Modular Package for Autonomous Driving V3

A *Major Qualifying Project*

Submitted to the Faculty of Worcester Polytechnic Institute

In fulfillment of the requirements for the degree in

Bachelor of Science

By

Rohan Anand (CS/RBE)

Martin Bleakley (RBE)

Allison Colón-Heyliger (ME)

Ian Khung (CS)

Ishan Rathi (RBE/CS)

Advised by

Professor Kaveh Pahlavan (ECE/CS)

Professor Pradeep Radhakrishnan(ME/RBE)

April 27, 2023

Abstract

The primary goal of this project is to create a modular package that can be seamlessly integrated into a wide range of vehicles, enhancing their functionality and promoting the broader adoption of autonomous driving technologies. As the base of our autonomous driving packages, cars from previous MQPs were used. However, many of them had mechanical failures, which needed to be addressed. Fixing these cars was very much needed for the team for this project to be successful and efficient.

The open-source Donkey Car platform was utilized as a foundation for developing the Modular Package for Autonomous Driving (mPAD) system to accomplish this objective. Donkey Car allows an RC car to control using PWM signals to send commands to the throttle and servo (steering) components. This platform collects training images alongside corresponding throttle and steering values at specific points in time, creating a comprehensive dataset for further analysis and model development. A linear regression model was created based on the collected data, which can be uploaded back to the vehicle to enable autonomous driving capabilities.

Furthermore, object detection was incorporated into the system to detect stop signs and other relevant objects in the environment, enhancing the safety and reliability of the autonomous driving system. Utilizing the Donkey Car platform had some limitations, specifically in the dashboard/UI. To modernize the looks and extend the capabilities, a new UI was generated, which included the ability to graph sensor data, throttle control using a slider, and a new color palette. These changes allowed for a solution that can be reliably driven autonomously for five laps after manual driving, with object detection and the ability to collect sensor data. The modular nature of the mPAD system offers the potential for widespread integration into various vehicle types, paving the way for the future of autonomous driving.

Acknowledgments

The authors of this paper would like to express their sincere gratitude to Worcester Polytechnic Institute (WPI) for providing the resources and support necessary to complete this research.

The authors would start by thanking Peter Hefti, who helped organize lab equipment and provided us with the guidance to utilize these tools. The authors would like to thank Barbra Furhman, who helped us purchase equipment when necessary. The authors thank Professor Kaveh Pahlavan of the Electrical and Computer Engineering and Computer Science Department for his guidance, support, and invaluable insights throughout this research. The authors also thank Professor Pradeep Radhakrishnan of the Mechanical Engineering and Robotics Engineering Department for his valuable feedback and support throughout the research.

Authorship

Chapter	Writer(s)
Abstract	IR, RA
Acknowledgments	IR, RA
Authorship	IR, RA
Introduction	RA
Previous MQPs	IR, IK, RA, AC
Project Goals	RA
Background	IR, RA
Methods	MB, IK, AC
Mechanical Changes	AC
Test Track	RA
Self-Driving Implementation	IR, IK, RA, MB
Object Detection	IR
Dashboard Changes	IK, RA
Testing	RA, IR, IK, AC
Discussion	IR, RA
Conclusion	IR, RA, MB, AC, IK

The writers of each section are displayed with their respective initials; Rohan Anand (RA), Martin Bleakley (MB), Allison Colón-Heyliger (AC), Ian Khung (IK), and Ishan Rathi (IR).

Table of Contents

List of Figures

1. Introduction

In the twenty-first century, autonomous driving has been a prominent technology. As stated by BCC, “Self-driving vehicles are steadily becoming a reality despite the many hurdles still to be overcome – and they could change our world in some unexpected ways [11].” This has been and will continue to be a significant driver of development and innovation throughout history. To attain a future where all cars drive independently, several firms are constantly redefining what autonomous driving is and how it works. These companies include Tesla, Comma AI, and many more. All the little things taken for granted, like adaptive cruise control and lane departure alerts, are key elements of autonomous driving. These are counted as Level 2 Autonomous functionality set by the SAE [12]. Numerous businesses have diverse ideas on how to approach self-driving vehicles overall and provide original solutions that assist the advancement of this technology.

Many automakers are now working on developing many completely autonomous vehicles. The most common one to point out is Tesla. Tesla offers the most cutting-edge autonomous driving technologies in its range of publicly accessible vehicles. As mentioned on their website, it has advanced safety and convenience features like Autopilot which are created to help users with the most difficult aspects of driving [13]. Over time, Autopilot adds new capabilities and enhances existing functionality to make Tesla more capable and safe. Their "Full Self Driving" function enables the vehicle to travel between locations with little to no interaction from the occupants. Remember that this function is exclusive to Tesla cars, rivaling other automakers developing comparable programs. A startup called Comma AI is producing an innovative self-driving product. As told on their website, Comma AI aims to offer a solution to improve people's driving experiences on vehicles they own [14]. They aim to incorporate attention-monitoring technology and driver assistance capabilities like automated highway driving. Comma AI is utilizing already-built cars and reusing them with new technology to drive independently, in contrast to Tesla's approach. While these developments represent a significant step toward the future, they still need to be perfected.

Scalability is a critical consideration in the development and deployment of autonomous cars. As the number of autonomous vehicles on the road increases, the technology must be able to scale to accommodate the growing demand. Scalability in the context of autonomous cars refers to the technology's ability to handle increased complexity, data processing, and communication requirements as more vehicles are added to the system. One way to achieve scalability in autonomous cars is through modular design, which allows for easy integration of new components and features. This modular approach can help reduce costs by using off-the-shelf components, which can be easily replaced or upgraded.

Another important aspect of scalability in autonomous cars is the ability to handle large amounts of data generated by sensors, cameras, and other onboard systems. To achieve this, autonomous car systems often rely on powerful computing systems, including cloud-based processing and edge computing technologies. These systems can scale to handle the increased data processing requirements as more autonomous vehicles are added. Additionally, the communication infrastructure must be scalable; as the number of autonomous vehicles increases, the communication network must handle the increased volume of data generated by each vehicle while maintaining a fast and reliable connection. Scalability is critical for successfully deploying autonomous cars and must be carefully considered in designing these systems.

The mPAD V3 is an excellent example of a product that prioritizes scalability in autonomous vehicle technology. Its objective is to provide a reliable and affordable device that can be mounted on a remote-controlled vehicle and ultimately autonomously drive. The product's modularity is thanks to its robustness and scalability, which is rarely seen in an autonomous driving package. This modularity will make the product less expensive for customers and enable even novice users to install a straightforward package for their cars.

The mPAD V3's modularity is essential to its scalability, as it allows the product to handle increased complexity and data processing requirements as more autonomous vehicles are added to the package. This modular design enables easy integration of new components and features, which helps reduce costs by using off-the-shelf components that can be easily replaced or upgraded as needed. As a result, the mPAD V3 is an affordable and scalable solution for autonomous driving, making it an excellent choice for those looking to invest in this rapidly growing technology. Implementing the mPAD system can be initiated at a small scale by applying it to basic remote-controlled (R/C) cars, gradually advancing to larger motor vehicles, such as golf carts, to expand its scope and test its efficacy.

The mPAD project has three main objectives. The first objective is to improve the reliability of the self-driving algorithm to enable the RC car to navigate a track with two established outer lanes and stay between the lanes at all times. The second objective is to develop an object detection program that allows the RC car to detect stop signs and bring the car to a complete stop at the sign's location. The third objective is to enhance the user experience of the autonomous driving program's dashboard by creating a more intuitive and visually pleasing interface. The final one is to create a modular and scalable package for users.

The report is structured to facilitate comprehension for readers with varying levels of technical knowledge. Chapter 2 provides an overview of prior work accomplished by previous MQP teams and their contributions to the current year's team. Chapter 3 outlines the project goals for mPAD V3 and expounds on the corresponding objectives and requirements necessary to achieve these aims. Chapter 4 provides background information on autonomous driving, including an exploration of the various algorithms currently available. Chapter 5 presents the methodology utilized by the team throughout the different terms. Chapter 6 delves into the mechanical changes that were implemented this year. Chapter 7 details the modifications made to the track throughout the project. Chapter 8 centers on the team's autonomous driving implementation and the comparison between the Husky rover approach and the use of Donkey Car. Chapter 9 focuses on the team's discussion of object detection, particularly regarding Stop Signs. Chapter 10 highlights the dashboard changes made to the project. Chapter 11 analyzes the team's progress in meeting the expectations outlined in the project goals. Finally, Chapter 12 provides a comprehensive conclusion to the project.

