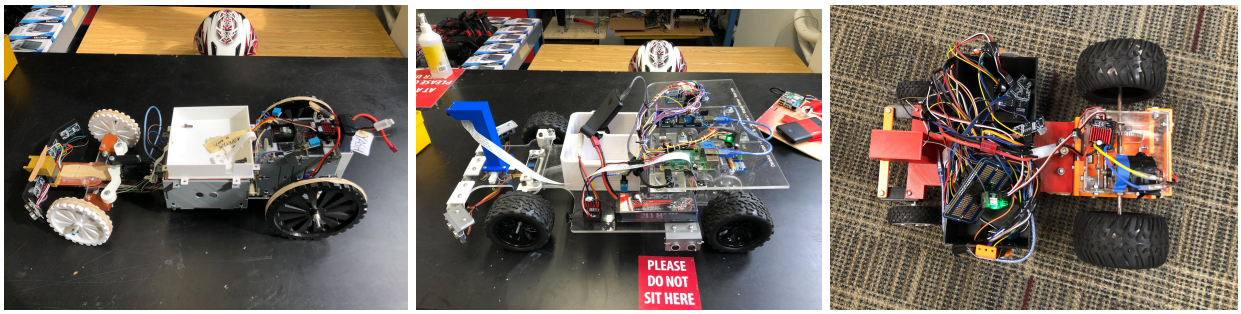

Figure 4.1: Vehicles used for early testing of mPAD V2

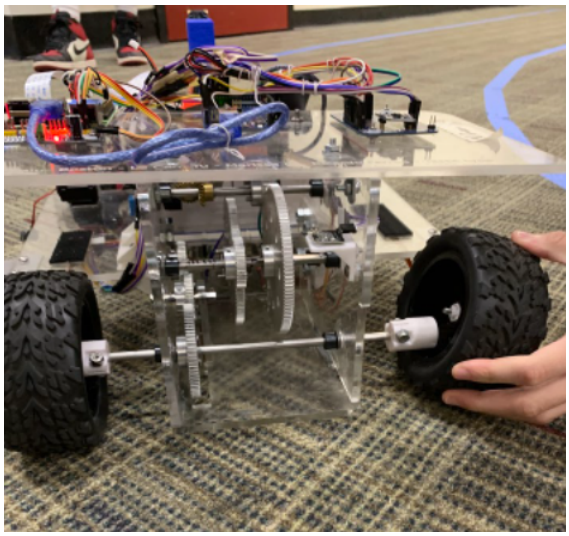

Figure 4.2: Damages to Rear Wheel Hub

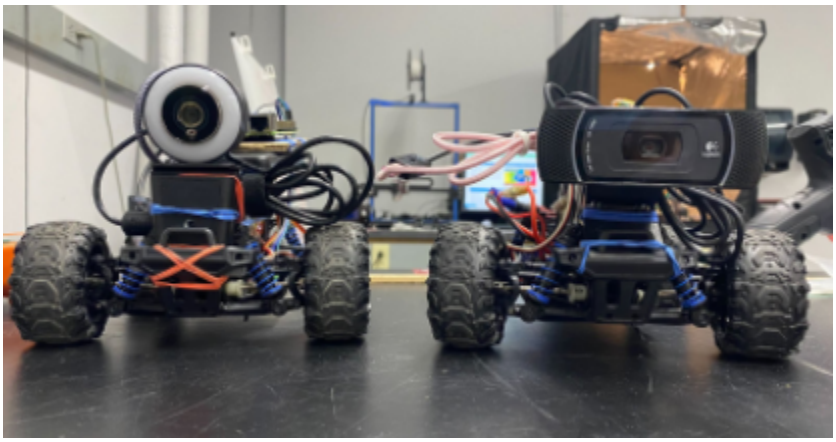

Figure 4.3: Vehicles used for later testing of mPAD V2

To hook up mPAD V2 to these new vehicles, we removed the plastic shell on top of the RC cars and decided to add a top plate that could be used to place the hardware on. We did this by laser cutting ¾” birch wood. We had 4 separate vehicles, but 2 of them were the same model. With this in mind we designed three top plates. They all had the same simple design. Thin in the front to prevent the wheels from hitting the plate while steering, and the same width past the wheels to the rear of the plate. One of the CAD models is shown in Figure 4.4. Additionally, the holes cut into the plate were used to mount the plate onto the RC car the same way the pegs are used to hold the plastic shell in place. These plates made adding electronic hardware to the small vehicles much cleaner and easier. Figure 4.5 and Figure 4.6 show before and after the top plate was added to the RC cars.

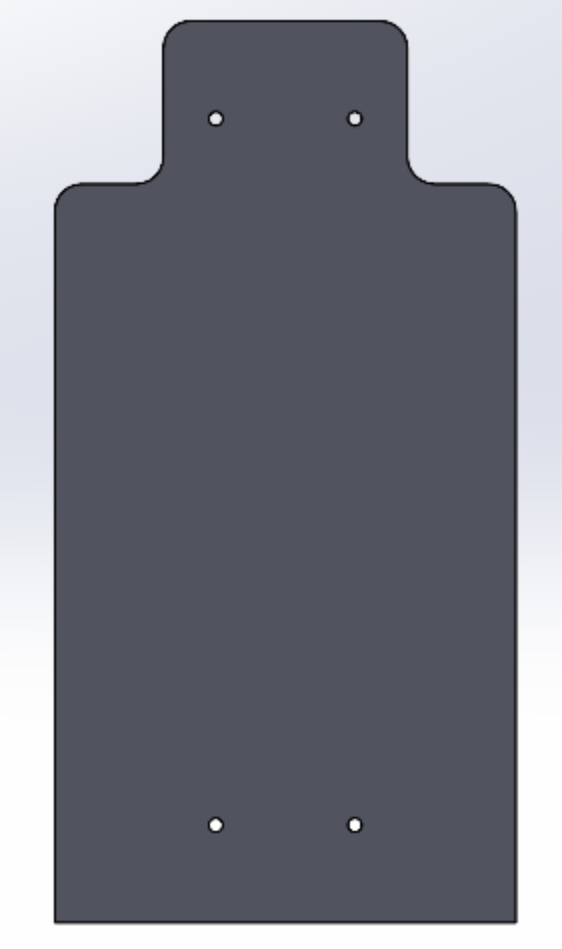

Figure 4.4 CAD Model of Top Plate for RC Cars

Figure 4.5: Before Adding the Top Plate

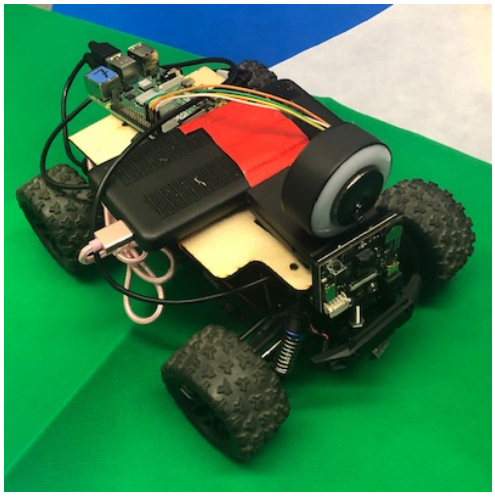

Figure 4.6 After Adding the Top Plate

4.7 Track Issues

The track used for testing mPAD is located in the basement of WPI's Higgins Laboratory, consisting of colored tape that snakes in a loop around the hallways as shown below. A sketch of the track can be found in Figure 4.7.

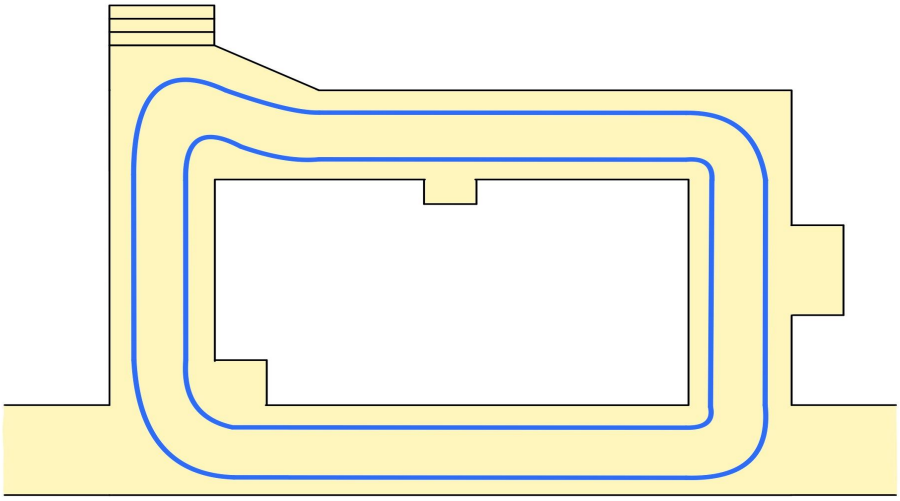

Figure 4.7: mPAD Testing Track in Higgins Laboratory

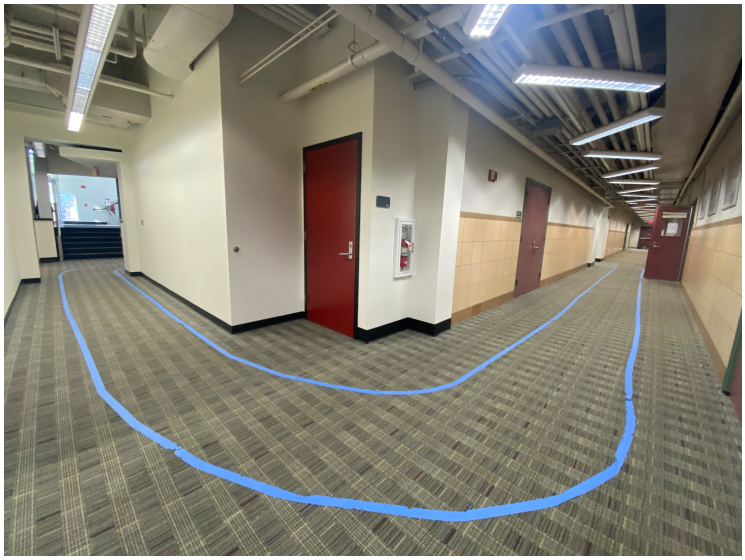

Figure 4.8: mPAD Testing Track in Higgins Laboratory

Throughout the course of the project, the team ran into many issues where the track was not being detected in certain locations. There are some parts of the track where the lighting makes the track brighter or dimmer. As can be seen in Figure 4.8, there is a larger window that emits natural light. Depending on the time of day, this light can greatly impact the brightness of the track and mPADs detectability. There are also moments where this lighting brings out other colors within the environment that are similar to the track and confuses mPAD. To combat this, the team laid down a new set of blue tape to revive the old faded blue, and also added another yellow tape track next to the blue, which can be used when the blue is not the most optimal choice. At some points the blue works better, and at others the yellow works better. An example of the two-colored track can be seen in Figure 4.9 below.

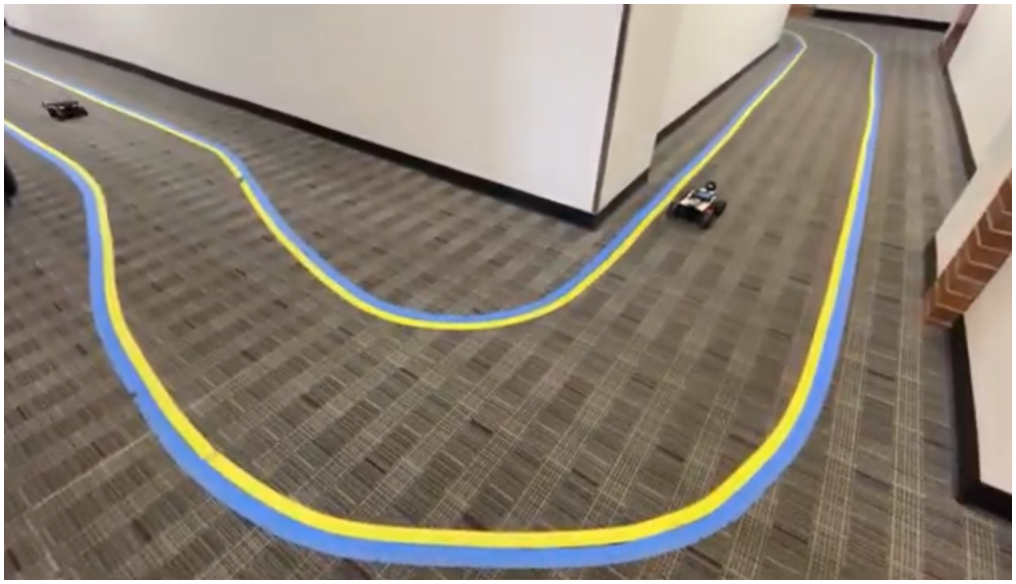

Figure 4.9: mPAD Testing Track with Blue and Yellow Tape

5. Initial Software Analysis and Development

The Raspberry Pi serves as the center for communication between the key components in mPAD V2. The device stores the code needed to run our program and handles the processing that allows the software to interact with the hardware of the system. As we were building off the work done by Giglio de Azevedo et al., the team naturally had to first perform a complete analysis of the mPAD V1 software, then make improvements based on what was discovered.

5.1 Software Analysis of mPAD V1

Upon first inspection of mPAD V1, the team faced several challenges trying to get the software running. The documentation from Modular Package for Autonomous Driving (MPAD) (Giglio de Azevedo et al, 2021) states that the mPAD software should startup automatically once the Pi is booted up. Once the Raspberry Pi was turned on, the self-driving program would run automatically dashboard features such as the cameras and sensors were not working correctly. We could not fix these problems as we were not able to identify where the error logs were or where exactly the self-driving program was being run on the Raspberry Pi. To overcome this challenge, we had to figure out how to disable the automatic startup of the self-driving program and debug the error messages sequentially. We researched the various steps taken to start processes automatically on the Raspberry Pi operating system (Raspbian) and discovered that the *systemctl* command can be used to manage the background processes and tasks running on the Pi. Using this command feature, we were able to determine that the process was being run in the background and disabled it using the *systemctl disable* command.

Once we disabled the automatic startup of the self-driving program, we were able to manually activate the program and view the error messages displayed on the console. The errors were being caused by the camera, temperature sensor and power going to the sensors. There were also issues with getting the sensor data to display on the system's dashboard. These issues and solutions were addressed in detail throughout sections 4.1 to 4.5 of this report.

After resolving the initial issues with the mPAD V1 software, the team wanted to make sure we knew how to set up the system on any new cars we had to test. Thus, we had to figure out how to do a complete setup of the software suite on a new Raspberry Pi. We started by installing a fresh copy of Raspbian onto a Raspberry Pi. We did this by connecting a Secure Digital (SD) card to a personal laptop, then using the Raspberry Pi Imager tool to format the card

and installing the 32-bit version of Raspbian onto it ("Introducing Raspberry Pi Imager, Our New Imaging Utility," 2020). Next, we had to connect the newly formatted Pi to the WPI Wireless network. This proved to be challenging as we initially tried using the native Raspberry Pi network software to connect, but were unable to do so due to the fact that WPI's wireless network requires a valid network certificate for any device that wishes to connect to it. By searching through WPI's technical support pages, we discovered that we needed to install a separate "network manager" software onto the Pi, then follow detailed instructions laid out by WPI to connect the device ("Connect to Wpi Wireless Using a Raspberry Pi with Gui", 2021).

Now that the Raspberry Pi had a fresh operating system and was connected to the internet, our next task was to test the self-driving program. We were able to download the mPAD V1 codebase from GitHub but encountered new issues when we tried to run it. We received error messages indicating that several parts of the code refused to execute due to missing dependencies. We had to use the *sudo apt install* command to download and install each dependency that was missing from the system. We took note of these dependencies and compiled them into a script that would download and install them automatically for the user. We were now able to run the code completely but ran into even more errors. We got messages saying that some of the hardware components were not connected, even though all the wiring was correct. After researching this problem, we noticed that the relevant interfaces were not enabled on the Pi, meaning the hardware components were unable to communicate with the software. We then used the *sudo raspi-config* command to bring up a menu containing configuration options for the Raspberry P, as shown in Figure 5.1 ("Enable I2c Interface on the Raspberry Pi," 2014). To enable the relevant interfaces, we navigated to the *Interfacing Options* menu and enabled the I2C Bus and the camera, as shown in Figure 5.2 ("Enable I2c Interface on the Raspberry Pi," 2014). The camera had to be enabled since it is used for lane detection in autonomous driving. The I2C Bus was enabled to connect to the servo shield on the car so that the Raspberry Pi could send instructions controlling its throttle and steering. We also enabled the SSH option to be able to connect remotely to the Raspberry Pi from our personal laptops to make software development and debugging easier.