2. Previous Work

This section talks about the previous work teams for this project have done, the work handed down to this year's team, and the obstacles initially faced when starting this project.

2.1 Donkey Car (19-20')

DonkeyCar was the first MQP in the series of mPAD [15]. During this MQP, the students utilized the Donkey Car open-source platform to create autonomous vehicles [18]. They conducted multiple tests on different tracks to improve the neural network models and self-driving performance. The team used three different types of cars to develop their modular self-driving kits. The first was the commercially available Mission D car (Figure 2.1.1), used for initial testing to learn how the Donkey Car platform and Neural Network implementations work on a scale car.

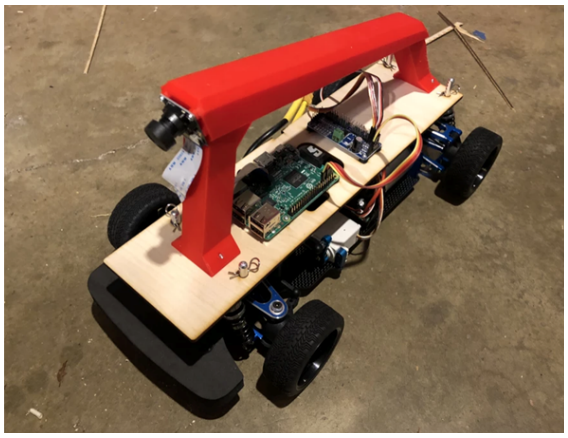


Figure 2.1.1: Picture of mission D car, reproduced as is from [15]

The second type of car used was student-made cars from a previous Advanced Engineering Design course (ME 4320) at WPI, which were not commercially manufactured and provided an opportunity to test the self-driving system on different types of vehicles (Figure 2.1.2).

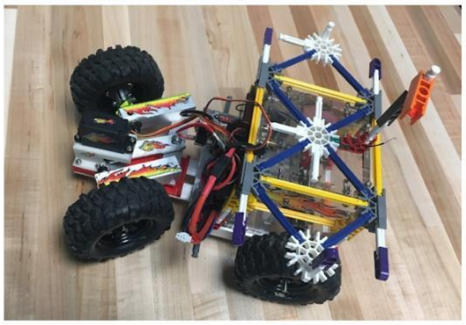


Figure 2.1.2: A student's advanced engineering design car, reproduced as is from [15]

Finally, the team also tested their package on 3D-printed cars made by a sister MQP team, which were also intended for use in ME 4320 and ME 4322. By testing their package on these different types of cars, the team ensured that their kit would apply to various vehicles (Figure 2.1.3).

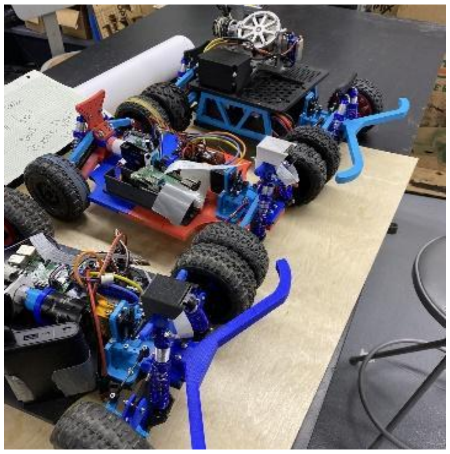


Figure 2.1.3: Example 3D printed scale cars, reproduced as is from [15]

With the goal of these cars being used by mechanical students in a classroom setting, a user guide was created to quickly implement the kit, which was tested and improved with feedback from two students. A large part of this classroom was the sensor data analysis, so the authors additionally created a modular sensor kit to record valuable data about the function of scale cars. The sensor kit developed by the team incorporates a sensor package that includes several sensors to provide data on various aspects of the car's performance. These sensors include a Hall effect sensor that measures the car's velocity, thermal sensors to monitor component temperatures, and an IMU to record the car's roll, pitch, and yaw. The sensor data is recorded using an Arduino Mega and a Raspberry Pi, which saves the data to a CSV file for later analysis. The sensors are housed in an exterior casing, designed with mounting holes and an external portable battery, enabling independent power and increased modularity. The sensors are also designed to be easily mounted and relocated, allowing users to change the sensor locations during testing. This sensor package is an independent unit that can be mounted on any car with minimal space requirements, making it a versatile and convenient tool for collecting data during self-driving experiments. The sensor kit was integrated into the cars to record speed, rotations, and temperatures. Like the self-driving package, the authors used feedback to test and improve the sensor kit's user guide (as shown below in Figure 2.1.4).

The team collected training data by manually driving the RC car around the track. At the same time, the Raspberry Pi saved pictures from the front camera along with the motor and servo levels that controlled the car's speed and steering angle. The collected data was used to train a neural network model. When running on a car, processed images from the camera were used by the model to predict the appropriate speed and steering angle to enable the car to self-drive. To optimize the neural network's performance, the team followed a procedure to collect training data and improve it with each test. They identified four different driving styles to collect data: accurate laps, small oscillation laps, extreme laps, and fast laps. The different driving styles impacted the overall results by teaching the car to stay in the center of the track, learn to correct itself, and drive back to the center if it went off course and go around the track faster.

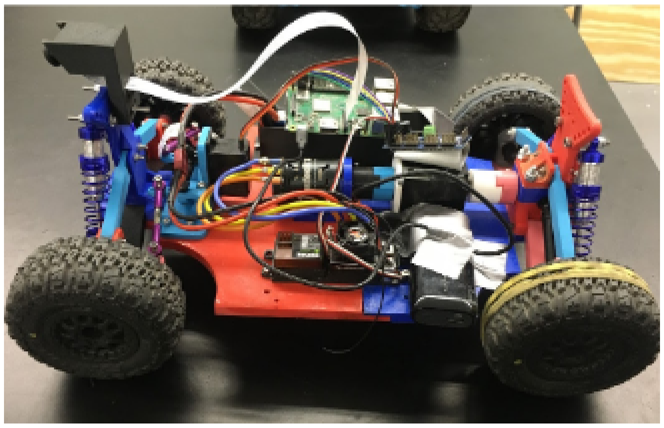


Figure 2.1.4: Donkey Car setup during 2019-2020 mPAD, reproduced as is from [15]

DonkeyCar has some issues that need to be addressed for future work. An overview of the car can be seen in Figure 2.1.4. One of the issues is that the neural network models used in the kit are non-transferable, meaning they only work on the car and track they were trained on. This presents a challenge for students needing to train their cars rather than simply installing the kit. Another issue is that the scale car's responsiveness to turns decreases after four or five laps, possibly due to the servo's torque decreasing with continuous use. More precise sensors can be installed to improve the project, and the code package can be compressed into smaller printed circuit boards for more modular applications. Calibration tests can be performed with these new sensors to ensure better precision and accuracy. The user manual can be improved to include more detailed analysis and instructions for adding new sensors.

2.2 mPAD Version 1 (20-21’)

mPAD V1 was the first MQP with an in-house solution for self-driving[16]. The package was developed with the Raspberry Pi 4 8GB and Elegoo Mega 2560 development boards and features a sensor package that uses a Raspberry Pi camera for lane recognition and seven ultrasonic sensors for obstacle avoidance, along with an inertial measuring unit (IMU), two temperature sensors, a hall-effect sensor, and a battery sensing circuit. All sensors went through rigorous accuracy testing to ensure reliable data collection.