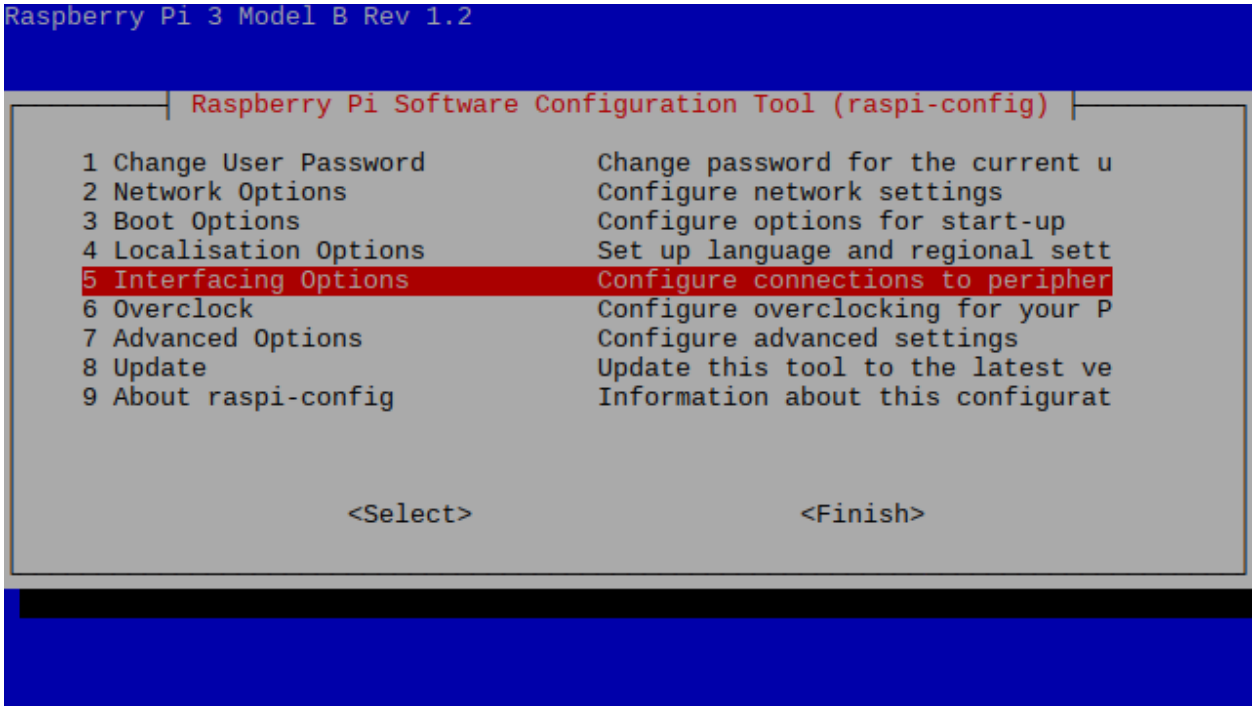

Figure 5.1: Raspberry Pi Software Configuration Tool

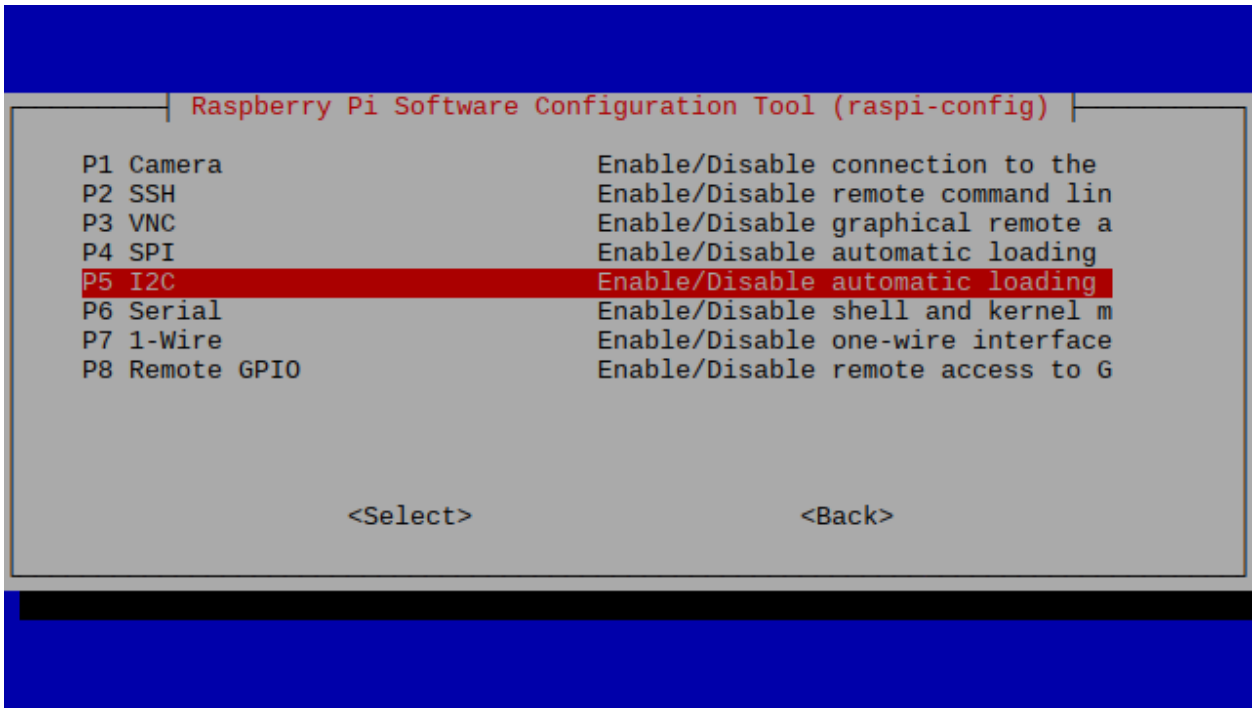

Figure 5.2: Raspberry Pi Software Configuration Tool Interfacing Options

Once these underlying issues were resolved, the system was finally able to run the autonomous driving program without any software errors related to missing dependencies. Next, the team had to learn to start the program in the background automatically again. Since we had previously learned about the *systemctl* command utility, we were able to create a new system service that would run the mPAD startup script once the Raspberry Pi is booted up.

5.2 Developing a Software Setup Script for mPAD V2

Through the comprehensive analysis of the mPAD V1 software, the team was able to gain an understanding of the software development lifecycle of this project. In line with our objective of developing an intuitive package, our next main goal was to ensure that the setup and activation process of mPAD V2 would be as simple as possible. To achieve this goal, we used the startup script from Giglio de Azevedo et al. as a basis to create a new startup script that could automate most steps of the process we went through in section 5.1

The first task the new startup script should be able to perform is to install all of the required dependencies discovered during the analysis phase above. The directory storing code for the dashboard and server is separate from the directory containing code for the self-driving program, so they relied on different dependencies. Thus, a script was created for each directory to execute *sudo apt install* and *pip3 install* commands that install the necessary packages and python libraries. The main startup script was then made to run those two new scripts in order to perform a full installation of all directories.

The second objective of the startup script was for it to set up the system correctly, regardless of where the codebase is stored on the Raspberry Pi. In its current state, the code runs correctly no matter where it is stored, when it is run manually. However, the startup script needs to be able to determine where the code is stored in the filesystem in order for it to start the system properly. To implement this feature, the startup script uses the *pwd* command, which returns the full path of the current working directory. Then, the *sed* command is used to set the location of the startup script to the current working directory. Next, the *systemctl* command is used to enable the service that runs the startup script once the Raspberry Pi is turned on.

Finally, to prove the modularity and consistency of our startup script, we reset three Raspberry Pis and performed a complete setup. As intended, the script started automatically once the Pi turned on and installed all required dependencies. Then, it took about two minutes to fully

launch the dashboard and connect to the car. The script worked seamlessly on all three Pis, with each car having the full functionality of the mPAD V2 package without showing any error messages.

6. Dashboard Changes

There were a number of new features added to the front-end dashboard of mPAD V2 to further improve the useability of our system. Our goal was to make it easier for users to utilize mPAD with any RC car without having to make changes in the code. Some of the new features include servo angle adjustment, servo/motor channel selection, a comprehensive tutorial walkthrough, manual driving, hardware component test scripts, inverse steering and throttle switches, LiPo battery level indicator, and a number of UI/UX updates. Figure 6.1 shows mPAD V1's dashboard beside Figure 6.2 which showcases the new and improved mPAD V2 dashboard. In this section, we will discuss how these features were implemented in further detail and the reasoning behind why we believed these features were necessary.

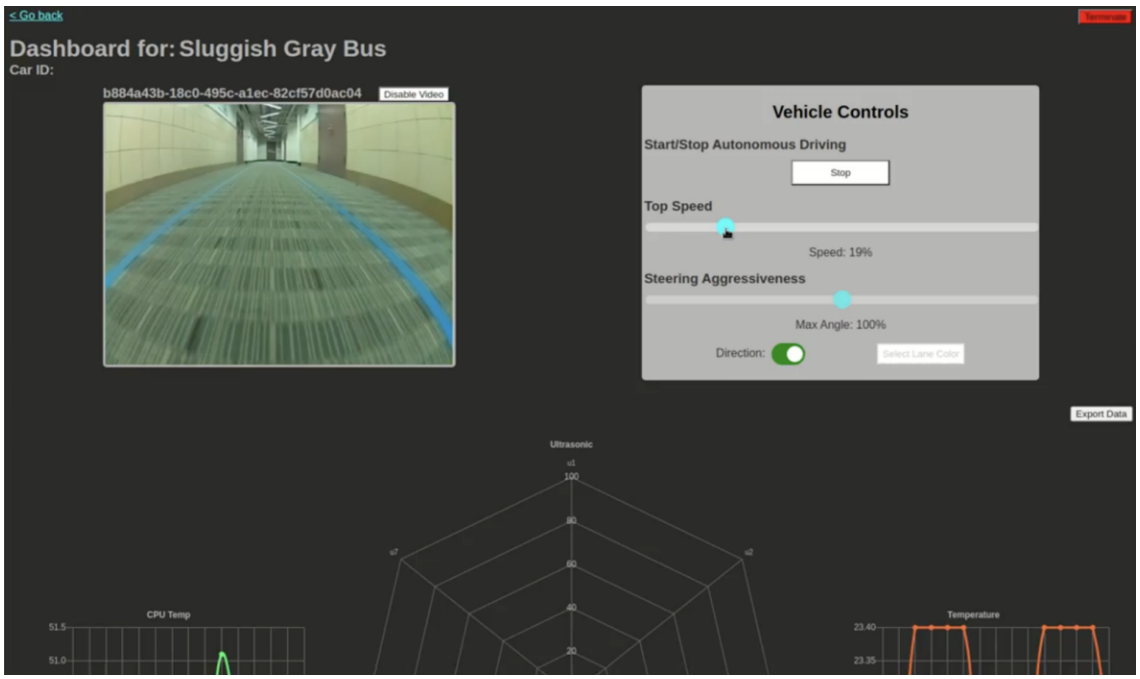

Figure 6.1: Front-End Dashboard of mPAD V1

(Giglio de Azevedo et al., 2021)

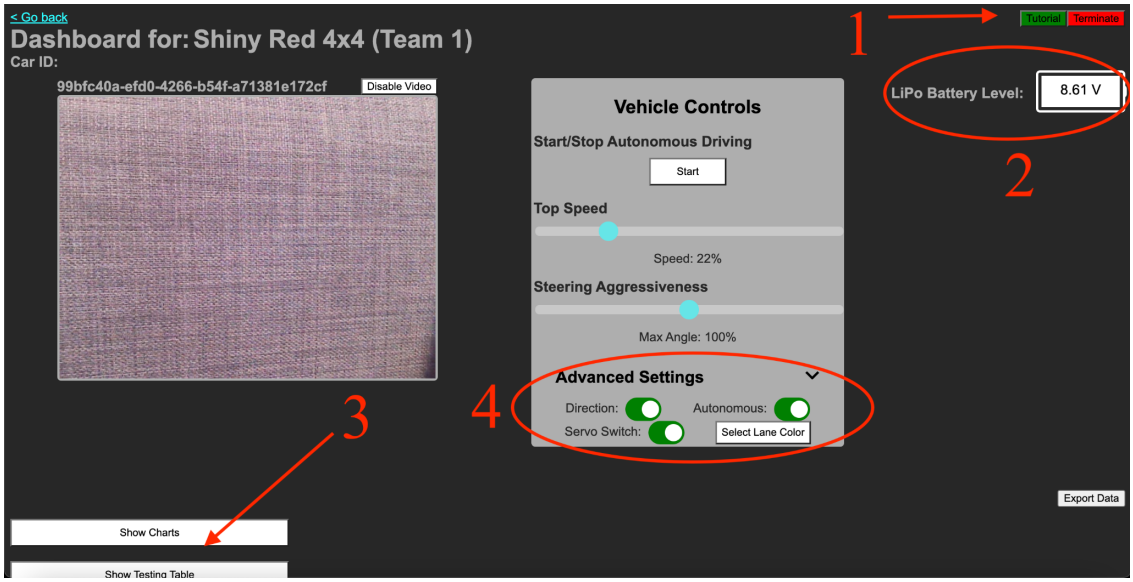

Figure 6.2: Front-End Dashboard of mPAD V2 with New Features (1: Tutorial Walkthrough Button, 2: LiPo Battery Level, 3: Hide/Show Buttons, 4: Advanced Settings Dropdown and Invert Switches)

6.1 Technologies

The primary integrated development environments (IDE) that were used to build the dashboard were JetBrains' WebStorm and PyCharm IDEs. WebStorm was utilized to build the JavaScript, HTML, and CSS files in the project while PyCharm handled all of our Python files. Our development team utilized GitHub and would develop on different feature branches to ensure our feature was working as desired prior to merging into our main code. Our team would also use Glitch to preview some of these features. Glitch is an online IDE that supports HTML, JavaScript, and CSS. This IDE is integrated with Git, so our changes would auto-deploy and we could easily preview them. Glitch made it easy to share previews of features with our advisors as they have shareable links where developers can share their projects with others.

6.2 Servo Shield Adjustment

An issue the team noticed while testing mPAD was that the score was not capable of driving in a straight direction without veering off at a slight angle. Our team noticed that this was due to the fact that the 90° servo angle we had set up in the software did not always correlate

with the wheels facing perfectly forward in the hardware of the servo. Figures 6.3 and 6.4 demonstrate the feature.

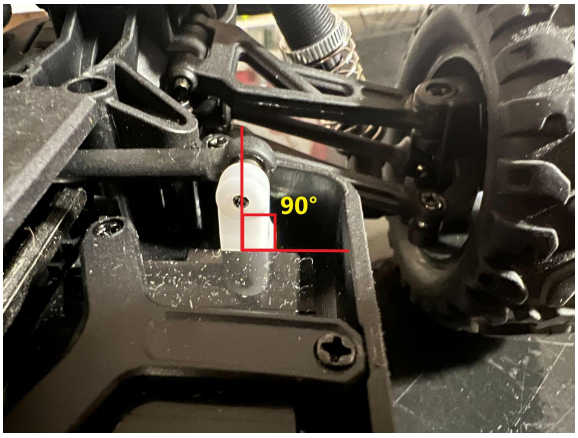

Figure 6.3: Servo at 90° (Wheels Straight)

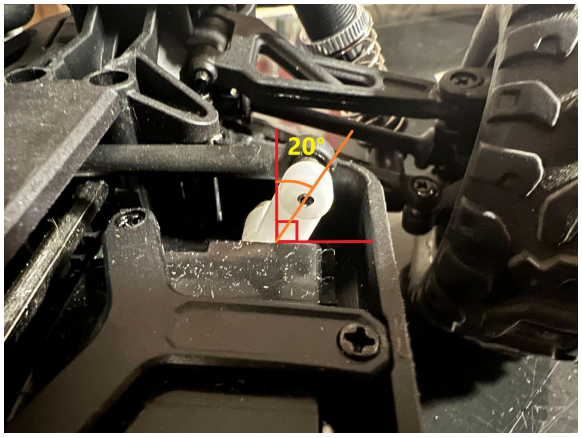

Figure 6.4: Servo offset at 110° (Wheels Turned 20° to the Right)

As such, an offset tool was developed to combat this problem. This tool can be accessed by opening up the *Advanced Settings* dropdown circled in Figure 6.2. Figure 6.5 shows what the *Servo Shield Adjustment* slider looks like. This new feature allows users to use a wider variety of servos with mPAD which furthers our goal of modularity.

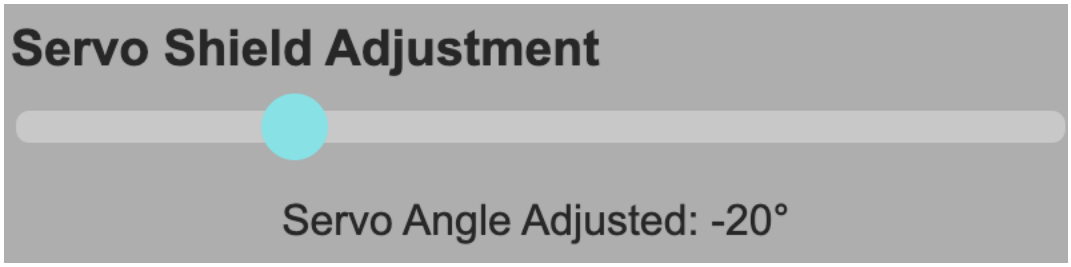

Figure 6.5: Servo Shield Adjustment Slider

The slider has a minimum of -40° and a maximum of 40°. When the slider is set, a HTTP POST request is sent and a variable named servo_adjustment in drive.py's initial_configs JSON is updated and added to the servo steering angle value. The vehicle's steering angle is then adjusted by the value the user sets on the *Speed Slider*. This feature was tested with cars with offset servos. Originally, driving straight with these vehicles while utilizing mPAD would cause

them to veer off, but after using our new feature and adding a couple of degrees of adjustment, the car was able to drive in a perfectly straight direction.

6.3 Servo/Motor Channel Selection

This feature furthers mPAD's modularity by allowing users to plug the servo and servo controller into any of the 16 channels on the servo shield. The servo controls the vehicle's steering while the servo controller is connected to the vehicle's motor and controls its throttle, as seen in Figures 6.6 and 6.7. When the user hovers over the dropdown button (shown in Figure 6.8), options of Channels 0 through 15 pop up and the user is free to select any channel. By default, the servo motor's channel is set to Channel 0 and the ESC Motor Channel is set to Channel 1. The dashboard does not allow the user to select the same channel for both dropdowns.

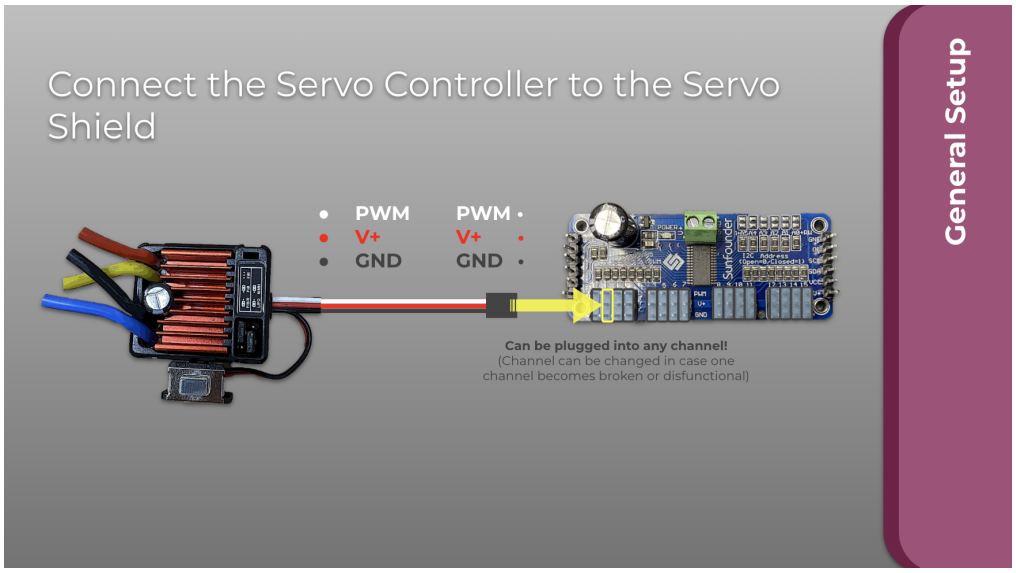

Figure 6.6: Hardware Setup of the Electronic Speed Controller

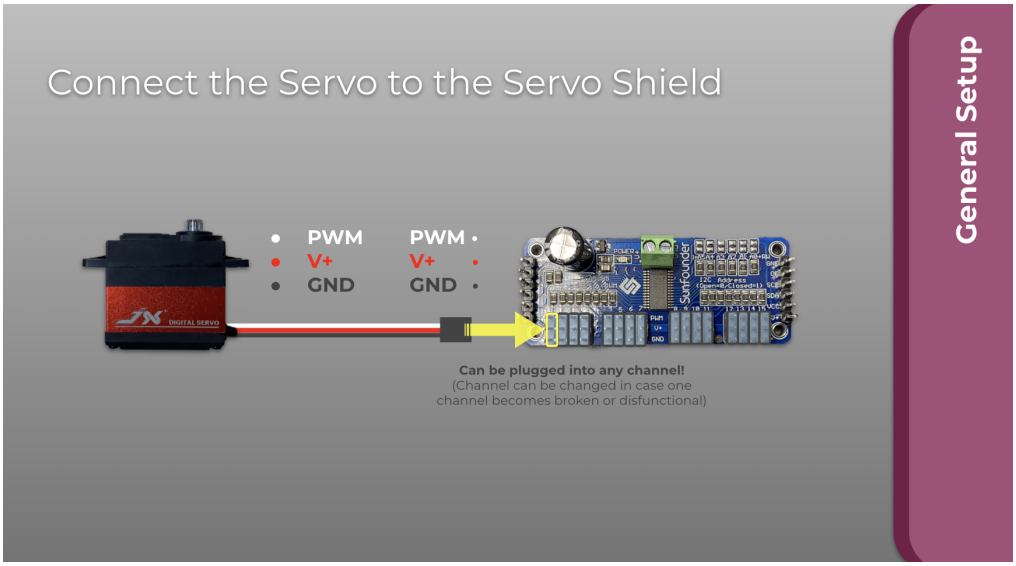

Figure 6.7: Hardware Setup of the Servo

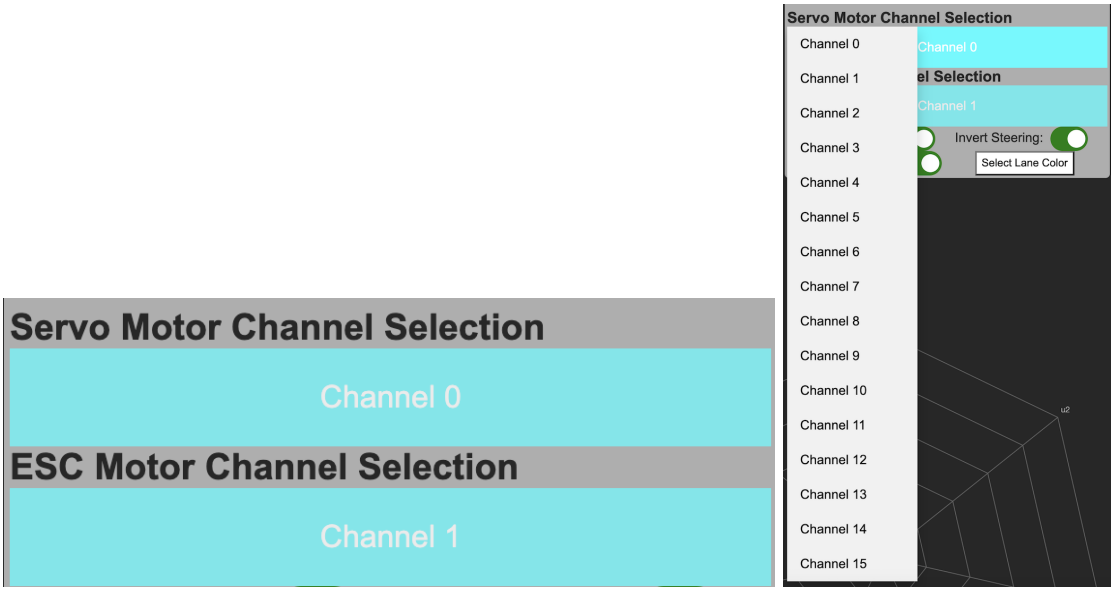

Figure 6.8: Servo & Motor Channel Selection Dropdowns

When a new channel is selected on the dashboard, the text on the button is parsed utilizing substring() to only get the number showcased on the button. This value is then sent through an HTTP POST request and updates the JSON values for servo_channel and esc_channel. This feature was tested on a number of cars. We tested it by plugging the servo and servo controller into different channels on the servo shield and testing to see what the effect was

on the car. The steering and throttle would only work if they were plugged into the same channels that are selected on the dashboard.

6.4 Comprehensive Tutorial Walkthrough

In order for us to simplify mPAD's front-end dashboard and make it easier for users to learn the controls, our team added a comprehensive 18 step tutorial walkthrough for every button and function on the dashboard. For the user to start the tutorial walkthrough, they must click the green "Tutorial" button in the top right corner of the webpage. To accomplish this, we utilized Intro.js, a JavaScript library for creating step-by-step and powerful website tours (Mehrabani). This made it very easy to implement a clean and modern looking overlay for each button's description.

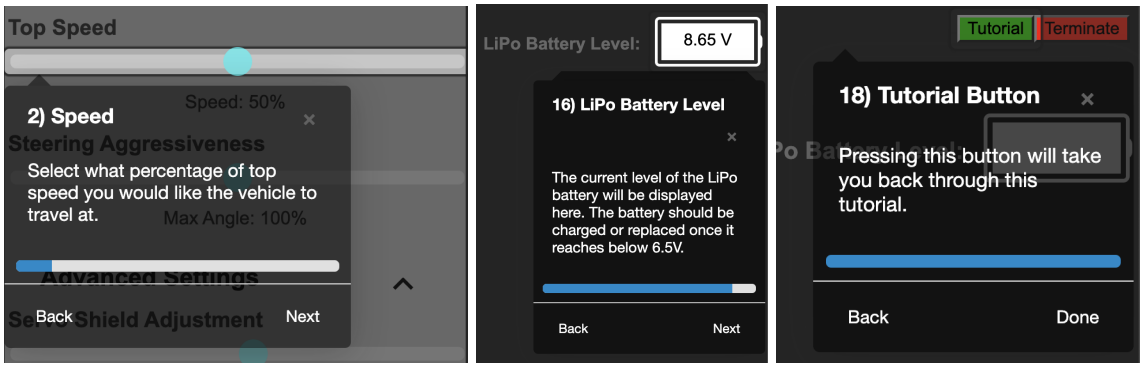

Figure 6.9: Different steps being showcased throughout the tutorial walkthrough

Our team simply utilized the introJs().setOptions() method through the Intro.js library (Mehrabani). In terms of styling we picked the "Modern" Intro.js theme and customized the tutorial to show a progress bar and disable interactions with the dashboard buttons while the tutorial was enabled. Examples of our tutorial walkthrough's style can be seen in Figure 6.9.

6.5 Manual Driving

Our manual driving feature allows the user to control the vehicle using the arrow keys on their keyboard if they have the dashboard webpage open. The necessity for this feature was to ensure that the user had wired the car properly and that the components were behaving as desired. In order to test the mechanical hardware, the user would have to disconnect mPAD and

connect the car's commercial transmitter. Integrating a manual driving feature into mPAD avoids that requirement and allows manual testing to occur. Without this feature, troubleshooting different areas of mPAD would be more difficult for novice users, since they would not be able to drive the vehicle. The manual driving also works without the camera, so mPAD could still be used if a user is missing a compatible camera. This feature was tested on multiple RC cars, either commercial or 3D printed, and is essentially the first step our team takes after setting up mPAD to ensure it was wired properly. Once the dashboard loads up, the user should switch the car into manual driving and ensure that the vehicle is capable of going in all directions.

During the tutorial walkthrough, the user will learn how to enable manual driving and that it is controlled using the arrow keys on their keyboard. The user simply needs to toggle the *Autonomous* switch, and they will be in manual driving mode. When the user disables the *Autonomous* switch, they also disable the ability to start autonomous driving.

Figure 6.10 showcases the conditional logic behind the manual driving code. The up arrow will throttle the motor in the forward direction at whatever rate the *Speed Slider* is set to. The left arrow will set the servo to 150° plus what the *Servo Angle Adjustment* is set to, ultimately turning the vehicle leftwards. The right arrow will set the servo to 30° plus the *Servo Angle Adjustment* as well, ultimately turning the vehicle rightwards. The down arrow will throttle the motor in the backwards direction at the value set by the *Speed Slider*. Section 13.3 in the Appendix showcases a code snippet with the logic for the manual driving in Python as well.

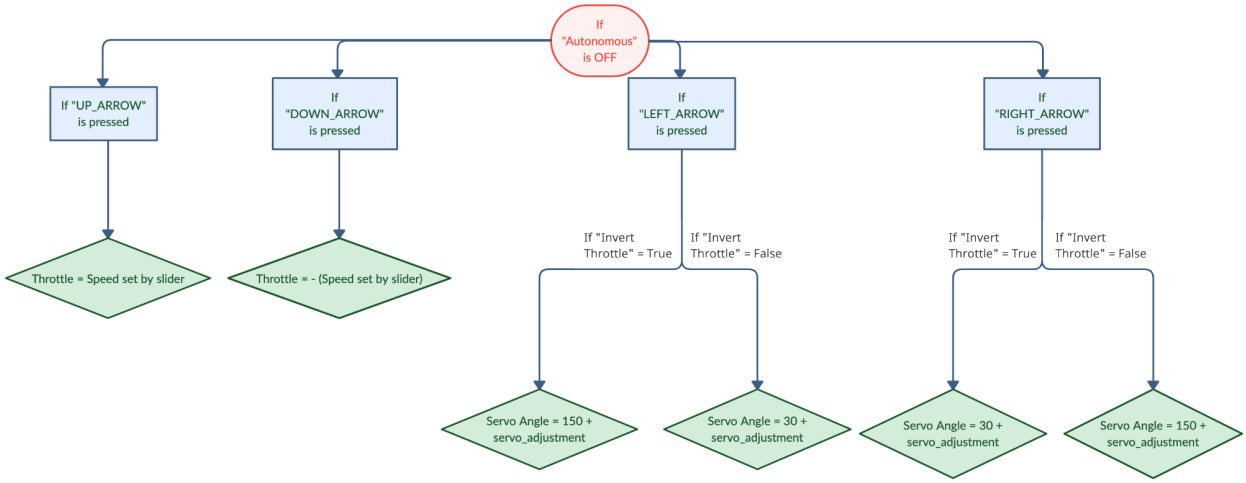

Figure 6.10: Diagram Showcasing Manual Driving Logic

6.6 Hardware Component Test Scripts

One of our goals was to add a way for the user to run certain test scripts on the vehicle via the dashboard. These test scripts allow the user to individually test the servo and motor throttle on the vehicle. If one of the scripts are not executing behavior on the vehicle, then that is a sign to the user that something is wired improperly. At the bottom of the dashboard webpage, our team added a Test Script Menu (that can be seen in Figure 6.11) that allows the user to ensure that they properly wired up mPAD and the motor and servo are behaving as desired. When the *Servo Angle* test is run, the front servo is set to an angle of 90°, which should appear straight forward, then an angle of 60° is set, which should appear as a right turn, then the angle is set to 120°, which should appear as a left turn, and finally an angle of 90° is set again as a default. The *Servo Throttle* test waits for 3 seconds, then throttles the motor in the forward direction at 15% for 2 seconds, and stops. If one of the tests is doing the opposite, so either turning in the opposite direction or throttling backwards, then that is a sign to the user to utilize the *Invert Throttle* and *Invert Steering* switches.

Figure 6.11: Front-End Dashboard Test Script Menu

Once either button is pressed, a HTTP GET request is sent to the car where a Python method is run that sleeps for 2-3 seconds and sets the servo angle or throttles the motor. This feature helped us with our goal of having a system that is intuitive. The purpose of the test scripts is to make it easy for all users to troubleshoot mPAD and check whether they made a mistake during hardware setup.

6.7 Invert Steering/Throttle Switches

One problem our team would run into when using commercially bought RC cars was that certain motors were improperly interpreting our software with mPAD. Our software is written with the intention that a positive value on the servo would move the vehicle in the forward direction, while a negative value would move the vehicle in the backwards direction. However, certain cars were driving backwards when it was intended to go forward because the motor had the opposite interpretation. In order to combat this and further reach our goals of robustness and modularity, we added an *Invert Throttle* and *Invert Steering* switch, as shown in Figure 6.12. If a car's steering or throttle is working the opposite of how our software intends, the user can simply flip the invert switch and the car would behave as desired with mPAD. This feature gives users more freedom of design as mPAD is now compatible with a wider variety of servos and motors.

Figure 6.12: Servo & Motor Channel Selection Dropdowns

The user will know that the throttle needs to be inverted if the vehicle is driving backwards when it is intended to go forward and going forwards when it is intended to go backward. Flipping the *Invert Throttle* switch will combat this issue. The user will know if the steering needs to be inverted when the car is turning right when it should be turning left, and turning left when it should be turning right. Simply clicking the *Invert Steering* switch will solve the problem.

6.8 LiPo Battery Level Indicator

Another feature that was desired by our team was warning the user when it was time to charge the LiPo batteries used in mPAD. The LiPo battery is responsible for powering the ESC and motor of the system. We wanted the user to always have an accurate representation of how much battery life is left in the system. To accomplish this, our team added a new sensor to the sensor package to constantly collect the voltage available in the LiPo battery. As shown in Figure

6.13, the current voltage in the battery is displayed and consistently updated while connected. The code is written in Arduino and parsed through via Python and displayed on the dashboard via JavaScript. The voltage available in the LiPo battery will only be displayed when the sensor package is connected, otherwise, the battery will not display a value.

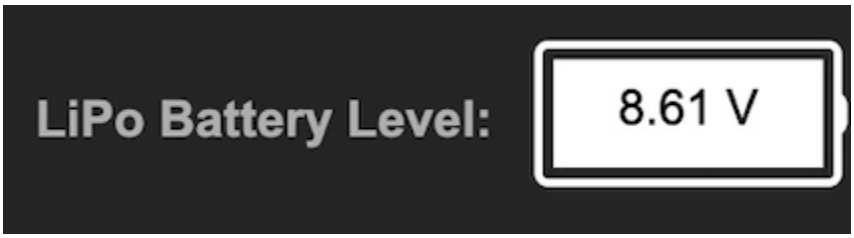

Figure 6.13: LiPo Battery Level Feature

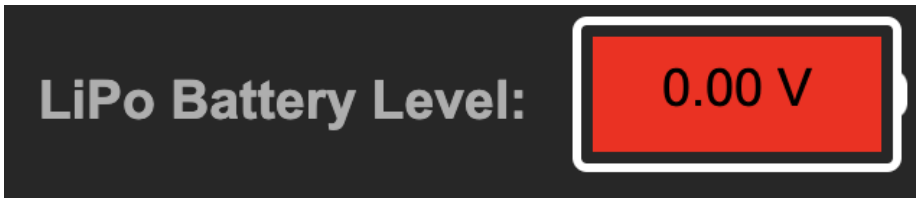

Figure 6.14: LiPo Battery Level Feature Indicating Charge or Replacement

When the voltage is below 6.5V the battery displayed on the dashboard will turn bright red and warn the user that it is time to charge or replace the LiPo battery, an example of which can be seen in Figure 6.14. The circuit utilized to detect the amount of voltage in the LiPo battery is further explained in Section 7.2.

6.9 UI/UX Updates

There were a number of UI/UX updates implemented to our version of mPAD. As mentioned in previous sections show/hide buttons were added to provide the option to hide the data being displayed. Using these buttons, which can be seen in Figure 6.15, the user can easily show or hide the data charts from the sensors or the test script menu depending on whether they are being utilized or not.