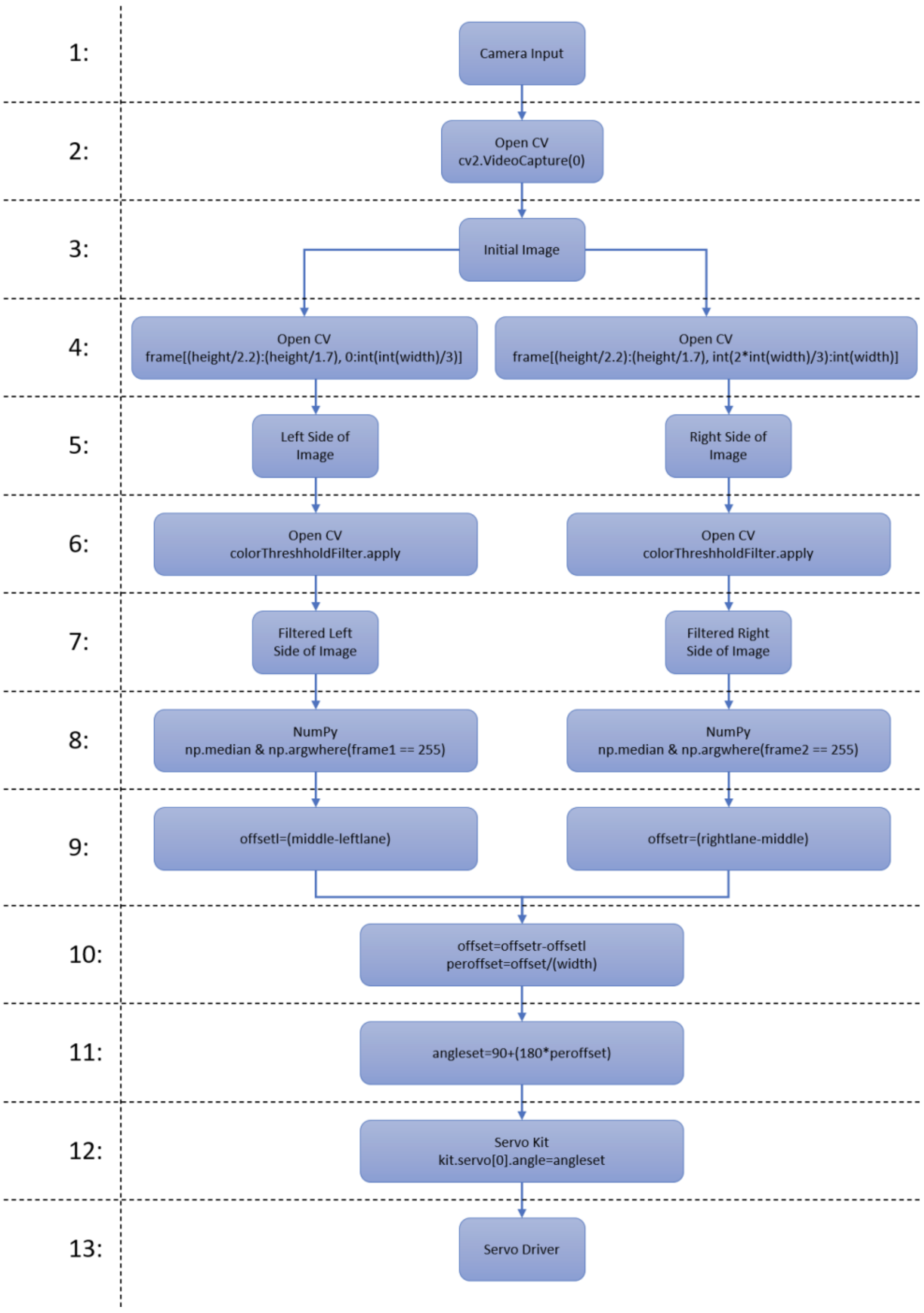

1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
Camera Input
Open CV
cv2.VideoCapture(0)
Initial Image
Open CV
frame[(height/2.2):(height/1.7), 0:int(int(width)/3)]
Open CV
frame[(height/2.2):(height/1.7), int(2*int(width)/3):int(width)]
Left Side of Image
Right Side of Image
Open CV
colorThreshholdFilter.apply
Open CV
colorThreshholdFilter.apply
Filtered Left Side of Image
Filtered Right Side of Image
NumPy
np.median & np.argwhere(frame1 == 255)
NumPy
np.median & np.argwhere(frame2 == 255)
offsetl=(middle-leftlane)
offsetr=(rightlane-middle)
offset=offsetr-offsetl
peroffset=offset/(width)
angleset=90+(180*peroffset)
Servo Kit
kit.servo[0].angle=angleset
Servo Driver

Figure 2.2.1: Block Diagram indicating mPAD AI Flow, Reproduced as is from [16]

They accomplished this by capturing and processing images from a physical camera attached to a Raspberry Pi (Figure 2.2.1). The captured image is then split, cropped, and subjected to a color filter to isolate the color matching the road's lanes. The algorithm then analyzes the bottom half of the image to determine which side of the screen has the minor lane color, and the servo is set to steer toward that direction. This process is looped hundreds of times a minute to achieve autonomous driving in a way that minimally taxes the Raspberry Pi's CPU. In addition, the self-driving algorithm is controllable from a web dashboard.

The self-driving software uses computer vision to navigate a course without any prior training, with the angles of the wheels calculated using the coordinates of the pixels in the camera input and readings from the ultrasonic sensors. The package is designed to be easily transferable from one RC car to another. All data collected can be viewed in real-time on a customizable web portal, making it highly configurable and easily testable.

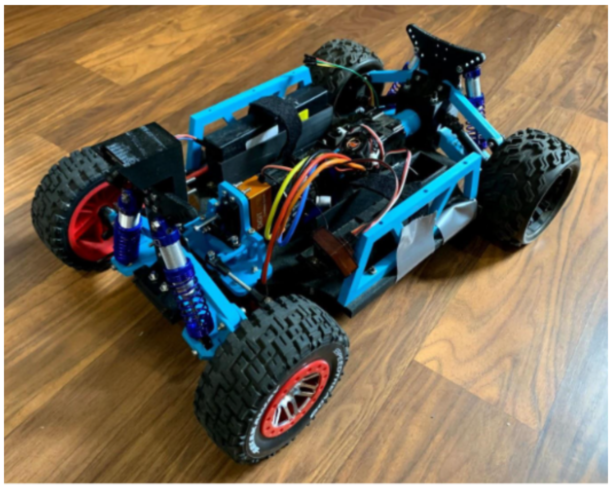


Figure 2.2.2: RC Car during mPAD V1, reproduced as is from [16]

Implemented in the Introduction to Engineering Design (ME2300) course at Worcester Polytechnic Institute (WPI), eight teams of mechanical engineering students received documentation in the form of two guides to create their version of mPAD V1. The integration was successful, and the students could fully utilize mPAD V1's capabilities with their RC car designs (Figure 2.2.2).

To demonstrate the objective, the mPAD V1 MQP team conducted testing. Testing aimed to test the autonomous driving capability of an RC car on multiple tracks using the complete modular package. Testing indicated that the RC car could complete the track reliably up to a speed of roughly 1.1 meters per second. Different speeds were tested, but higher speeds resulted in the car going off the track in the sharp turn found in the upper right corner of the track. Latency and sharp turns were the primary factors affecting the car's performance. The runtime of the driving code was efficient, with an average time of 35 ms over thousands of loops. The driving ran at 28.5 fps, and the videos of the car autonomously navigating the course demonstrated its performance in completing the lap, oscillating straightaway, and staying in the middle of the lanes. Overall, testing provided valuable insights into the autonomous driving capability of the RC car, and the results suggest further latency and obstacle avoidance improvements enhance the car's performance.

Overall, mPAD V1 is an efficient and low-cost solution for prospective engineers to learn real-world technical skills in autonomous driving. Its modular design, accurate sensor package, and user-friendly web portal provide an accessible way for students to bring their designs to life and test their mechanical integrity while avoiding human driving bias.

2.3 mPAD Version 2 (22-23')

mPAD V2 was the updated version of mPAD that was produced during the 2021-2022 MQP. Many changes were made to both the hardware and the software. Each portion is critical to the development of V3 of mPAD and to being able to create a finished working product.

2.3.1 Software Failures

The software of mPAD was updated to have a directory to store the dashboard and server code. This allows the folders to be separate from the directory that stores the self-driving program. This was due to the two directories relying on different dependencies. However, this team created a main script for making two scripts where each script downloaded the necessary packages and Python libraries to install each directory fully. The team also implemented both object detection and made changes to the dashboard.