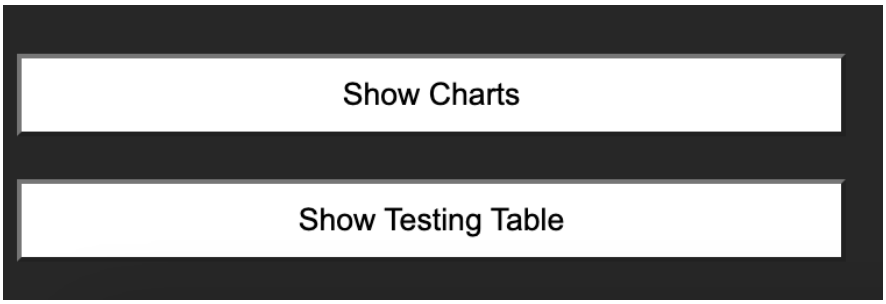

Figure 6.15: Hide/Show Buttons for Data Charts and Testing Table

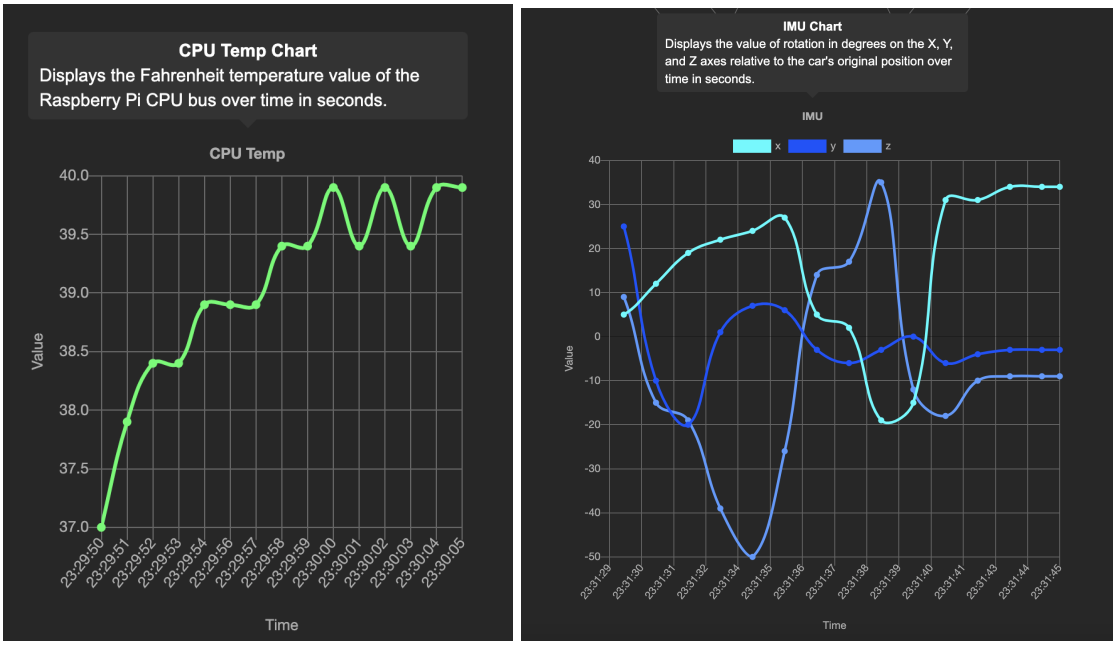

Figure 6.16: Data Chart Descriptions

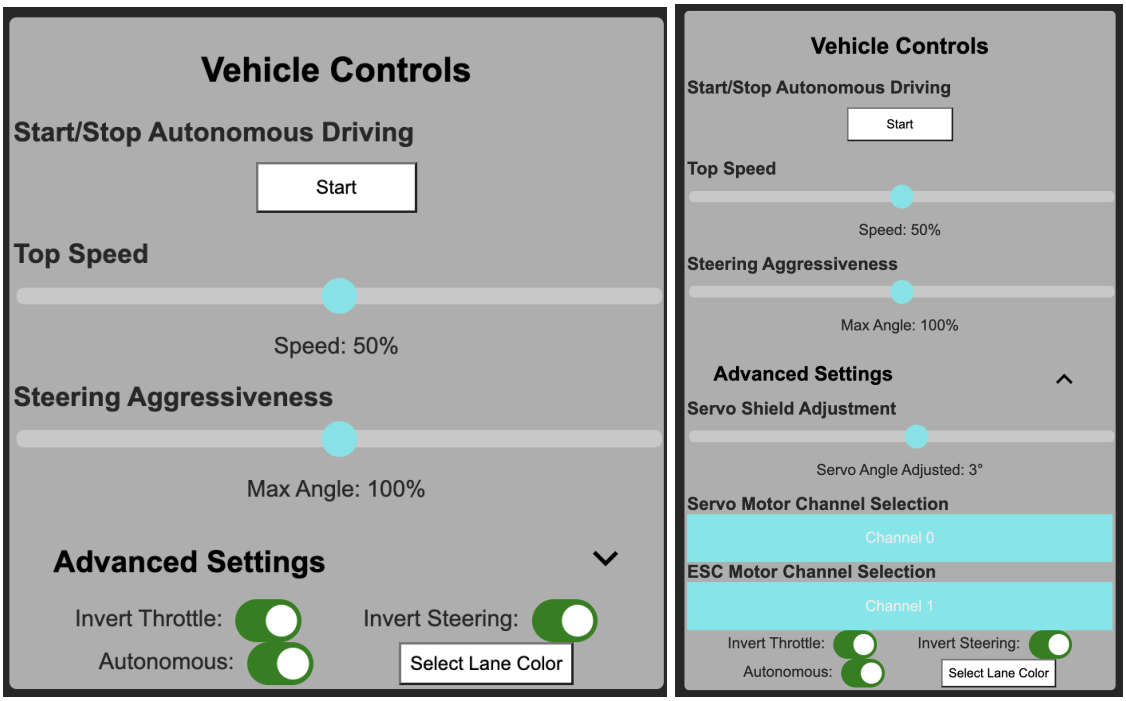

Figure 6.17: Data Chart Descriptions & Advanced Settings Dropdown

As a team, we also made the dashboard more intuitive and added descriptions to each of the sensor data charts, as shown in Figure 6.16. The user can simply click on any of the charts to be shown a brief description and learn more. Figure 6.17 shows an *Advanced Settings* dropdown that was added within the Vehicle Controls menu to hide settings that are not necessary for users at all times. Settings such as *Servo Shield Adjustment*, *Servo Motor Channel Selection*, and *ESC Motor Channel Selection* are hidden under the *Advanced Settings* dropdown. Once these settings are set after the initial set up, users will not need to change them, so they are given the option to hide these settings from the dashboard GUI. If users want to learn more about these settings, they can refer to the comprehensive tutorial walkthrough that is always available.

7. Updated Sensor Package

There have been two updates to the sensor package in this iteration of the design. The first update addressed issues with the hall effect sensor. The Arduino code was updated so that the RPM can be correctly calculated and displayed on the dashboard. The second update was the addition of a LiPo Battery charge detection circuit.

7.1 Hall Effect Sensor Update

Each RC car is equipped with a KY-003 hall effect sensor that is used to measure the RPM of a wheel or gear on the car. The sensor is placed near a wheel/gear of the car, in a position where it can read the magnetic pulses emitted by a small magnet attached to the wheel/gear. Each time the wheel or gear completes a rotation, the sensor receives a magnetic pulse and sends this information to the Arduino. The code on the Arduino counts the number of pulses and divides this number by the time taken, yielding the RPM of the wheel/gear. This RPM value is then collated with the readings from the other sensors on the car and sent as a string to the Raspberry Pi.

While testing the sensor systems on the cars, we noticed that no value was being output for the RPM. Initially we thought that this was due to a malfunction in the sensors, but they showed no sign of physical damage. Thus, we tested the hall effect sensors individually and we were able to record readings from magnetic pulses. Next, we decided to inspect the code on the Arduino for any errors or bugs related to the hall effect sensor. We noticed that the string of data values being sent to the Raspberry Pi did not include the calculated value for RPM. We also realized that the RPM calculation implementation was not properly implemented. This was the reason it would not properly display on the dashboard. After this fix, the calculated RPM value was added to the output string and sent to the Raspberry Pi. We then ran tests to ensure the code and the sensor worked as intended.

7.2 LiPo Battery Charge Detection

The team made a new addition to the sensor package to allow mPAD to monitor the charge of the RC car's LiPo battery. It should be noted that this feature only works with 2S (two cell) LiPo batteries, since this is what the team has been working with all year. The voltage charge of a 2-cell LiPo battery can be measured from the three little wires hanging off the side of

a LiPo Battery, as circled in Figure 7.1. A measurement between any two consecutive pins will show the charge of one of the LiPo cells. A measurement between the two farthest pins will show the charge of both LiPo cells combined.

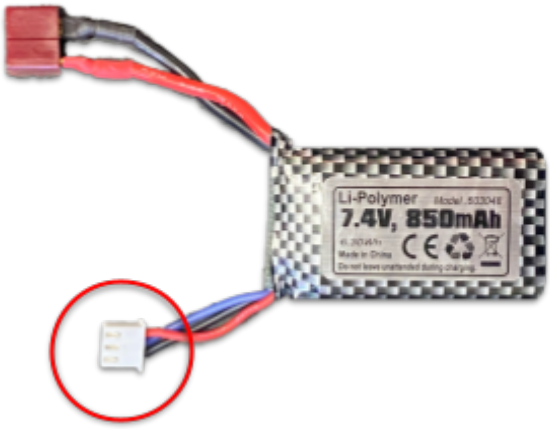

Figure 7.1: LiPo Battery

This circuit is based on a tutorial found online that uses a voltage divider to measure the voltage across one LiPo cell (Gus, 2022). The circuit designed for mPAD, shown in Figure 7.2, uses a voltage divider that uses 200Ω of extra resistance to split the voltage drop across two cells in half. This produces a voltage value that the Arduino can read on pin A0. The Zener diode also helps to protect the Arduino as well, by making sure that the voltage does not exceed 5.1V.

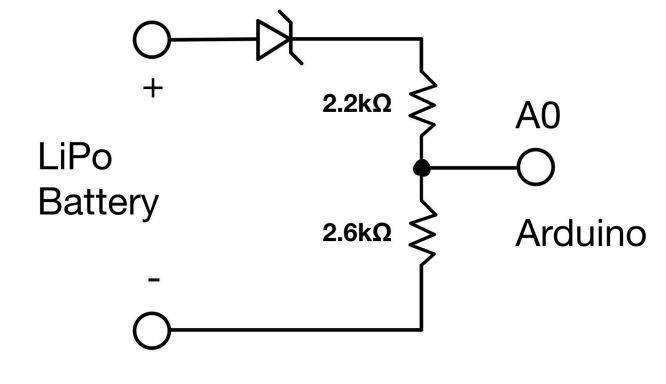

Figure 7.2: LiPo Charge Detection Circuit

Just like the other sensors' data, the LiPo charge is transmitted to the Raspberry Pi, where it can be displayed on the top right corner of the mPAD Dashboard, as mentioned previously in Section 6. When any cell on a LiPo battery falls below 3.0V, the battery's life starts to decay (Alex, 2015). Thus, when the total voltage falls below 6.5V, to be safe, the battery icon on the dashboard turns red in order to notify the user that the LiPo battery needs to be charged. The team tested and fine-tuned this circuit by comparing its values to that which was read across a multimeter and a handheld LiPo Battery Charge device.

7.3 Power by LiPo Battery

The previous mPAD system contains two batteries, a LiPo Battery and a portable power bank. These are both bulky batteries that add a lot of cargo weight, especially when attaching mPAD to a smaller scale vehicle. So the team decided to look into a solution that would power everything from just the LiPo Battery.

7.3.1 Current Power Circuit

The current schematic, as shown in Figure 7.3 below, can be split into two sections that feed power from two different power sources. The ESC takes 7.4V, 60A from the LiPo Battery, and the Raspberry Pi takes 5V, 3A from the power bank. The Raspberry Pi then powers the Arduino with 5V, and the Arduino powers the sensors with 5V.

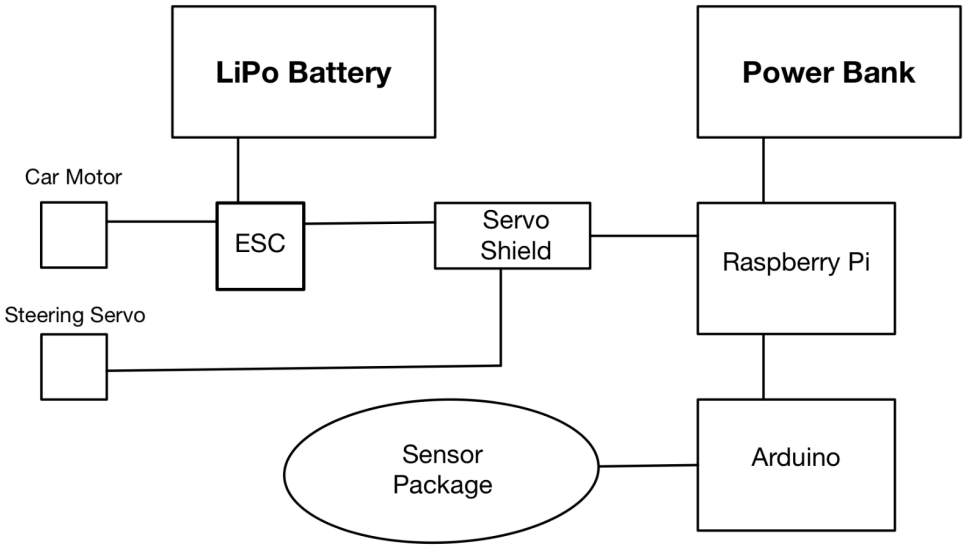

Figure 7.3: Current Circuit Design

7.3.2 LiPo Power Circuit

The team designed a new circuit, as shown in Figure 7.5, which powers everything from the LiPo Battery. The iFlight Micro BEC, Figure 7.4, was chosen to regulate the power coming from the LiPo battery, into a signal that can power the Raspberry Pi. It is a device commonly used in drones to convert the power from a 2-8 cell LiPo battery into +5V DC, 3A.

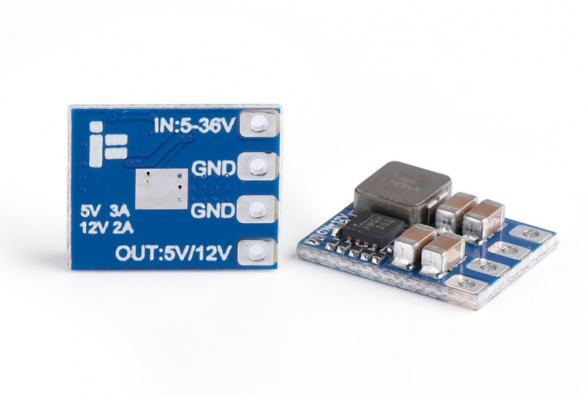

Figure 7.4: iFlight Micro BEC

reproduced as if from link

A parallel splitter is used to split the 7.4V from the LiPo battery between the ESC and the iFlight Micro BEC. The Micro BEC drops this down to +5V, 3A for the Raspberry Pi, and the Raspberry Pi powers the Arduino and sensors just as before.

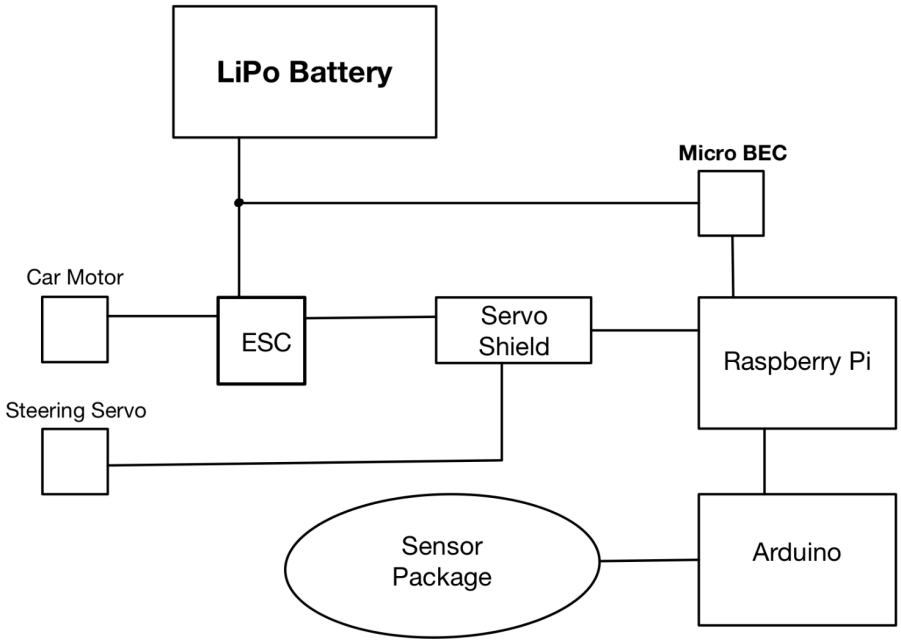

Figure 7.5: LiPo Power Circuit Design

7.3.3 LiPo Power Circuit Testing

Upon assembling the first LiPo power circuits, they were tested using a multimeter, and then with a Raspberry Pi. The multimeter read the intended +5V, but for some reason only showed a current of 500mA. These results are shown on the multimeter in Figure 7.6 below.