2.3.1.1 Autonomous Driving Algorithm

This MQP team aimed to determine if the system had consistent self-driving capabilities by testing if it could autonomously drive for at least five laps around a test track. They used a commercial RC car purchased from Amazon equipped with the mPAD system (Figure 2.3.1..1), which did not have the sensor package attached to it during most of the testing, to attribute issues to the camera and self-driving aspects of the system.

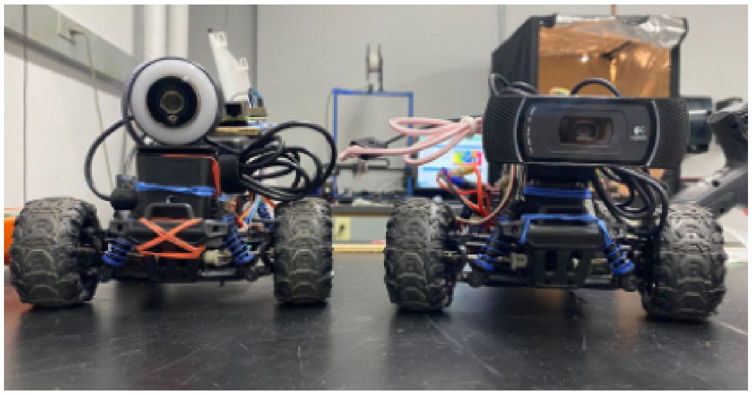


Figure 2.3.1..1.1: mPAD V2 cars, reproduced as is from [17]

Despite hardware optimizations, the system could only complete up to a few laps at a time. To further optimize the system, the team examined and adjusted the self-driving logic of mPAD, checking the code for logical errors and comparing a car's behavior in a situation to the expected behavior of the self-driving logic.

Unexpected behavior suggested that the problem lay in the car steering or color filtering logic. They found that the values used to filter the color of the lanes appeared to be hard coded, and the variable that held the actual color values obtained from the dashboard should have been used in the driving logic. By switching the hardcoded values to the variable set by the dashboard, they observed that the car behaved as expected initially but later behaved as if it did not see the track once it left the general vicinity of its original position on the track. They realized that the problem stemmed from differences in the lighting on different parts of the track.

2.3.1.2 Object Detection Errors

Although the team attempted to implement object detection using a camera to return the size and coordinates of detected objects, the system was not fully functional. Object detection faced challenges in accurately identifying stop signs, as it would often detect other red objects and mistakenly classify them as stop signs. Despite these limitations, this preliminary effort was a valuable foundation for further developing and refining the object detection system.

2.3.1.3 Dashboard Issues

Most of the changes on the dashboard were mainly frontend. This includes servo angle adjustment, manual driving, test scripts for the servo and motor throttle, and displaying the LiPo battery voltage. The dashboard also had graphs for the data collected by Hall-effect, IMU, ultrasonics, battery percentage, humidity, and temperature sensors. A dashboard tutorial was also added to make it easier for the user. This version of the project was the starting point of this year's MQP and gave the team a few problems when getting started. The team noticed several issues with the dashboard made it difficult for the car to run autonomously and manually. When trying to get the RC car moving with manual driving done through the arrows on the keyboard, the car was not very responsive to the inputs from the keystrokes. For example, when the left key was pressed, the car would take a few seconds to turn the wheels to the right, and when another key was pressed, it would do the same. Other times the car would not even react to these keystrokes and would stay as it previously was. The team was also unable to run the autonomous driving method as it failed to recognize the lines of the track. This may have been connected to another problem the team identified, the camera. Sometimes, when loading the dashboard, the camera display was very inconsistent. There were times when the camera would load, the car was switched into manual driving, and after a few keystrokes, the camera display would freeze. The issue was determined to be due to a poor connection between the server and the car.

2.3.2 UI User Testing

After analyzing the mPAD Self-Driving System Feedback Survey Results [25], the team found the mPAD User Review Section to be the most crucial aspect in improving the dashboard user experience. The mPAD Self-Driving System Feedback Survey was conducted by the mPAD V2 team to assess user experiences and inform future improvements to the platform. The survey was divided into four sections: User Info and Demographics, Background Knowledge, mPAD User Review, and Concluding Questions. A total of 26 students, primarily from the mechanical engineering (ME) and robotics engineering (RBE) fields, participated in the survey.
Results from the User Info and Demographics section indicated that most respondents had prior experience in relevant ME and RBE courses.

The Background Knowledge section revealed that most respondents were familiar with Arduino, temperature sensors, and brush/servo motors. In the mPAD User Review section, the participants found the mPAD Setup Guide generally easy to follow and expressed confidence in setting up the system using the guide. However, they also suggested improvements, such as clarifying wiring diagrams, adding safety guidance for LiPo batteries, and modifying the text font color for better readability. The dashboard interface was praised for its clean design and ease of navigation, although some participants suggested adding more descriptive text or instructions throughout the platform.

In the Concluding Questions section, 46.2% of respondents were interested in learning about self-driving code and related coding basics, while 53.8% were undecided. Participants desired hands-on experiences, more knowledge about sensors and circuitry, and a better understanding of the kinematics involved in autonomous driving. Overall, the survey results provide valuable insights into user experiences, allowing the mPAD team to make informed decisions about future enhancements to the platform.

Based on the section, the team learned that users prefer a minimalistic user interface that avoids clutter. A speedometer feature was also suggested to make the dashboard visually intuitive and enable users to type in their desired speed and angle. Moreover, the team plans to enhance the dashboard by including more detailed descriptions of its features, as some users suggested adding more text descriptions or instructions. By incorporating these changes, the team aspires to create a more user-friendly dashboard that caters to the needs and preferences of its users.

2.4 Mechanical Failures

The previous iterations of this MQP, back to and including 2019, deployed several unique variants of cars; some had been constructed from scratch, and others bought pre-built from online marketplaces. By the time the cars had reached the fall semester of 2022, most were in a state of disrepair.

Many of the handmade cars were split into large sections of components, some of which were damaged. The prebuilt Amazon Cars, named after the online marketplace they were bought from, displayed connection issues and a lack of receivers. PARV was a 3D-printed car that suffered a cracked frame, and its counterpart, PrePARV, was having issues keeping one of its wheels on its axle as it drove. Both cars were named, designed, and created by the 2019-2020 project team. Figure 2.4.1 shows what PARV and PrePARV were designed to look like by the said team.

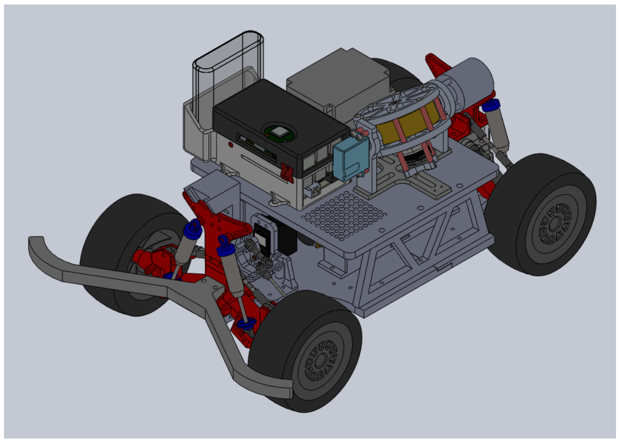


Figure 2.4.1: The ideal state of PrePARV, Reproduced as is from [17]

The source of each car's issues was eventually isolated, starting with PrePARV's unsecured wheel. The front right wheel was loosening itself from what was revealed to be a single screw acting as the axle, and that was in contrast to the front left wheel, which had a proper axle setup with ball bearings and fasteners. Upon further inspection, the same configuration existed within the front wheels of PARV. They were not the only student-made cars that displayed axle issues. The connection between Cleveland's wheel and steering linkage was shattered, and the steering servo motor was installed upside down.

Despite being pre-built, the four Amazon Cars and DonkeyCar- a pre-built car developed by the autonomous driving software team- also displayed issues. DonkeyCar's car bed lacked securing mechanisms, causing it to be ripped off when picked up, and each Amazon Car was missing key sensors and mounts. Amazon Car 1's steering servo was nonfunctional, making steering difficult, and Amazon Car 4 had foregone the car bed entirely, an Arduino rubber banded to the base (as shown below in Figure 2.4.2).