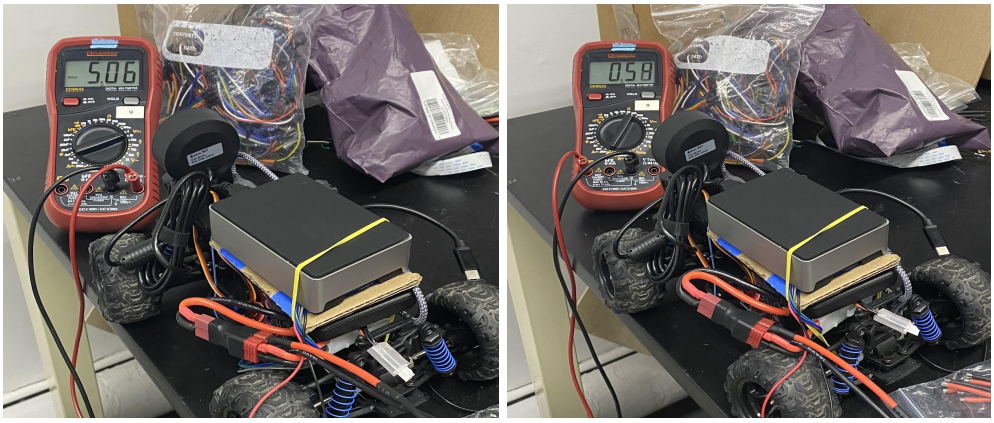

Figure 7.6: LiPo Power Circuit Multimeter Testing

However, when testing with the Raspberry Pi, it was successfully able to boot up the Pi and run the mPAD system. It was able to drive manually, but then failed in the autonomous mode. At this point in the project, there were some errors within the Raspberry Pi code causing the autonomous drive to malfunction. Thus it is unknown whether this failure was due to the LiPo Power Circuit or other problems present within the mPAD code.

8. USB Camera Implementation and Autonomous Driving Testing

As mentioned in Section 4.2, we switched from the Raspberry Pi camera to a USB webcam in order to solve the instability issues present in mPAD V1. This change also increased the overall modularity of the system as it is now able to use any available USB webcam instead of a proprietary one. The following gives an in-depth description of the entire process that the team undertook in order to switch the system over to a USB camera and ensure that autonomous driving was consistent with it.

8.1 Attempted Fixes for the Raspberry Pi Camera

Before switching to using an off-the-shelf USB webcam, the team took a number of steps in order to attempt to fix the instability issues that the Raspberry Pi Camera was creating. As mentioned in Section 4.2, while the Raspberry Pi camera would be consistently detected if it was the only device plugged into the Raspberry Pi, connecting other devices would render the detection of all connected peripherals inconsistent.

Initially, the team attempted to find an order for which components should be connected to the Raspberry Pi to ensure that each would be detected by the system. To do this, we cycled through every different possible order of connecting the camera, servo shield, and Arduino to the Raspberry Pi and tested whether or not the system would consistently detect all components by starting the system five times for each configuration. If the system successfully detected all connected components at least four out of said five times for a configuration, we would consider it as “stable”. Unfortunately, none of the configurations met our desired criteria.

It was at this point that the team decided to start investigating alternative camera solutions that would potentially sidestep the problems occurring because of the Raspberry Pi camera. There were two main alternatives: a different camera that utilized the same ribbon connector as the Raspberry Pi camera, or a camera that would connect to the Raspberry Pi via a USB cable. We decided to test the system with a team member’s webcam, then test an alternate ribbon connector camera if the system did not work well with a USB webcam.

8.2 USB Camera Implementation

Fortunately, initial testing with the team member’s USB webcam yielded promising results. Switching the software to detect and utilize the USB camera instead of the Raspberry Pi

one was a relatively simple process, and upon running the system with the USB webcam attached to it we encountered none of the component detection issues described in Section 4.2, no matter what order each component was connected in. The camera we tested with was an Angetube streaming webcam and had a resolution of 1080P, an attached ring light that could be turned on in low-light situations, autofocus, and a 78° field of view ("Angetube Streaming 1080P HD Webcam Built in Adjustable Ring Light and Mic. Advanced autofocus AF Web Camera for Google Meet Xbox Gamer Facebook YouTube Streamer," n.d.). We tested the solution with multiple different hardware configurations (e.g. different Raspberry Pis, servo shields, arduinos, etc. of the same model) and found that each time every connected component would be detected without issue.

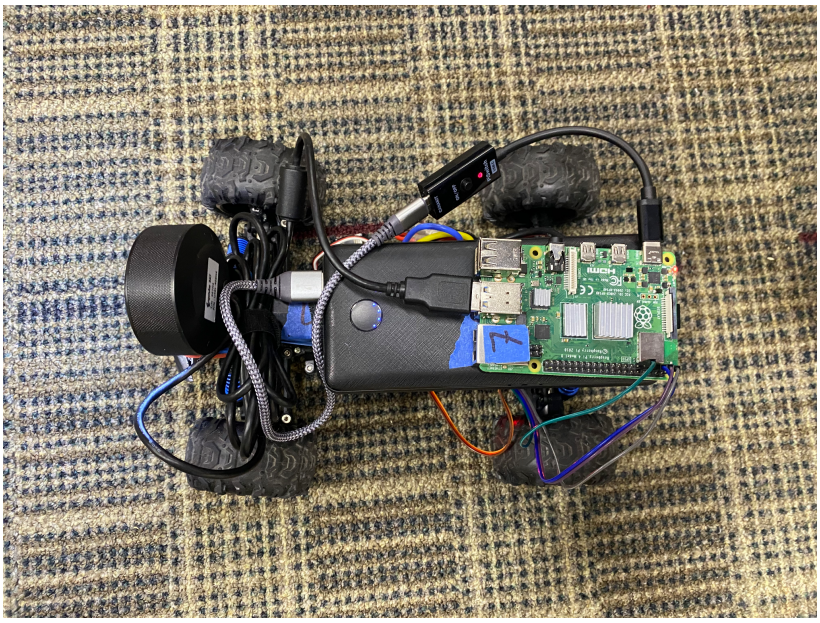

Figure 8.1: USB Camera and mPAD Mounted to RC Car

8.3 Testing Self-Driving with the USB Camera

Changing the camera hardware meant that the team had to rediscover the optimal mounting position and angle of the camera in order for the system's self-driving to work. As such, the team decided the system has consistent self-driving if it is capable of autonomously driving for at least five laps around a test track. The car we used for the majority of our self-driving testing - shown in Figure 8.1 - was a commercial RC car purchased from Amazon

with the mPAD system attached to it. The car is a 1:18 scale model equipped with a brushed motor, four-wheel drive, a spring suspension, a servo for turning, and is compatible with 7.4V LiPo batteries ("VCANNY Remote Control Car, Terrain RC Cars, Electric Remote Control Off-Road Monster Truck, 1: 18 Scale 2.4Ghz Radio 4WD Fast 30+ mph RC Car, with 2 Rechargeable Batteries," n.d.). The car did not have the sensor package attached to it for most of the testing in order to ensure that issues we encountered could be easily attributed to camera and self-driving aspects of the system. Additionally, we found that mounting the camera to the front of the car using a rubber band was sufficient to keep it in place, as we could still adjust the angle of the camera if needed but did not need to worry about the camera falling off the platform while testing the car.

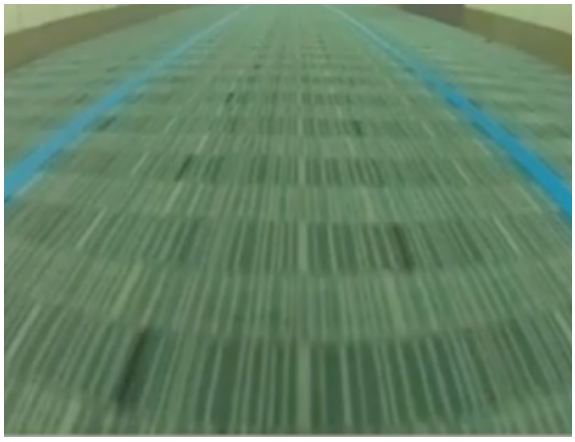

Figure 8.2: Example Dashboard Camera View

Initially, the team mimicked the position and angle of the original Raspberry Pi camera with the new USB camera by placing it roughly eight inches off the ground and at a 60° angle relative to the face of the car (Giglio de Azevedo et al., 2021). Unfortunately, the self-driving logic failed to keep the car consistently within the lanes with this setup. Afterwards, the team adjusted the camera height and angle slightly and retried the self-driving, repeating the process until an angle and height at which the car could self-drive consistently was found.

Namely, we found that keeping the camera near the car and anywhere from 4.5 to 5.5 inches off the ground and angling it so that the edges of the lane leaves the camera view slightly above the vertical center of the frame produced the most consistent self-driving, which can be seen above in Figure 8.2. A diagram of the recommended configuration can be seen below in Figure 8.3. However, even with the new camera in its most optimal position, the system still was only able to complete up to a few laps at a time before, much less achieve the team’s definition

of consistent driving. Videos documenting the car's performance at this time can be found in Section 13.4.2 of the Appendix. Thus, with the hardware aspect of self-driving optimized, it was time to optimize the autonomous driving software of the system.

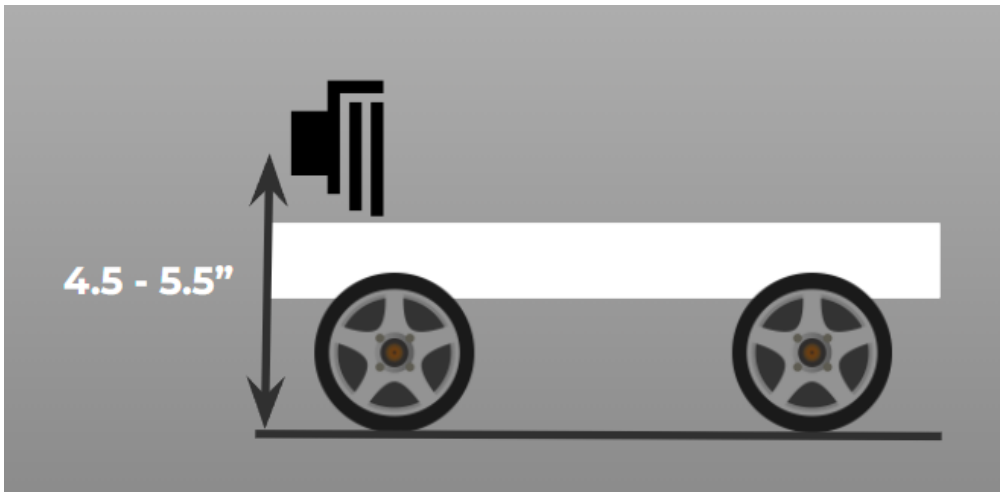

Figure 8.3: Camera Height and Position Diagram

8.4 Self-Driving Logic Adjustments and Testing

The team's work on adjusting the self-driving logic of mPAD started with an in-depth examination of the part of the codebase that the autonomous driving feature relies on. This involved both checking the code for glaring logical errors as well as reviewing the report by Giglio de Azevedo et al. from WPI's 2020-2021 academic year to confirm that the system was, in fact, able to drive autonomously consistently. Neither the self-driving logic nor the paper suggested that there should be any issue with the behavior of the self-driving system, and examining them gave us an intimate knowledge of how the system should behave in a given situation. As such, instead of simply evaluating the overall performance of the system like we did while testing the hardware, we instead examined a car's behavior in a situation and compared it to the expected behavior of the self-driving logic. While testing with this new strategy, we noticed that the car would sometimes behave unexpectedly. Namely, sometimes the car would perform as if it did not detect the lanes even though it does detect them, and other times the car would steer in response to the track even though no track was selected yet. This unexpected behavior suggested that the problem lay in one of two places:

1. The car steering logic

2. The color filtering logic

We examined the color filtering logic first as we had already gone through the steering logic during our earlier review of the self-driving logic, the details of which can be found in Section 2.3. During said examination, we realized that the values being used to filter the color of the lanes appeared to be hardcoded, and the variable that held the true color values obtained from the dashboard was left unused in the driving logic.

Thus, we switched the hardcoded values to the variable set by the dashboard, and once again tested the self-driving functionality. We then once again ran self-driving while both evaluating its overall performance and examining whether it reacted to its environment as expected. During each of these runs, we observed that while at first, the car would behave as expected, once it left the general vicinity of its original position on the track it would behave as if it did not see the track again. As such, we checked to see if the software was correctly detecting the lane once the car left its original area. It was not, and as such simply selecting the lane so that it could be detected in the new area again mostly solved the problem. Due to this, we realized that the problem stemmed from differences in the lighting on different parts of the track. When setting up the car, we simply needed to ensure that the lane color filter created via the dashboard lane color selected accounts for the difference in appearance of the track in different lighting conditions.

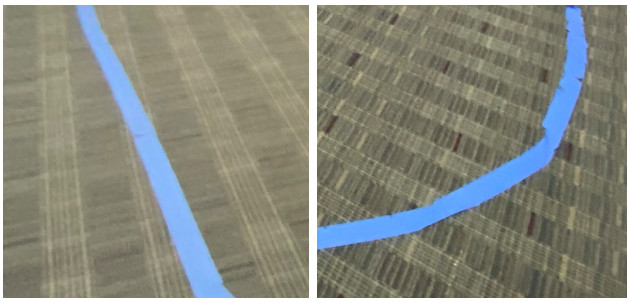

Figure 8.4: Effects of Lighting on Different Parts of Test Track
Left: Brighter Lighting
Right: Dimmer Lighting

We then once again tested the system to see if it met our standards for consistency. Initial results were promising, with our test system successfully completing over five laps around our test track. However, within roughly half an hour on subsequent trials we noticed that the car

would often fail to successfully complete a turn where it had previously been able to do so.

We attributed this problem to the possibility that the system was overheating its host Raspberry Pi due to the fact that the computer was hosting both the server and self-driving portions of the software. We tested this by stripping out as much of the server communication as possible (e.g. camera display on the dashboard, live sensor data, etc.) and then autonomously driving the platform over a longer period of time to see if we would run into the same issue.

Cutting out unnecessary server communication seemed to delay the performance issues, but not completely eliminate them. We were able to have our test car self-drive for nearly an hour before it failed due to lag. As such, we added logic to the system that would disable the camera feed from being displayed on the dashboard (but still utilized for the self-driving logic itself) in order to avoid overheating the Raspberry Pi as quickly. Unfortunately, any further work on solving or mitigating the performance issues with the system could not be completed due to time constraints, and as such our suggestions for further work on the topic will be outlined in Section 11.1.4. Despite said performance issues, however, our system was now able to meet our definition of consistent self-driving. Namely, it was able to complete more than five laps around our test track multiple times. Footage of testing at this stage of development can be found in Section 13.4.3 of the Appendix.

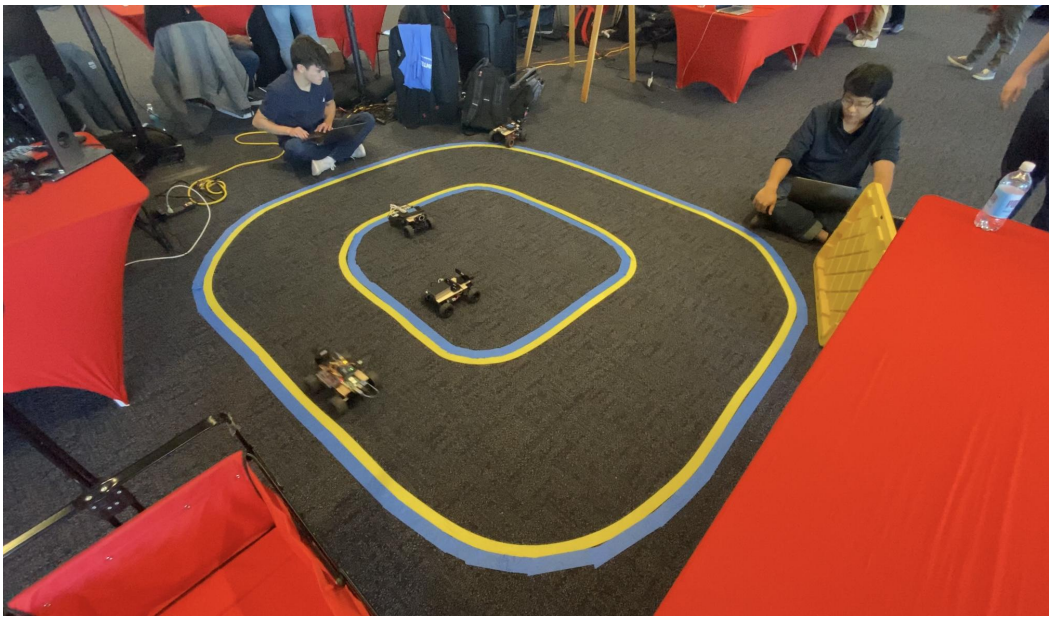

Figure 8.5: mPAD on Display at Touch Tomorrow

This was demonstrated during 2022's Touch Tomorrow event, where after some initial issues we eventually were able to run self-driving consistently on a different track, which can be seen above in Figure 8.5. These issues initially consisted of suboptimal conditions created by lighting problems generated by pedestrian traffic occurring between the track and the windows - which initially let in a large amount of light due to the time of day. The windows also washed out the image whenever they entered the camera's view the light was so bright it would effectively blind the camera. In an effort to minimize the effects, we placed several obstacles to block the track from the unwanted lighting, which succeeded in solving the problem related to pedestrian traffic but failed to completely fix the bright window problem. As such, the team was forced to make the camera face more towards the ground in order to correctly see the lane without any interference. Unfortunately, the track featured a sharp turn that the cars were unable to complete correctly due to this change of camera angle, which we fixed by making the turn less abrupt. After these problems caused by the suboptimal conditions at Touch Tomorrow were mitigated, we were able to have several cars consistently self-drive at the event.