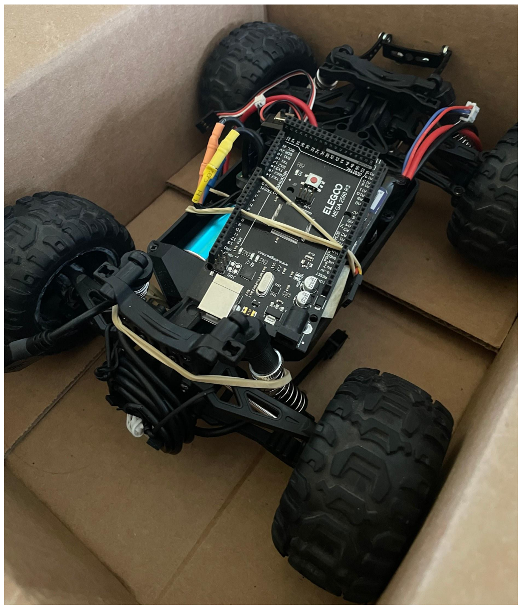


Figure 2.4.2: The original state of Amazon Car 4, reproduced as is from [17]

Every car lacked at least one required sensor or electrical component, such as ultrasonic sensors, IMUs, or servo shields. Fixing the cars was critical as the team entered the project, as out of the eight available cars, only one was functional enough to run the autonomous driving software.

2.4.2 Sensor Packages

The sensor packages and a USB camera were the two main areas that received change for the hardware updates. The sensor package saw two updates, adding a KY-003 hall effect sensor and a LiPo battery charge detection circuit for the dashboard to display the battery voltage. The Hall effect sensor is used to measure the RPM of a wheel or gear as it is placed near the wheel or gear so it can read the magnetic pulses emitted by a small magnet on the wheel or gear.

2.5 Conclusion

Overall, the mPAD v2 is a good starting point for the continuation of this project. With features such as the self-driving algorithm, object detection, and proper hardware for the cars, the team this year can work off of the current code and cars as just update it for more of a modular and functioning project.

3. Project Goals

The mission statement of mPAD is to "develop and refine a modular package that will enable any vehicle matching mPAD's minimum vehicle requirements to autonomously navigate a track and improve the user experience of the interface." To simplify this, the goal is to create an autonomous driving package that can be plugged into any vehicle that meets the requirements noted. With no prior information or training required, this should function on the track. With that stated, the mission statement is broken down into three objectives that must be achieved for the mission statement to be realized.

3.1 Improve Self-Driving Reliability

3.1.1 Overview

The first objective of this project is to improve the overall reliability of mPAD and its ability to navigate around a track on its own. This means mPAD should be able to consistently navigate on a track with two well-established outer lanes and stay between the lanes at all times. Compared to mPAD V2, the team must refine the self-driving algorithm. If it looks like it would be more effective to resume all of the prior work, the team will review the previous codes utilized and make improvements to the appropriate functions. To evaluate how the relationship would function with numerous vehicles driving simultaneously and how effective the self-driving algorithm would be, the team must add a multi-car configuration to the code so that all cars may execute the same code simultaneously.

3.1.2 Requirements

To achieve the stated objective of designing an autonomous program that can navigate a track, it is imperative that certain requirements be met to ensure the program functions effectively. Firstly, the RC car must be capable of maintaining its position within the designated lanes at all times, regardless of the color of the lane. This includes executing turns while staying within the confines of the lane. The program should be able to interpret its surroundings and identify the position of the lane relative to the car to ensure it stays within the prescribed boundaries. Furthermore, lighting conditions can be an obstacle to the effectiveness of the autonomous algorithm. The program must be designed to function efficiently in various lighting conditions, ensuring that the car can isolate and navigate the lanes despite any disturbances caused by lighting.

In addition to the above requirements, it is essential to note that the car should remain in its designated lane throughout its journey. The program should be tailored to ensure that the car does not intrude into other lanes, thus preventing any collisions or accidents. Finally, the car must be programmed to maintain a consistent speed of at least 15% throttle during its journey. While the car may slow down when approaching sharp turns, it should be able to maintain a steady speed throughout the journey. Overall, the successful implementation of an autonomous program that meets the requirements above would be a significant milestone in the development of self-driving technology, providing valuable insights into the capabilities and limitations of such systems.

3.2 Object Detection

3.2.1 Overview

Developing an autonomous driving program for an RC car that can stop at a detected stop sign is challenging yet necessary. The program must detect the stop sign, interpret its meaning, and execute a complete stop at the sign's location. To achieve this objective, the program must be able to process data from various sensors and cameras, identify the stop sign's position relative to the car, and make accurate decisions to bring the vehicle to a complete stop. Successful development of such an autonomous driving program would be a significant achievement, bringing us one step closer to the realization of fully autonomous vehicles.

3.2.2 Requirements

To develop an autonomous driving program for an RC car that can stop at a detected stop sign, specific requirements must be met to ensure the program functions effectively. Firstly, the car must be equipped with appropriate sensors, such as cameras or LIDAR, that can accurately detect the presence of a stop sign. The program must interpret the data collected by these sensors and detect stop signs. Secondly, the program must be designed to identify the precise location of the stop sign relative to the car. The program should be able to determine the distance and direction to the stop sign and make necessary adjustments to the car's speed and trajectory to bring it to a complete stop at the sign's location.

Additionally, the program must be able to make accurate decisions based on the data received from the sensors. It should be able to determine the appropriate time to initiate the braking procedure, the distance required to come to a complete stop, and any other variables that may affect the car's ability to stop safely at the sign. Overall, the successful implementation of an autonomous driving program that meets the requirements mentioned above would be a significant achievement in the field of self-driving technology, paving the way for the development of more advanced autonomous vehicles.

3.3 Web Dashboard

3.3.1 Overview

The next objective for the project is to enhance the user experience of the autonomous driving program's dashboard. After reviewing the current mPAD V2 dashboard, the team has identified an opportunity to improve its aesthetics and functionality to make it more appealing to a broader audience. To achieve this, the team aims to utilize the Donkey Car Open Source program to create a more cohesive and efficient dashboard that combines the best features from previous works.

To accomplish this objective, the team will focus on improving both the front-end component of the dashboard. Regarding the front end, the team will prioritize upgrading the dashboard's UI/UX to provide users with a more intuitive and visually pleasing experience. This will require expertise in front-end development and design principles to ensure the dashboard is functional and aesthetically pleasing.

3.3.2 Requirements

This project aims to enhance the user experience of the autonomous driving program's dashboard. To accomplish this, the team has identified specific requirements that must be met. First, the team shall redesign the dashboard's user interface (UI) to create a more visually appealing and cohesive user experience. Inspiration will be drawn from the results of the UI survey done by the previous MQP team [17]. The new UI design should be intuitive, easy to navigate, and provide relevant information to the users. Additionally, the team shall improve the dashboard's overall user experience (UX) by optimizing interactions to be effortless and straightforward, prioritizing the user's needs. The enhanced dashboard must be compatible with the Donkey Car Open Source program and seamlessly integrate with its features, tools, and functionalities.

The team shall utilize their expertise in front-end development to create a more efficient and responsive dashboard that is fast, reliable, and easily accessible across various devices. In addition to upgrading the UI/UX, the team will incorporate sensor data into the dashboard to provide users with clear and easily understandable information. The sensor data will be presented in a visually appealing and intuitive graph format, available for download upon request. The team shall ensure that the display of sensor data remains visible and accessible at all times while the car is driving. Finally, the team shall adhere to established design principles, incorporating appropriate color schemes, typography, and visual elements to ensure the dashboard is functional and aesthetically pleasing.

3.4 Modularity and Scalability

3.4.1 Overview

The last objective is developing and refining mPAD to be modular and scalable. This will require the team to make the modular package able to scale to any vehicle size as long as it has the necessary components. Redefining the software program that will allow customers to enter specifications of the vehicle dimensions is part of the objective. Developing a modular box containing all the required sensors to adapt to any automobile may also be possible. This will be done and detailed in a comprehensive user guide (as seen in the Appendix) that will assist the user in correctly installing mPAD and making the most of it.