8.5 Self-Driving Setup Paradigm

As mentioned previously, a certain amount of setup is needed in order to enable an RC car equipped with mPAD to self-drive consistently. Namely, the camera needs to be mounted at the correct angle and the lane color filter needs to account for any potential lighting differences around the track. As such, the team developed a paradigm for finding the correct software settings for the car to autonomously drive consistently. The paradigm is as follows:

1. Set speed, steering, and servo adjustment values.
2. Adjust the camera angle so that the edges of the track exit the view around the middle of the frame.
3. Select lane color
4. Start self-driving and continue until the car fails to correctly follow the lane
5. Stop self-driving and place the car at the location where it failed to follow the lane
6. Repeat steps 3 to 5 until self-driving behaves consistently

While we had previously tested the system and ensured that it could meet our definition of consistently self-driving by being able to drive around our test track more than five times, we still needed to test the system's consistency when it is set up using the paradigm. As such, we ran

several more trial runs, this time making sure to set up the software by following the paradigm, and noted the overall performance of the system during each run. With the exception of the last - which likely failed due to the performance issues - the test car was able to successfully complete five laps around the track without failure during each trial. Additionally, during the Touch Tomorrow event mentioned in Section 8.4, the team used this paradigm to get the cars self-driving. Footage of the paradigm at work on various systems can be seen in Section 13.4.4 of the Appendix.

9. Object Detection Testing

The team started to explore three new object detection devices, the RPLiDAR, PixyCam2, and the HuskyLens, as explained in Chapter 3. The team performed testing on all three of these devices to pinpoint the one that will have the best impact on mPAD. The PixyCam2 and HuskyLens are very comparable due to the type of data collected. As shown in Figures 9.1 and 9.2, they both utilize a camera to draw boxes around detected objects, and return the coordinates and size of those boxes.

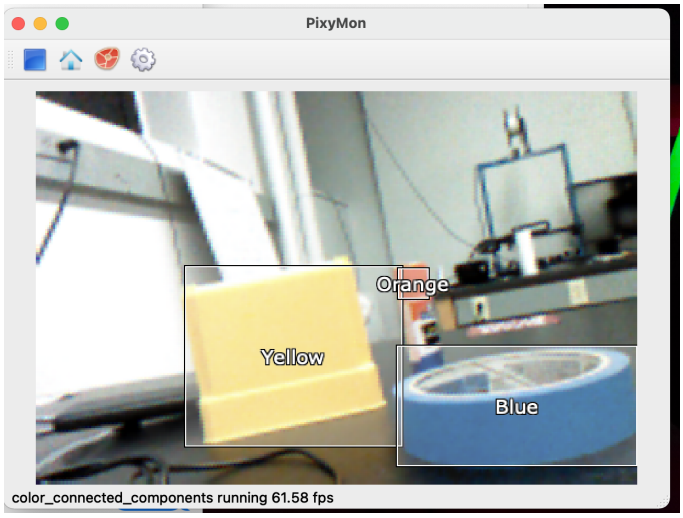

Figure 9.1: Pixy Cam Color Detection Testing

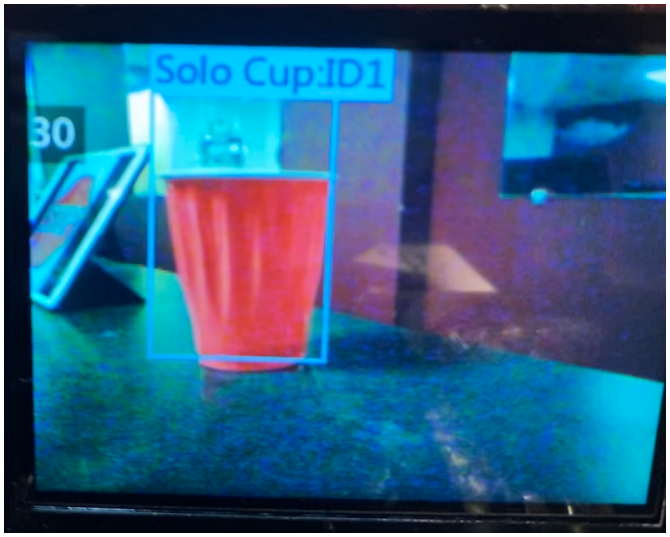

Figure 9.2: HuskyLens Object Recognition Testing

The RPLiDAR, on the other hand, is a little different as it just provides 360° two-dimensional coordinate points of physical objects in its environment as shown as red lines in Figure 9.3. These red lines could represent any physical object within the LiDARs range such as a wall or a vehicle. It could be better compared to the ultrasonic implementation that Giglio de Azevedo et al. had put together in 2021.

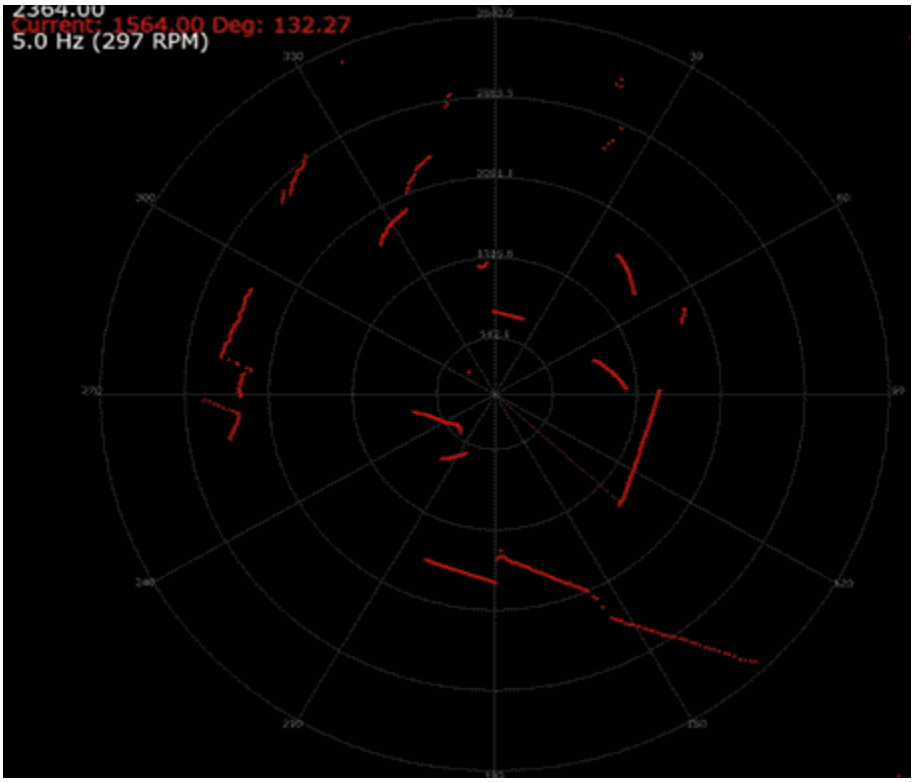

Figure 9.3: RPLiDAR Data Display

9.1 Vision Sensor Selection for Obstacle Detection

Power consumption was one factor considered due to the issues relating to power that were experienced during the beginning of our project. It can be noted that the RPLiDAR consumes the most since it requires power to both the laser and the motor spinning the laser. This is the reason the LiDAR was not chosen. These values are summarized below in Table 9.4.

Table 9.4: Power Requirement Summary for Researched Devices

Component	Start-up Voltage	Start-up Current	Operating Voltage	Operating Current
RPLiDAR (motor)	+5V	500mA	+5V	300mA
RPLiDAR (laser)			+5V	100mA
PixyCam2	-	-	+5V	140mA
HuskyLens	-	-	+5V (or 3.3V)	230mA (or 320mA)

Another factor that was heavily considered was the compatibility with mPADs current design. It must be programmable via a Raspberry Pi or Arduino, and provide data that can be used in the drive algorithm. Table 9.5 shows a comparison chart for the feasibility of the systems in question.

Table 9.5: Feasibility Summary for Researched Devices

Component	Price	Compatibility	Functionality
RPLiDAR	$99.99	Raspberry Pi	360° Coordinates
PixyCam2	$59.99	Raspberry Pi, Arduino	• Color Detection • Line Tracking • Barcode Detection
HuskyLens	$44.90	Raspberry Pi, Arduino	• Color Recognition • Object Tracking • Object Recognition • Line Tracking • Tag Recognition • Face Recognition

Between the two vision sensor modules, the HuskyLens was determined to be the most optimal choice because it has better recognition functions and a cheaper price. The PixyCam2

can only detect objects based on their color, which could cause some confusion for the device if seeing other objects that are of similar color to what is intended on being seen.

9.1.1 Software Comparison

The PixyCam's Python software library has very little documentation ("Charmedlabs/pixy2," n.d.). The extent of it is a small instructional page on how to install the library on a Linux system and a set of scripts to demonstrate how each different function works. Additionally, the functions the library offers are barebones and lack quality-of-life features such as changing the settings for what the user wants it to detect.

Table 9.6: Python Library Feature Comparison for PixyCam and HuskyLens

Device/Feature	Switch Object Being Tracked	Train to Track a New Object	Get Position Information of Tracked Object
PixyCam	No	No	Yes
HuskyLens	Yes	Yes	Yes

In contrast, the HuskyLens Python software library provides interested parties with a large amount of documentation on its GitHub page (Prast, 2020). These differences can be seen in Table 9.6 above. Like the PixyCam library, the HuskyLens library (Prast, 2020) also provides an example script to show how each of its functions work in practice. Finally, the number of functions allow the user to utilize the entirety of functionality of the vision sensor through code, instead of having to use the GUI like with PixyCam.

9.2 HuskyLens Road Sign Detection

The team started to move towards utilizing the HuskyLens for road sign detection. If mPAD could gain the ability to read road signs, it could react better to the road in front of it. For example, if there is a sharp turn ahead that the car may need to slow down for, a sharp turn sign would help signify this. This would also apply better to real-life autonomous driving, by allowing the car to follow speed limits and stop at stop signs.

In order for the HuskyLens to recognize objects, it needs to be taught said objects using the learn function on the HuskyLens. This can be done using the learn button located on the top of the HuskyLens device, as circled below. Learning can also be done programmatically using the Python or C/C++ library.

Figure 9.4: HuskyLens Learn Button

The team trained the HuskyLens using both of these methods, however it should be noted that the programmatic method is best practice in order to make the HuskyLens configurable on the mPAD Dashboard. The user points the HuskyLens camera at an object, and presses the learn button to teach this object to the recognition algorithm.

The HuskyLens was trained using a 3D printed stop sign that was placed at different locations around the testing track, and also outdoors on WPI campus. In order to improve the versatility of the algorithm, the HuskyLens was also trained using pictures of various stop signs on Google Images.

Figure 9.5: Google Images Stop Signs

After training, the HuskyLens was able to recognize stop signs while driving by, as shown in Figure 9.6. However, the HuskyLens also produced false positive values when seeing things such as someone with a red backpack (Figure 9.8), or lights on the ceiling (Figure 9.7). It is believed that this confusion for the lights on the ceiling is due to all of the pictures learned to the device which included the stop sign and the lights.

Figure 9.6: HuskyLens Correctly Detecting Road Sign

Figure 9.7: HuskyLens Falsely Detecting Ceiling Lights as Road Sign

Figure 9.8: HuskyLens Falsely Detecting Red Backpack as Road Sign

Perhaps, these errors could be improved upon with more training data for the HuskyLens. There are several different AI control settings that can be tampered with to receive different results, including NMS Threshold, Recognition Threshold, light exposure and more. These settings affect the reliability of the HuskyLens to accurately detect a learned object. On one end of the spectrum, these settings can detect objects very well, but produce many false positive results. On the other end of the spectrum it can detect learned objects less frequently, and in turn becomes more accurate with less false positive results. If the HuskyLens were built into the mPAD Dashboard as a chart, these settings could be tested and fine tuned to make the HuskyLens object recognition more reliable.

10. Discussion

The development of the second version of the mPAD is meant to accomplish the technical design objectives of being more robust, modular and intuitive than its predecessor. This selection outlines the process of achieving the previously stated objectives as well as the broader reaching impacts of our project.

10.1 Make mPAD More Robust

As stated earlier, one of the core objectives of mPAD V2 was to make the system more robust. Namely, the system software needs to be stable enough that crashes are a rarity and consistent enough to drive a minimum of five laps on any track under optimal conditions. As a result, a large amount of work went towards improving the stability of the software package and increasing the consistency of the self-driving.

One of the major changes made to mPAD aimed at increasing the stability of the system by implementing a new USB camera. The camera that Giglio de Azevedo et al. used for mPAD V1 produced stability problems that would often cause the system to fail to start. Changing the system to use an USB camera largely solved this issue, making the system very stable and resistant to crashing. As such, most of our further work focused on increasing the self-driving performance of mPAD V2.

Our changes aimed at increasing the performance of mPAD V2 were twofold. Firstly, we noticed a number of bugs related to the lane detection logic, and as such made changes to the code to address them. We then developed a paradigm for a user to follow in order to obtain the correct settings to be able to autonomously drive the car and meet our definition of consistency. Next, we attempted to address performance issues that negatively impacted the consistency of the system by cutting out unnecessary communication between the server backend and the self-driving logic. While these changes did increase the amount of time that could elapse before the system started suffering from performance issues, they did not eliminate them completely.

Overall, the changes made - which are described in more detail throughout Section 8 - ultimately helped the team meet our goal for making mPAD more robust. Despite the fact that performance issues related to the backend systems overloading the Raspberry Pi are still present, the work done to improve the consistency of both the software itself and its self-driving had tangible effects on the system as a whole, allowing it to drive a minimum of five laps on multiple

tracks under optimal conditions. This was demonstrated both through the team's in-lab testing of the system after we had made the software changes to it, videos of which can be seen in Appendix Section 13.4.3, and through the Touch Tomorrow 2022 event, where multiple cars equipped with mPAD were able to complete over five laps on a track that had been set up earlier that day.

10.2 Make mPAD More Scalable

Another key objective of mPAD V2 was to make the system more scalable. The end goal would be to see if mPAD could be scaled up to a full sized vehicle. To achieve this long term goal the team set an objective to test mPAD V2 on vehicles of different sizes. The largest size being a toddler vehicle as shown in Figure 10.1.

Figure 10.1: Size reference image: Toddler Vehicle for Scalability testing of mPAD V2

reproduced as if from link

One aspect of scalability in mPAD V2 is the freedom to pick and choose which hardware and software components to use. This is especially important to the smaller vehicles as they may not have enough space to fit the entirety of mPAD V2 due to lack of room on their chassis. To run the self-driving code the only hardware necessary is the vehicle, a Debian based system that has USB ports, wireless connectivity, and an i2c bus (in this case the Raspberry Pi), a power source for said device, the ESC, and a servo shield. Being able to adapt the hardware based on the user's needs was a design intention carried over from mPAD V1, and expanded upon in this

version. This was done by testing on vehicles of different sizes and testing the system using specific components and features of the package. Initial testing was done on vehicles created by students. Some examples of cars used early on are shown in Figure 10.2. These vehicles were all different sizes and could only accommodate certain features of mPAD V2. For example, the middle vehicle in Figure 10.2 was a large vehicle with enough real estate on its body to fit every component of the sensor package, while the right most vehicle was much smaller and could not accommodate the ultrasonic sensors. During testing with these vehicles we found that mPAD V2 can work on vehicles of different sizes, however we were limited by the hardware used to build the vehicles, since after only brief amounts of testing the hardware components of most of the vehicles would fail. This encouraged the team to find another alternative to continue testing. Further testing was done, as seen in Section 8, on small purchased RC vehicles as seen in Figure 10.3. Through our testing with RC cars of various shapes and sizes we discovered that mPAD could still successfully autonomously drive and be adapted to the users needs.

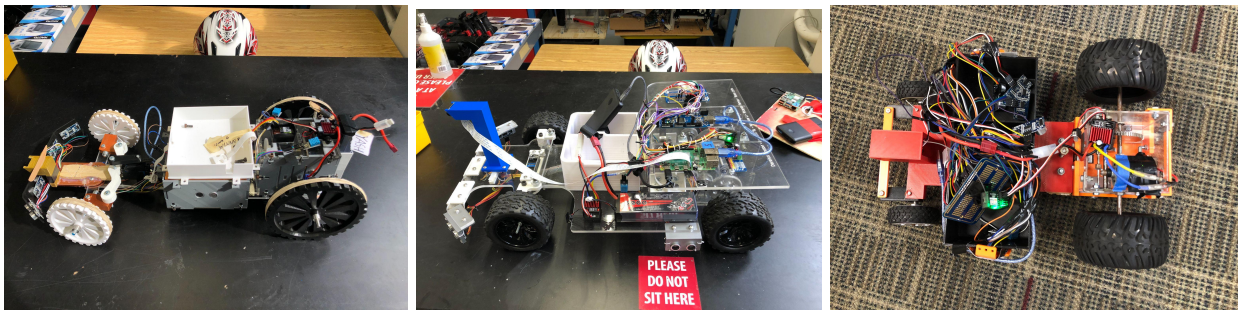

Figure 10.2: Vehicles used for early testing of mPAD V2

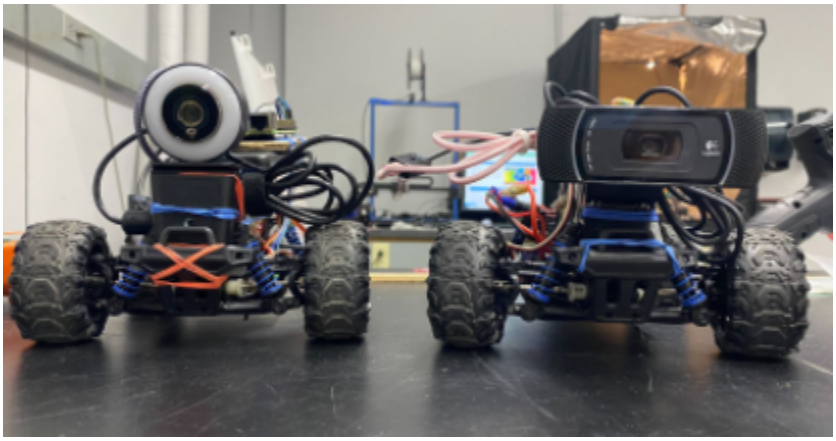

Figure 10.3: Vehicles used for later testing of mPAD V2

Currently, mPAD V2 successfully works on RC scaled vehicles. One component that we did not test was how well the system would perform if we were to scale the package up. The team had the intention of purchasing a toddler sized vehicle and testing the software package on a vehicle of that size. Unfortunately, the team was unable to achieve this goal due to time devotion in what the team felt were more important aspects of the project.