3.4.2 Requirements

The objective of developing a modular and scalable mPAD system requires a set of specific requirements that the team shall adhere to. First, the team shall develop a modular package for mPAD that is adaptable and flexible to a wide range of automobile types and sizes. This modular package shall accommodate different vehicle dimensions and configurations, making it scalable and compatible with various automobiles. Second, the team shall redefine the software program to enable customers to enter specifications about their vehicle dimensions. This will allow for customization of the mPAD package according to the vehicle's specific requirements, ensuring the package is correctly configured for optimal performance.

Next, the team shall investigate developing a modular box containing all the required sensors to enable adaptation to any automobile. This modular box shall be easily installed and removed, ensuring that the mPAD system can be quickly and efficiently deployed across multiple vehicles. Fourth, the team shall create a comprehensive user guide (as shown in the Appendix) that will assist the user in correctly installing and making the most of the mPAD system. The user guide shall be easy to understand and include detailed installation, configuration, and system maintenance instructions. Finally, the team shall ensure that all design and development processes are adequately documented to facilitate future maintenance and upgrades. The documentation shall include detailed design specifications, software code, testing results, and user guide updates. By adhering to these requirements, the team shall develop a modular and scalable mPAD system adaptable to a wide range of vehicles, easy to install, maintain and operate, and provides accurate and reliable information to support safe and efficient autonomous driving.

3.5 Conclusion

In conclusion, the project goals outlined in this section are crucial for success. By clearly defining the objectives and desired outcomes, the team can work towards achieving them effectively and efficiently. The project goals will guide decision-making and provide a benchmark for measuring progress. Regular reviews and updates of the project goals will ensure that the project stays on track and that any necessary adjustments are made promptly. Overall, the project goals provide a clear direction and are essential for success.

4. Background

The development of self-driving cars, also known as autonomous vehicles, has been an area of active research and development for several decades. In this background section, the current state of knowledge and understanding in the field of autonomous vehicles will be discussed, including the technology behind autonomous vehicles, the types of autonomous vehicles currently being developed and tested, and the challenges that still need to be addressed before fully autonomous vehicles can be widely deployed. Furthermore, the section will delve into the companies currently developing and testing autonomous vehicles and how the technology is evolving. This knowledge will enlighten the obstacles faced in this MQP as a real-world company would face trying to achieve the mission of autonomous driving.

4.1 Autonomous driving

Autonomous vehicles, or self-driving or driverless cars, can sense their environment and navigate without human input. The development of autonomous vehicles is driven by a desire to improve safety, increase efficiency, and reduce the environmental impact of transportation. The technology behind autonomous vehicles is complex and includes a variety of sensors, such as lidar, radar, and cameras, as well as advanced algorithms for perception, decision-making, and control.

Vehicles have several levels of autonomy, as defined by the Society of Automotive Engineers (SAE) [12]. Level 0 is no automation, where the vehicle always controls the driver. Level 1 is driver assistance, where the vehicle can assist with steering or braking, but the driver is still responsible for monitoring the driving environment. Level 2 is partial automation, where the vehicle can control both steering and acceleration/deceleration, but the driver must still be prepared to take control in certain situations. Level 3 is conditional automation, where the vehicle can handle most driving tasks, but the driver must be ready to take over in certain situations. Level 4 is high automation, where the vehicle can handle all driving tasks but may still have limitations in specific environments or situations. Level 5 is full automation, where the vehicle can handle all driving tasks in all environments without any human input. This MQP is trying to achieve level 2 automation for various RC cars.

Autonomous vehicles have the potential to significantly improve safety, as they can reduce or eliminate human error, which is a significant cause of accidents. They can also increase efficiency by reducing traffic congestion, optimizing route planning, and reducing the environmental impact of transportation by reducing fuel consumption and emissions. The development of autonomous vehicles is ongoing, and technology is constantly improving. However, many challenges remain to be addressed, such as ensuring the safety of both passengers and other road users, dealing with unexpected or rare events, and developing robust, secure, and reliable systems.

4.2 Existing Self-Driving Vehicles

Autonomous vehicles have developed for several decades, with early research dating back to the 1950s. However, it was only in the 21st century that technology began to advance rapidly, with significant progress in perception, decision-making, and control. Currently, several types of autonomous vehicles are being developed and tested. Self-driving cars are fully autonomous vehicles designed for use on public roads. They use a variety of sensors (Such as LiDAR and Cameras) to sense their environment and navigate. On the other hand, autonomous trucks are autonomous vehicles designed for commercial transportation, such as long-haul trucking. They use similar technology to self-driving cars but are optimized for the specific requirements of commercial transportation. Autonomous shuttles are smaller autonomous vehicles designed in specific environments such as university campuses, industrial parks, and retirement communities. Autonomous drones are aerial vehicles for delivery, inspection, and surveillance.

Figure 4.2.1: Tesla Car, reproduced as is from [7]

Several companies are currently involved in developing and testing autonomous vehicles. Some notable examples include Waymo, Uber, and Tesla (Figure 4.2.1) in the field of self-driving cars. However, it should be noted that despite significant progress in recent years, the majority of autonomous vehicles are still in the testing phase, with limited deployment on public roads. Several challenges must be addressed before fully autonomous vehicles can be widely deployed, including ensuring safety for passengers and other road users, dealing with unexpected or rare events, and developing robust, secure, and reliable systems.

Autonomous vehicles have revolutionized the transportation industry and brought significant advancements in technology. The current autonomous vehicles rely on hardware and software to operate safely and efficiently. The software component includes complex algorithms and machine learning models that process data from various sensors, such as cameras and lidars, to make real-time decisions. Developing an autonomous vehicle requires rigorous testing and validation to ensure its safety and reliability. This workflow can inspire other industries to adopt a similar approach, where the focus is on developing a robust and efficient system. Moreover, the definition of autonomous vehicles can help refine our objectives by clearly understanding the capabilities and limitations of such systems. an intervention and can perceive and respond to their environment. By defining our objectives based on these capabilities, we can create more realistic and achievable goals for developing autonomous systems.

4.2.1 Donkey Car

DonkeyCar (Figure 4.2.1.1) is an open-source platform for building autonomous vehicles, specifically small-scale RC cars[18]. It provides a set of software tools and hardware components that can be used to build and train autonomous vehicles. The platform is built on Python and uses popular libraries such as TensorFlow and Keras for machine learning and OpenCV for image processing.

Figure 4.2.1.1: Donkey Car logo, reproduced as is from [18]

DonkeyCar allows users to collect data from an RC car using a variety of sensors such as cameras and IMU (inertial measurement unit) and then use this data to train machine learning models for tasks such as object detection and control. Once a model is trained, it can be deployed on the RC car, allowing it to navigate autonomously. To learn more about how this Donkey Car UI works, please refer to Appendix [18].

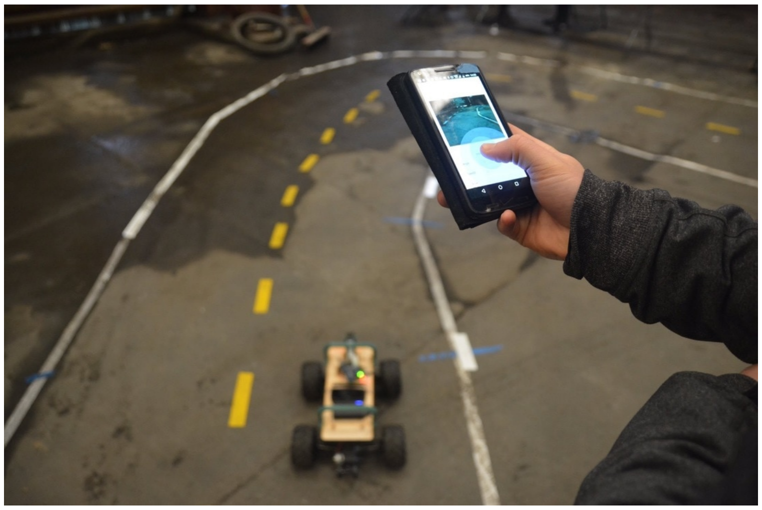


Figure 4.2.1.2: Car manually driving using Donkey Car, reproduced as is from [18]

The platform also includes a web-based interface for collecting and labeling data and monitoring and controlling the car (Figure 4.2.1.2). This interface allows users to collect data from the car in real time, label it, and then use it to train models. DonkeyCar also includes several pre-built models (examples of what an autonomous model looks like) and components such as a PID controller and a drive loop that can be used to control the car's movement. These pre-built components can be used to get a car up and running quickly, or they can be modified to suit the specific requirements of a project.