10.3 Make mPAD More Intuitive

The third portion of the team's objective was to make mPAD more intuitive so that students of all skill levels can assemble and run the autonomous driving package. The team accomplished this by creating a simple, easy-to-follow setup guide for the construction of hardware and installation of software. This setup guide was designed to look like an instruction manual that you might find for a LEGO set or children's toy, using colors and arrows. Some pages of the guide can be seen in Figure 10.4 below and the complete guide can be found by following the link in Appendix Section 13.2.

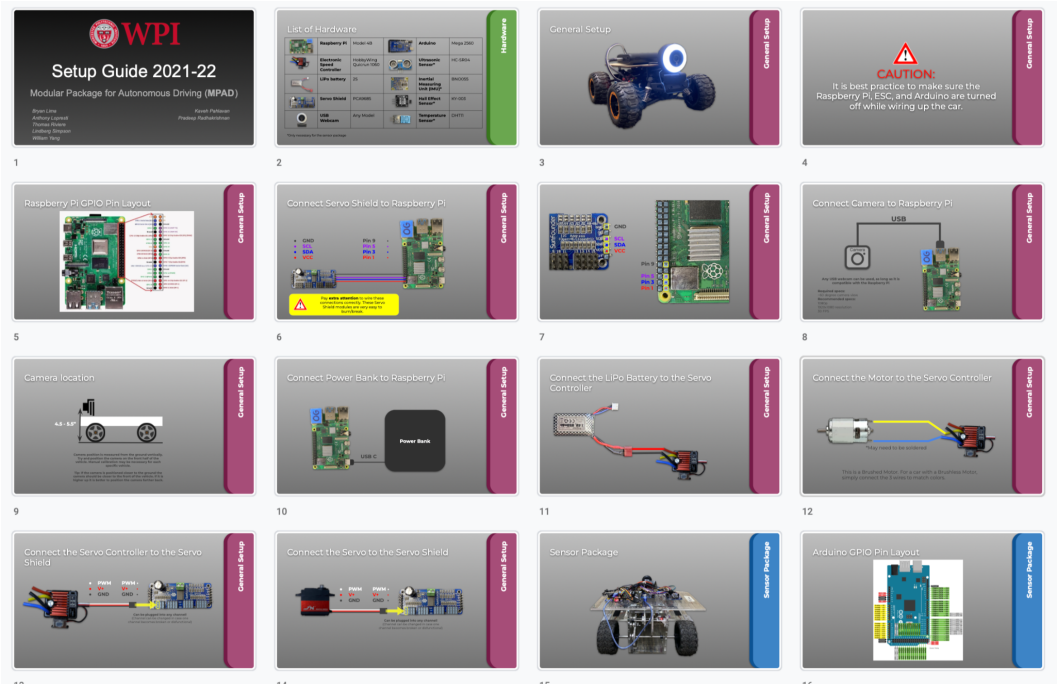

Figure 10.4: Setup Guide Screenshot

The setup guide steps through each hardware component and shows exactly where and how to wire it. It even includes warnings in areas where critical errors are commonly made. Besides this, the setup guide also shows how to install the code to the Raspberry Pi using the github link and startup script, which can be done in just a couple lines of code on the Linux terminal.

As mentioned in Section 6, the team also implemented a tutorial that walks the user through each button and chart on the mPAD dashboard. This can be accessed by pressing the "Tutorial" button at the top right corner of the dashboard page. Chapter 6 also mentions some additions to the dashboard that will allow students to troubleshoot the car better and recognize where things might be going wrong. Two buttons were built into the bottom of the dashboard to initiate throttle and steering test programs. Using these, students can evaluate whether the car's motors are functioning properly. A LiPo Battery charge monitor was also built in to notify students when the car's battery needs to be charged if the compatible sensor hardware was attached to the system. Finally, a manual driving feature was added to make trivial things such as moving the car when not self-driving less difficult. As a result of these changes, the team believes that we have sufficiently made the setup process much smoother and straightforward.

10.4 Broader Impacts

The following section will cover several broader impacts that are outside the scope of the project.

10.4.1 Engineering Ethics

Throughout the project, the mPAD team took every course of action with consideration to the IEEE Code of Ethics. This includes holding honesty to the quality and competency of the product designed, ensuring that the results were safe and sustainable for all human beings and the environment, and taking into account awareness for the inclusion and emotions of other persons. Even more so, each team member was encouraged by each other to uphold these values.

10.4.2 Societal and Global Impact

Our project has impacts on society from an educational perspective. One of the main goals of this project is for it to be used by students in classrooms at the collegiate level. Due to

the interdisciplinary nature of this project, it can teach students about concepts in fields such as Computer Science, Electrical and Computer Engineering and Mechanical Engineering. Further analysis and development of this project can also show students the ways in which different majors intersect and can encourage cooperation between students with different specialities. It is important that students get exposed to this technology as they may be designing cars in the future that apply these concepts.

Also, this project can have a major impact on elementary and high-school level education as well. We showcased mPAD V2 at a science fair for young students called “TouchTomorrow 2022” and noticed that several students were fascinated by the practical aspects of our project. If the package is employed in early education, it can be used to stimulate interest in STEM and encourage more students to pursue education in engineering fields.

10.4.3 Environmental Impact

Our project does not have any major impacts on plants or animals, but it does have a significant impact on the environment. The idea of autonomous driving and electric vehicles as a whole has an enormous positive impact on the climate in comparison to cars that run on gas. mPAD runs on a LiPo battery and it is important for mPAD users to properly recycle or dispose of their LiPo batteries, so that they do not end up having a negative impact on the environment due to being improperly disposed of.

10.4.4 Codes and Standards

The SAE Levels of Driving Automation defines a series of levels from 0-5 that denote the capability of a self-driving vehicle ("SAE Levels of Driving Automation™ Refined for Clarity and International Audience," 2021). Figure 10.5 below provides details on the expected functionality of each Level. mPAD V2 most closely meets SAE Level 1, as while it does not have the ability to adjust its speed depending on its surroundings, the system does have the capability to center itself in a lane due to its lane-following logic.

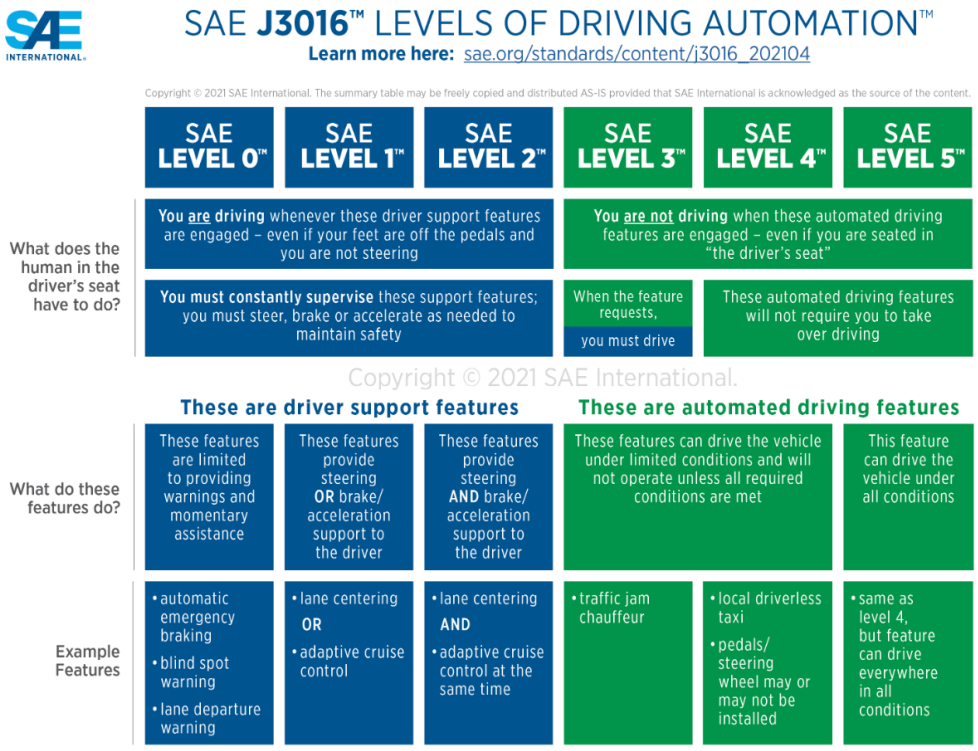

Figure 10.5: SAE Levels of Driving Automation

("SAE Levels of Driving Automation™ Refined for Clarity and International Audience," 2021)

10.4.5 Economic Factors.

Budget was not a large consideration for this project. However, there were not a lot of very expensive purchases. Overall the total cost of mPAD V2 was calculated in Appendix 13.1. The total cost of the system currently ignoring the inflated price of the raspberry in era post COVID-19 is $325. Of course, because of COVID-19, there are shortages of some widely used electrical components. Luckily for mPAD V2, the only item that has really been affected is the Raspberry Pi 4. This is due to manufacturing being outsourced to international locations, which are closed to trade for a variety of reasons due to factors related to the COVID-19 pandemic. Overall a smaller price tag supports that goals of this project even if it was not intended. The purpose of this package is for educational use, so many parts of this package may be purchased in bulk, since having multiple vehicles may have benefits in this type of environment.

11. Conclusion

As suggested in the Discussions section, the team believes that we have successfully met the objectives for mPAD V2. Namely, we have improved the robustness, scalability, and intuitiveness of the system by introduce changes including but not limited to implementing a new camera, enhancing the self-driving code, increasing support for different sensors, testing the system on cars of varying sizes, adding quality of life features to the dashboard, and introducing a comprehensive setup guide for the system. Unfortunately, despite our best efforts, there is still much work to be done in improving the overall performance of the system as well as in implementing object detection and avoidance into the self-driving logic, both of which the team wishes we were able to implement.

Additionally, our definitions for robustness, modularity, and intuitiveness were somewhat vague, which makes it difficult to evaluate just how well we met our goals. Our definition of robustness was that the system could be capable of running at least five times around a track. However, we do not mention what track we are measuring this with. The project's definition of modularity similarly runs into an issue - we state that the system should be able to work on multiple scale RC cars, but fail to state any specific hardware requirements like size and shape to measure our progress against. Finally, we also fail to specify any way to measure the intuitiveness of our system and instead rely on a general sense of ease of use to inform our progress on improving the intuitiveness of mPAD.

Despite the above issues, however, there are tangible improvements to mPAD V2 when it is compared to mPAD V1, and as such we are comfortable with claiming that we have largely met our goals for the project.

11.1 Future Work

The following recommendations include recommended work that Giglio de Azevedo et al. recommended in their report, but we did not have time to implement this year. Also included are recommendations from our team for future teams moving forward. The recommendations from last year's team that we were able to accomplish were the implementation of a battery detection circuit, local hosting for the Raspberry Pis, and alternative server solutions for the Heroku dashboard in the form of the local hosting. The recommendations that the team did not

reach were the implementation of cloud computing, switching to using 5G, implementation of LiDAR, and the implementation of algorithm switching for the self-driving code.

11.1.1 Machine Learning Approach to Line Detection Via Simulator

One of the strong suites of an algorithmic approach in comparison to a machine learning approach for lane following is its flexibility. An algorithm will work with most compatible tracks without much setup. However, implementing advanced features such as object detection and varying behaviors for said object detection is complex and difficult to implement.

As such, a machine learning approach that keeps the flexibility of an algorithmic approach but does not have the development complexity of one could be an avenue worth pursuing. Namely, developing a tool that simulates a track - and any potential obstacles - and training a machine learning model on it will allow developers to make a flexible and feature-rich system without needing a huge computer science or math background.

11.1.2 Road Sign and Object Detection

One of the key features that our team was hoping to improve upon mPAD was object detection. The goal of this feature was to have the system identify objects in its path to avoid. Eventually this also spawned ideas for mPAD to detect road signs as well, which would direct the car to do different things such as stop the vehicle, change the vehicle's speed, or raise warning flags. It is highly recommended that the HuskyLens be used for this, due to its many object recognition capabilities. As explained in Chapter 9, the HuskyLens was tested for road sign detection, but was never fully implemented.

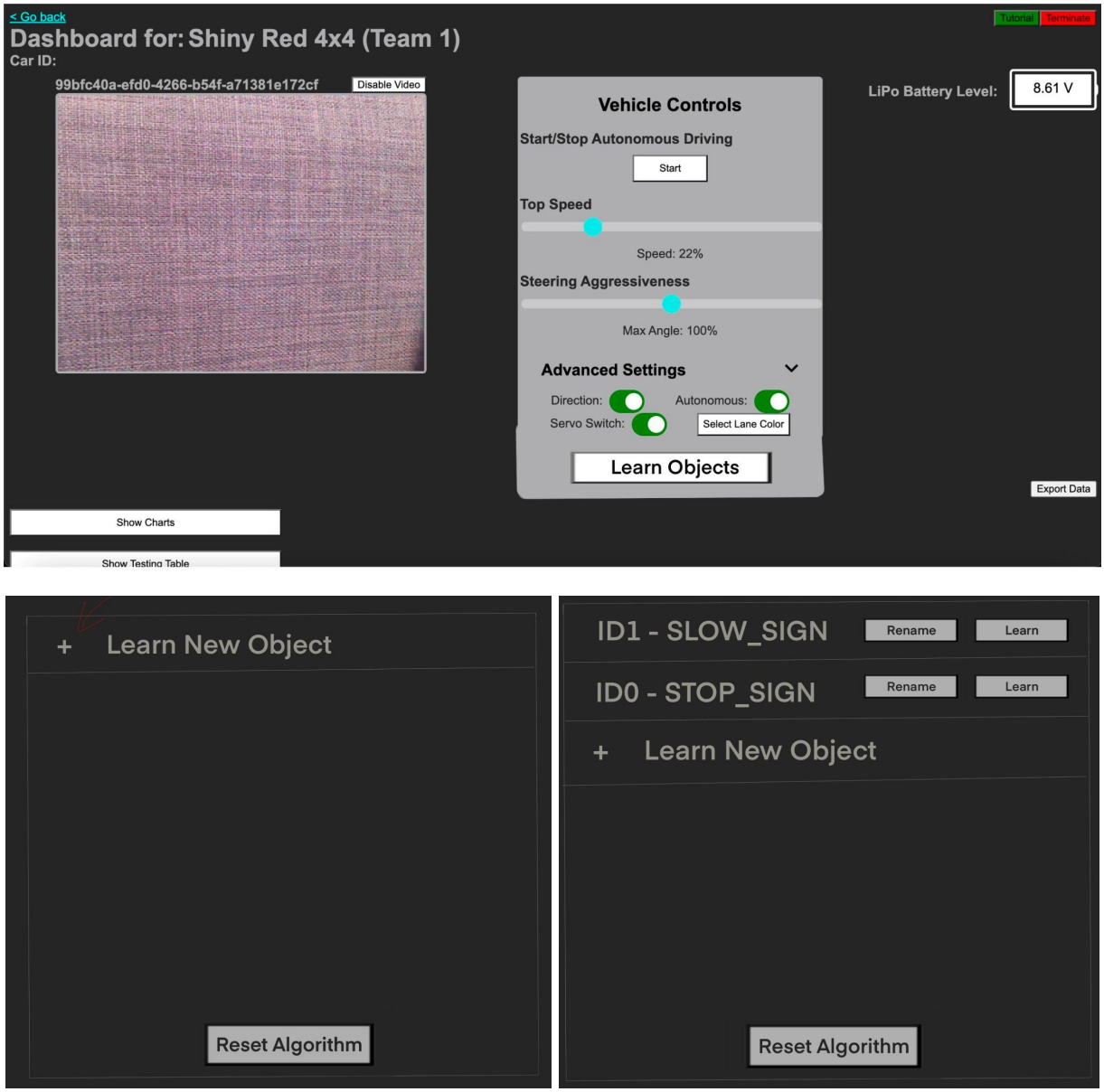

Figure 11.1: HuskyLens Learning Dashboard Page Design

Above is an initial design that was put together to allow users to control the HuskyLens from the dashboard. As you can see, a “Learn Objects” button was added to the regular dashboard page and clicking this button brings the user to a learning page for the HuskyLens. This way the user can learn different custom objects with ease. The “Learn New Object” button will give the user the option to name the object and will create this object in the HuskyLems data. Afterwards, the “Rename” button would call the HuskyLens setCustomName() function and the “Learn” button would call the HuskyLens learn() function to be used to train the device

overtime. The final “Reset Algorithm” button would call the HuskyLens forget() function and would clear the saved data on the HuskyLens.