The platform employs a sophisticated web-based interface complemented by an advanced tool known as Donkey UI. The latter serves as a graphical training interface that streamlines the creation of a linear regression model by leveraging the user-selected dataset or "tub." This dataset comprises images and accompanying throttle and steering values(as seen below in Figure 4.2.1.3).

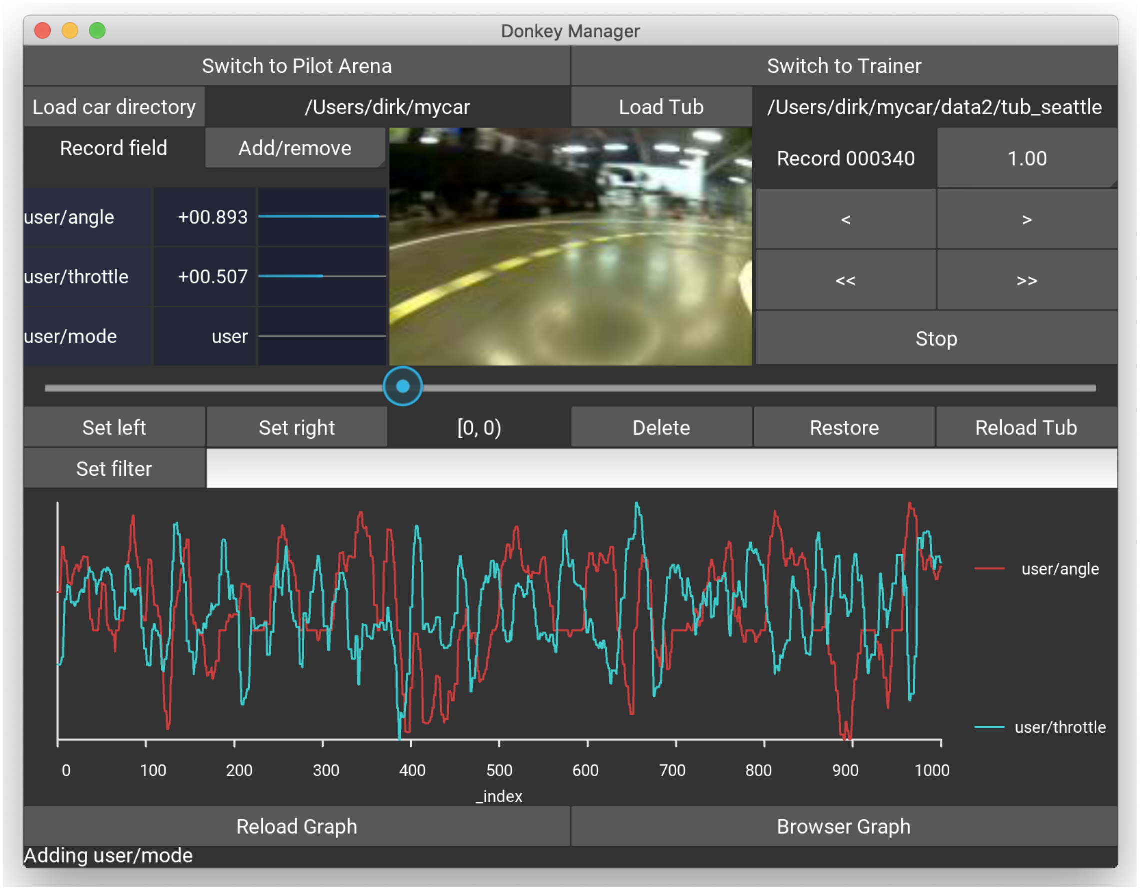


Figure 4.2.1.3: The graphical interface for Donkey Car UI, reproduced as is from [18]

The Donkey UI interface is designed to provide users with a comprehensive overview of the recorded images, affording them the ability to perform various modifications such as image deletion and filtering. Once the user completes the necessary adjustments, the "create model" button generates the linear regression model, which is then saved to the user-specified directory. This model can subsequently be transferred to a Raspberry PI platform, facilitating autonomous driving.

4.2.2 Amazon Deepracer

Amazon Deepracer (Figure 4.2.2.1) is another autonomous RC car driving solution. However, what separates DonkeyCar from Amazon Deepracer, is the scalability and modularity. While DonkeyCar can be utilized on any custom RC car, Amazon Deepracer is built specifically on cars designed by Amazon.

Figure 4.2.2.1: Amazon Deepracer, reproduced as is from [23]

The objective of Amazon Deepracer is to test RL models, as well as to get users into the Amazon Web Services (AWS) platform. The deepracer is built on an underlying Q-Model as its driving algorithm. A Q-Model is a version of reinforcement learning, which means the car learns independently on a Q equation (as shown below in Figure 4.2.2.2).

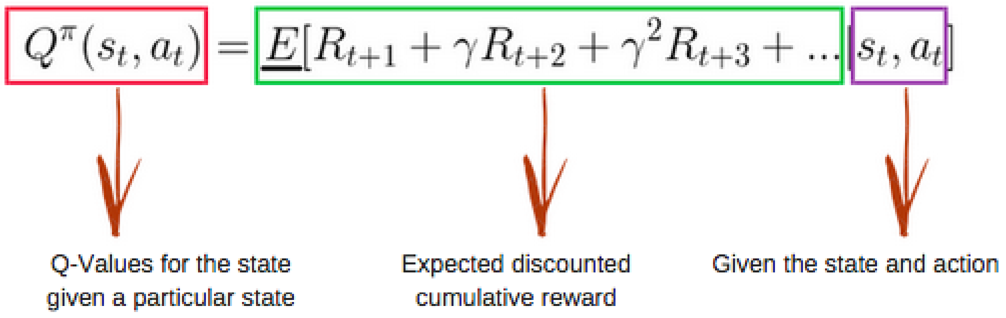

$Q^\pi(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$
Q-Values for the state given a particular state
Expected discounted cumulative reward
Given the state and action

Figure 4.2.2.2: Q Model, reproduced as is from [24]

This equation calculates reward values for actions, such as driving to the finish line as a positive value and crashing as a negative reward. Based on these rewards and what reward it received last time in a similar state, it learns over time to maximize reward. The drawback of an RL model is that it is heavily dependent on a simulation since a car in the real world can not afford to keep crashing into the wall to learn.

4.2.3 HuskyLens

HuskyLens (Figure 4.2.3.1) is a visual artificial intelligence sensor developed by Husky AI, a company specializing in AI products [9]. It is a small, low-cost, easy-to-use sensor accessible to makers, educators, and researchers.

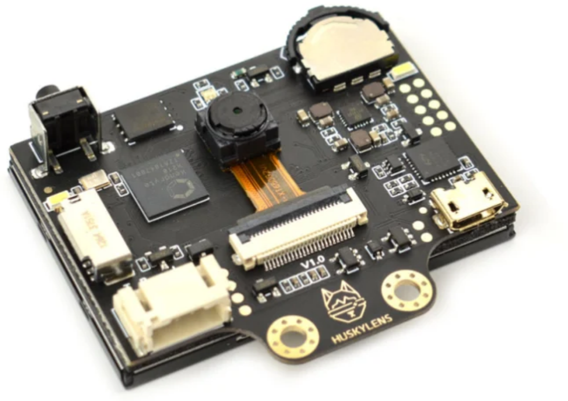


Figure 4.2.3.1: HuskyLens camera, reproduced as is from [27]

The sensor is based on machine learning algorithms and is capable of performing a variety of tasks, such as object detection, object tracking, facial recognition, and gesture recognition. It is equipped with a built-in camera, a microphone, and a speaker, which allows it to interact with the real world. HuskyLens can detect multiple objects simultaneously, track their movement, and classify them based on their characteristics, such as color, shape, and size. This feature makes it highly useful for applications such as robotic control and autonomous driving. Additionally, the sensor can recognize facial expressions, hand gestures, and speech commands, making it useful for interactive displays and human-computer interactions.