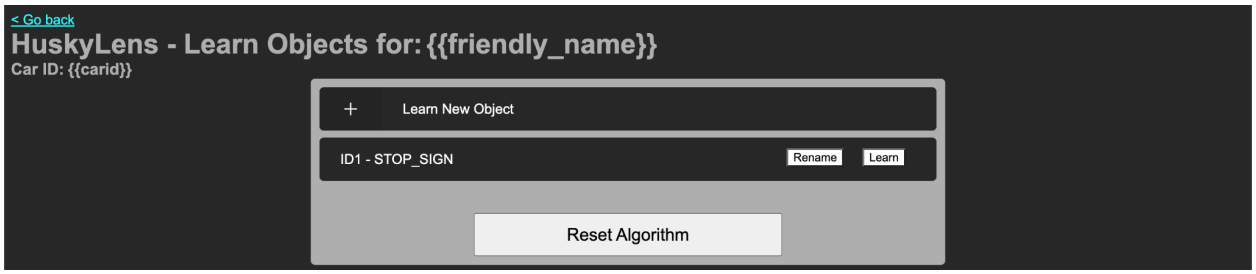

Figure 11.2: HuskyLens Learning Dashboard Page that was built but never implemented

Our team was able to build the following page, but was never officially implemented into the backend of mPAD. There is still code to be written to handle the requests being sent between the server and HuskyLens. The work was left in progress on one of our developer feature branches named “huskylens”.

11.1.3 5-wire Servo Compatibility

The mPAD design is currently only compatible with 3-wire servo motors, due to the nature of the Adafruit Servo Shield. However, there are some RC cars that are instead equipped with 5-wire servos. On some of the RC cars used for mPAD testing, the team had to replace the 5-wire servo that the car came packaged with, for one of our own 3-wire servos. In order to make mPAD truly modular, it would be of best interest for the team to look into a way for making the system compatible with both 3-wire and 5-wire.

Figure 11.3: 5-wire Servo vs 3-wire Servo

11.1.4 Further Performance Improvements

As mentioned in Section 8.4, mPAD V2 currently still has some performance issues related to it despite changes made to the backend of the system to reduce the load of the system on its host computer. As such, further work will need to be done in the area in order to continue to improve the performance and consistency of mPAD. There are a number of potential approaches interested parties can take, which are outlined below.

Move Server Backend to a Remote Hosting Service

One of the first things our team did was move the server from a Heroku webapp to be hosted locally in order to increase the consistency of the platform and decrease the system's reliance on having an internet connection. However, after other issues that were causing consistency issues were fixed, the team realized that moving the server to be hosted on the same hardware as the self-driving logic introduced other performance issues that were difficult to mitigate. As such, with other aspects of the system modified to be more stable, we suggest moving the server portion of mPAD back to a remote hosting service, such as Heroku or even simply a different Raspberry Pi located on the same wireless network as the one that is running the self-driving code.

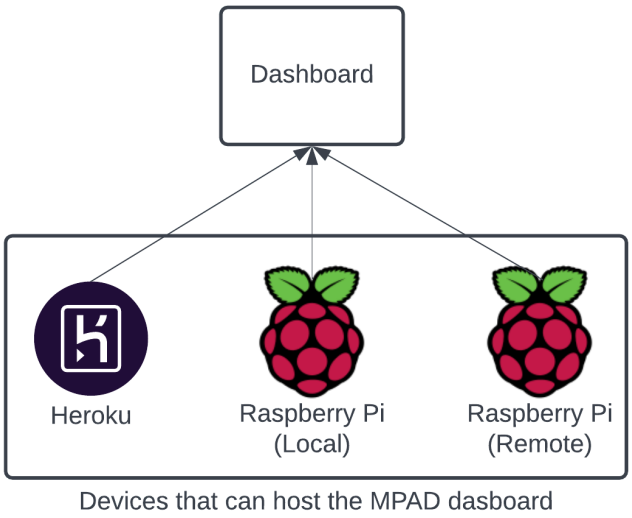

Figure 11.4: Potential Services to Host the Server Portion of mPAD

Combine Server and Self-Driving Backend

Another, much more involved, option to fix the performance issues in the system that are present while running both the server and driving logic portions of mPAD on the same hardware is to revamp the backend and combine the server and self-driving aspect of the car. Currently, the system communicates using GET and POST requests, as well as SocketIO in order to convey commands and data between the server and the self-driving logic. This is much more inefficient than communicating using variables stored in memory. As such, switching over to this system will decrease the amount of computing power used by the system.

Cloud Computing

A third potential option to reduce the strain on the host system is to move the majority of both the server and self-driving backend off of the Raspberry Pi and into the cloud. This would completely resolve any performance issues related to overloading the computing power of the host hardware, as all that would be running on it is a lightweight script to send and receive information from a server. Unfortunately, this would also make the system entirely dependent on having a sufficiently fast connection to the server doing the majority of the processing.

11.1.5 Testing Scalability of mPAD

As mentioned in section 10.2 our team did not have the chance to see if mPAD would operate on a system scaled any larger than a RC car. The team intended to implement mPAD on a toddler size vehicle and run it on a larger track that could handle the increase in size of the vehicle. One track that was considered for testing was the track around WPI's football field. We wanted to learn if the self driving code would work on a larger vehicle. Continued scaling and testing should be done to see how large a vehicle mPAD could accommodate.

11.1.6 Multiple Vehicles Driving Around the Track with Collision Detection

Finally, we recommend that future teams implement a system where multiple vehicles can drive around the track autonomously while successfully being able to sense one another on the track and avoid collisions. Currently, object detection is done via seven ultrasonic sensors connected to a bumper; refer to section 2.1. Over the course of this project additional object

detection avenues were explored in the form of the PixyCam, HuskyLens, and RPLiDAR. However, no proper collision detection was implemented. This was a result of unforeseen issues with implementing other areas of our project, skewing our projected timeline for project completion, however some initial research and testing was still achieved as seen in Sections 3.2 and 9 of this report. Our team recommends that this system be explored either using some form of LiDAR or another alternative option.

11.2 Project Experience

Throughout the development of the mPAD V2, we learnt several lessons as a team and as individuals. Collectively, we came in with a wealth of individual knowledge, and over the course of the term we learned how to leverage each other's skills to make the team more than the sum of its parts. The following sections detail our individual backgrounds before working together, the skills we have gained through the project and overall reflections of the project experience.

11.2.1 Bryan Lima

During A Term 2021 I took CS4241: Webware: Computational Technology for Network Information Systems. Throughout this course, I was able to learn a lot about web design and how to utilize HTML, JavaScript, and CSS. This allowed me to add new features to our web dashboard and give the mPAD user more creative freedom in design. In C Term, I also took CS4731: Computer Graphics, which also helped further my HTML and JavaScript knowledge.

I had little to no Python background coming into this project and relied heavily on some of my teammates' assistance in order to implement certain features. However, working on mPAD and working alongside my teammates helped me improve my programming in Python. I learned how to write different functions in Python and was able to build features on my own towards the tail end of our project. This project and team were awesome to be a part of. It felt that it was more of a hobby to tinker with than an assignment to be working on.

11.2.2 Anthony LoPresti

This project was a great learning experience for me. I was a member of both sides of this project, the hardware and the software and there was a lot to learn from both sides. On the hardware side of things I learned to collaborate properly on a large scale CAD assembly. Also I

learned to work on a large scale additive manufacturing project. I learned a ton about 3D printing and 3D modeling that will surely help me tremendously further on in my career. The software side of the project was a lot of problem solving. Early on it was a lot of bug fixing both code and hardware issues. I also had the opportunity to learn more about LiDAR and other obstacle detection technology. Another large aspect of the software side of the project was learning to communicate engineering work in a simple manner that is easy to understand and follow. Overall I feel like I have developed a lot more as an engineer and I hope to put my skills to the test in the future.

11.2.3 Thomas Riviere

As an Electrical and Computer Engineering major, I was able to understand the electrical layout of mPAD with ease and used this as an asset for my team, while they put their efforts into more of the programming side of things. My previous experience with Arduino surely helped to ensure that our sensor package was finely tuned. My graphic design skills became very handy when putting together the setup guide. I would say however, due to COVID, my hands-on experience with ECE has been hindered up until this point. While working on this project, I learned how to solder on small devices, and how to debug hardware that was notdoing what was intended. I also learned a lot of new things that I never really knew that I would need, even as simple as using zip ties and velcro to clean up wires.

11.2.4 Lindberg Simpson

This project was an amazing opportunity for me to apply my practical experience in Computer Science to work in an interdisciplinary team on a physical project. Through previous projects and classes, I had experience with many aspects of the software side of this project. I was already familiar with many of the programming languages used in the previous iteration of mPAD such as Python, HTML, javascript and CSS. However, I had very limited experience working with Linux-based operating systems, command line interfaces (CLIs), electrical systems and sensors. Throughout the course of developing mPAD V2, I was able to employ the skills I already possessed to help the team in various parts of the software analysis and development phases of the project. I also became more familiar using the CLI, working with a team in GitHub and gained a better understanding of how sensors work with physical, electrical systems. I wish

I had more experience in Electrical and Computer Engineering before doing this project so that I could offer more help on the hardware side as well. In general, this project was an incredible learning experience for me and a great capstone project to conclude my 4 years at WPI.

11.2.5 William Yang

Coming into this project, most of my CS background lay in software application development and the Unix terminal, which I utilized extensively while working on the dashboard and Raspberry Pi's respectively. However, I learned a lot about working with SocketIO and servers in Python, which is something I had very little experience with before working on the project. I definitely wish I had known about these topics before the project, as I was the main contributor towards the server backend and had to learn everything on short notice. Overall, though, I believe the project was a positive experience for me, as I learned a lot from it.

12 References

1. Alex. (2015, Oct 11). “Everything You Need to Know about Lipo Battery Chargers.” *DroneTrest*. Unmanned Tech. https://www.dronetrest.com/t/everything-you-need-to-know-about-lipo-battery-chargers/1326
2. Angetube Streaming 1080P HD Webcam Built in Adjustable Ring Light and Mic. Advanced autofocus AF Web Camera for Google Meet Xbox Gamer Facebook YouTube Streamer. (n.d.). Amazon.com. https://www.amazon.com/gp/product/B07RXYG295/ref=ppx_yo_dt_b_search_asin_title?ie=UTF8&psc=1
3. AWS DeepRacer. (2022). Amazon Web Services, Inc. https://aws.amazon.com/deepracer/
4. *Connect to wpi wireless using a raspberry pi with gui*. (2021, September 22). The WPI Hub. https://hub.wpi.edu
5. Charmedlabs/pixy2. (n.d.). GitHub. Retrieved May 1, 2022, from https://github.com/charmedlabs/pixy2
6. Donkeycar: a python self driving library. (n.d.). GitHub. Retrieved May 1, 2022, from https://github.com/autorope/donkeycar
7. Enable i2c interface on the raspberry pi. (2014, November 2). *Raspberry Pi Spy*. https://www.raspberrypi-spy.co.uk/2014/11/enabling-the-i2c-interface-on-the-raspberry-pi/
8. Giglio de Azevedo, E. et al. (2021). Modular Package for Autonomous Driving (MPAD). : Worcester Polytechnic Institute., https://digital.wpi.edu/concern/student_works/mp48sg60t?locale=en
9. Gus. “A Simple Arduino Battery Tester.” (2022, Jan 31). *Pi My Life Up*. ELITE CAFEMEDIA. https://pimylifeup.com/arduino-battery-tester/
10. Huang, T. (n.d.). *RPLIDAR-A1 360°Laser Range Scanner _ domestic laser range scanner*. SLAMTEC. https://www.slamtec.com/en/Lidar/A1
11. Introducing Raspberry Pi Imager, our new imaging utility. (2020, March 5). *Raspberry Pi*. https://www.raspberrypi.com/news/raspberry-pi-imager-imaging-utility/

12. Kim, T. et al. (2020). Modular Self-Driving and Sensor Packages for R/C Cars. : Worcester Polytechnic Institute., https://digital.wpi.edu/concern/student_works/bk128d46c?locale=en
13. Mehrabani, Afshin. “User Onboarding and Product Walkthrough Library | Intro.Js.” *Introjs.Com*, https://introjs.com/.
14. *OSOYOO Robot car kit Lesson 1: Basic Robot car. (2018, December 7). osoyoo.com. https://osoyoo.com/2018/12/07/new-smart-car-lesson1/*
15. *PIXY2 for Lego*. PixyCam. (n.d.). https://pixycam.com/pixy2-lego/
16. Prast, Robert. (2020, Aug 4). “Huskylens/HUSKYLENSPython.” *GitHub*, Github, Inc. https://github.com/HuskyLens/HUSKYLENSPython
17. SAE Levels of Driving Automation™ Refined for Clarity and International Audience. (2021, May 3). SAE International. https://www.sae.org/blog/sae-j3016-update
18. VCANNY Remote Control Car, Terrain RC Cars, Electric Remote Control Off Road Monster Truck, 1: 18 Scale 2.4Ghz Radio 4WD Fast 30+ mph RC Car, with 2 Rechargeable Batteries. (n.d.). Amazon.com. https://www.amazon.com/dp/B07H4KCP6M?ref=ppx_pop_mob_ap_share

13 Appendix

13.1 Hardware Cost and Specifications

Component	**Cost**	**Specifications**
ELEGOO MEGA R3 Board ATmega 2560	$22.99	https://www.elegoo.com/products/elegoo-mega-2560-r3-board
Raspberry Pi 4	$75 - out of stock $192.89 - inflated price on amazon	https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/
HobbyWing 1060 Electronic Speed Controller (ESC)	$27.83	https://www.hobbywingdirect.com/products/quicrun-10-esc-2-3s-brushed
HiLetgo PCA9685 Servo Shield	$9.19	https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf
HC-SR04 Ultrasonic Sensor	5 sensors for $8.79	https://www.electroschematics.com/hc-sr04-datasheet/
DHT11 Temperature Sensor	2 sensors for $6.69	https://components101.com/sensors/dht11-temperature-sensor
BNO055 IMU	Out of Stock - older technology, so can be replaced with a newer model in the future Similar product cost $35-40	https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor?gclid=CjwKCAiA1JGRBhBSEiwAxXblwc3yA1iUJsOr1YVBK7-6wKu3kx9x0iSY7Zr-2ZmN4MlQVvCb

		dcYDHRoC2gMQAvD_BwE
KY-003 Hall Effect Sensor	10 sensors for $5.99	https://arduinomodules.info/ky-003-hall-magnetic-sensor-module/
USB Camera	$59.85 - price may vary	Only important spec is wide angle lens, which is extremely common in almost every purchasable USB camera
HuskyLens	$54.90	https://shop.pimoroni.com/products/huskylens-an-easy-to-use-ai-machine-vision-sensor?variant=31603336282195
RPLidar	$95	https://www.seeedstudio.com/RPLiDAR-A1M8-360-Degree-Laser-Scanner-Kit-12M-Range.html
Total Cost of mPAD V2 (Cost did not include RPLiDAR and the pre COVID-19 pricing of the Raspberry Pi 4)	$325	

13.2 mPAD V2 Setup Guide

https://docs.google.com/presentation/d/1rBzo-pQz23uOcFlPZBSbfsVc6GUMW2yvpQrBV4xP6dw/edit?usp=sharing

13.3 Manual Driving Python Code Snippet

```
       if not streamer.drive:
   # code to manual drive
   manual_driving = False
   # if up arrow is depressed
   if UP_ARROW:
       kit.continuous_servo[MOTOR_CHANNEL].throttle = streamer.direction *
(max_speed*(speed/100))
       manual_driving = True

   # if down arrow is depressed
   if DOWN_ARROW:
       kit.continuous_servo[MOTOR_CHANNEL].throttle = -(streamer.direction *
(max_speed*(speed/100)))
       manual_driving = True

   # if left arrow is depressed
   if LEFT_ARROW:
       if streamer.servo_direction == 1:
           kit.servo[SERVO_CHANNEL].angle = 150 + servo_adjustment
           manual_driving = True
       if streamer.servo_direction == -1:
           kit.servo[SERVO_CHANNEL].angle = 30 + servo_adjustment
           manual_driving = True

   # if right arrow is depressed
   if RIGHT_ARROW:
       if streamer.servo_direction == 1:
           kit.servo[SERVO_CHANNEL].angle = 30 + servo_adjustment
           manual_driving = True
       if streamer.servo_direction == -1:
           kit.servo[SERVO_CHANNEL].angle = 150 + servo_adjustment
           manual_driving = True

   # if no forward or backward
   if not UP_ARROW and not DOWN_ARROW:
```

```
        kit.continuous_servo[MOTOR_CHANNEL].throttle = 0

    # if no left or right
    if not LEFT_ARROW and not RIGHT_ARROW:
        kit.servo[SERVO_CHANNEL].angle = 90 + servo_adjustment
```

13.4 Self-Driving Videos

13.4.1 mPAD (Modular Package for Autonomous Driving) - Testing Compilation

The following Youtube link shows a video demonstrating various parts of the self-driving testing process:

https://youtu.be/XrY1tMupUrU

13.4.2 Early-Stage Self-Driving Testing Video

The following link holds videos that showcase early-stage testing of the self-driving platform. Early-stage testing consists of self-driving tests done on an RC car equipped with an USB camera before any software changes were made to the mPAD package.

https://drive.google.com/drive/folders/1DdrZyGnDJGI-ZZxaTP2ZL28UminwRd0M?usp=sharing

13.4.3 Late-Stage Self-Driving Testing Video

The following link holds videos that showcase late-stage testing of the self-driving platform. Late-stage testing consists of self-driving tests done on an RC car equipped with an USB camera after software changes regarding lane detection and self-driving were made to the mPAD package.

https://drive.google.com/drive/folders/1Z0uKUk7WZND330wvKSbkOtPurIibDq5-?usp=sharing

13.4.4 Testing mPAD on Other Cars

The following link holds videos that showcase testing the mPAD platform after the aforementioned software changes were made on new RC cars that mPAD had yet to be run on.

These tests were performed to ensure that mPAD worked correctly on hardware that the team wasn't testing on during development.
https://drive.google.com/drive/folders/1qUCKzSicKKEYiIqXChFBFWSqXUeO9zE1?usp=sharing

Copyright Information