Furthermore, HuskyLens provides a cloud platform for easy data storage and analysis, allowing remote access and monitoring of the sensor's data. In summary, HuskyLens is a visual AI sensor that is low-cost, easy to use, and designed for makers, educators, and researchers. It can perform various tasks such as object detection, object tracking, facial recognition, and gesture recognition, and it can be integrated with microcontrollers to interact with the real world. It also provides a cloud platform for easy data storage and analysis.

4.3 Raspberry Pi

In this project, the team utilizes a Raspberry Pi (Figure 4.3.1) throughout the year. This is an essential piece of this project to gain autonomous driving functionality.

Figure 4.3.1: Raspberry PI 4, reproduced as is from [26]

4.3.1 Raspberry Pi 3B+

The Raspberry Pi 3 Model B+ is a small single-board computer developed by the Raspberry Pi Foundation. It is the third generation of the Raspberry Pi and was released in March 2018. It has a 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU, dual-band 802.11ac wireless, Bluetooth 4.2/BLE, faster Ethernet, and Power-over-Ethernet support (with separate PoE HAT). It also has the same 40-pin GPIO header as the previous models and is compatible with the same peripherals and expansion boards. The Raspberry Pi 3 Model B+ is a versatile device that can be used for various projects, including media centers, game consoles, and home automation systems.

4.3.2 Raspberry Pi 4

The Raspberry Pi 4 is a small single-board computer developed by the Raspberry Pi Foundation. It is the fourth generation of the Raspberry Pi and was released in June 2019. It has a 64-bit quad-core Cortex-A72 CPU running at 1.5GHz, dual-band 802.11ac wireless, Bluetooth

5.0/BLE, Gigabit Ethernet, USB 3.0 ports, and dual-monitor support via a pair of micro-HDMI ports. It also has the same 40-pin GPIO header as the previous models and is compatible with most of the same peripherals and expansion boards. The Raspberry Pi 4 offers significantly more power than its predecessors, making it suitable for more demanding projects such as running multiple applications, video playback, and gaming.

4.4 Python

Python is a widely-used, high-level programming language well-suited for scientific computing, data analysis, artificial intelligence, and other fields. It is known for its simplicity, readability, and expressiveness, which makes it an ideal choice for many developers.

One of Python's main advantages is its vast library and framework ecosystem. These libraries and frameworks provide a wide range of functionality and make it easy to perform everyday tasks, such as data manipulation, machine learning, and web development. Some popular libraries in autonomous vehicles include NumPy, pandas, and scikit-learn for data manipulation and analysis and TensorFlow, PyTorch, and Keras for machine learning [6].

Python is widely used in autonomous vehicles for data preprocessing, sensor fusion, object detection, and decision-making tasks. Many autonomous vehicle software platforms, such as ROS (Robot Operating System) and Autoware, use Python as the primary programming language [28]. In recent years, Python has been increasingly adopted in autonomous vehicles due to its simplicity, expressiveness, and vast ecosystem of libraries and frameworks that make it easy to perform complex tasks. Python is expected to grow in autonomous vehicles as the technology evolves and becomes more sophisticated.

4.5 OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions widely used in computer vision and image processing. It provides a wide range of functionality, including image capture, image processing, feature detection, and machine learning, making it an ideal choice for many autonomous vehicle developers.

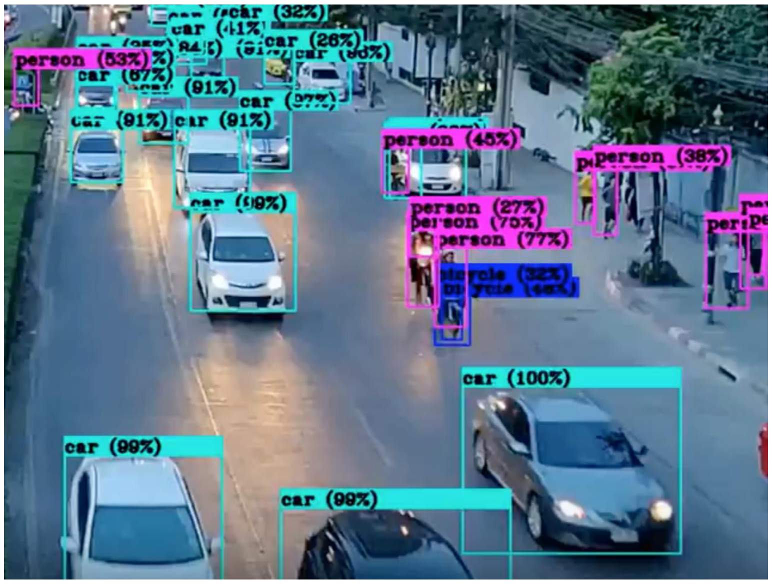


Figure 4.5.1: Demonstration of OpenCV, reproduced as is from [4].

One of the main advantages of OpenCV is its support for a wide range of image processing techniques, including edge detection, feature detection, object detection, and image recognition (as shown in Figure 4.5.1). These techniques are essential for the perception component of autonomous vehicles, as they allow the vehicle to detect and recognize objects in its environment, such as vehicles, pedestrians, and traffic signs.

OpenCV also includes several machine learning algorithms, such as neural networks, decision trees, and support vector machines, which can be used to train and develop models for object detection and image recognition tasks. These models can detect and recognize objects in real time, making them suitable for autonomous vehicles. In addition to its image processing and machine learning capabilities, OpenCV provides a wide range of functionality for working with video streams, including motion detection and object tracking. These capabilities are essential for the perception component of autonomous vehicles, as they allow the vehicle to track and understand the motion of objects in its environment, such as other vehicles or pedestrians.

OpenCV is widely used in autonomous vehicles and integrated into many software platforms, including ROS (Robot Operating System) and Autoware. It is expected that the use of OpenCV will continue to grow in autonomous vehicles as the technology evolves and becomes more sophisticated.

5. Methods

In the methods section of this paper, the team will cover how they made mPAD operational and utilized and modified the Donkey Car open-source autonomous driving platform. This section is delineated by term and is ordered chronologically.

5.1 A Term

The team's first task was to analyze and become familiar with the project and its progress. This primarily included reading the reports from the three previous teams (2019-2020 to 2021-2022), going through the setup guide they left, and going through the mPAD V2 Github Repository. More can be read about these in chapter 2 of this paper. The team primarily faced issues when reading through the latter two, as there were some errors in the setup guide, such as the servo wiring and little to no documentation in the Github repository. During this time, the team also collected related external sources to provide further background on autonomous driving, computer vision, and remote control vehicles. More can be read about these sources in chapter 4 of this paper. Once the team felt they had enough background with the topics and had developed enough familiarity with the project, they moved forward with defining their own goals. Once the goals were defined, they moved on to operationalizing mPAD. The goals the team created can be found in Chapter 3.

When the team started working on mPAD software, they discovered their hardware had many crucial flaws. Wheels were missing from some of the cars, drive trains were broken, and many electrical components had been wired incorrectly or were damaged to the point of uselessness. The team had to work around these issues as the software was their main priority, and they would log any problems they could not temporarily fix themselves. They could then bring these issues up for discussion at weekly meetings.

In addition to hardware issues, the team faced many challenges in working on the mPAD software. Most of the Raspberry Pis had their SD cards swapped or erased and needed the mPAD software to be reloaded. There was a limited supply of HDMI cables and adapters compatible with these Pis, and the ones would often show nothing more than a black screen on startup. This created a bottleneck and meant that for much of the term, only one or two Raspberry Pis would be operational at a time, slowing down the team's productivity.

Finally, once the team could access the previous year's software, they found it lacking documentation and difficult to use. This, combined with hardware issues, made mPAD very difficult to work on, and finding the origin of any problems was almost impossible; it was here that the team decided to split into two subteams: one which would continue to work on mPAD to see if they could make it operational and one which would try to use Donkey Car, an open-source autonomous driving platform with ample support and documentation.

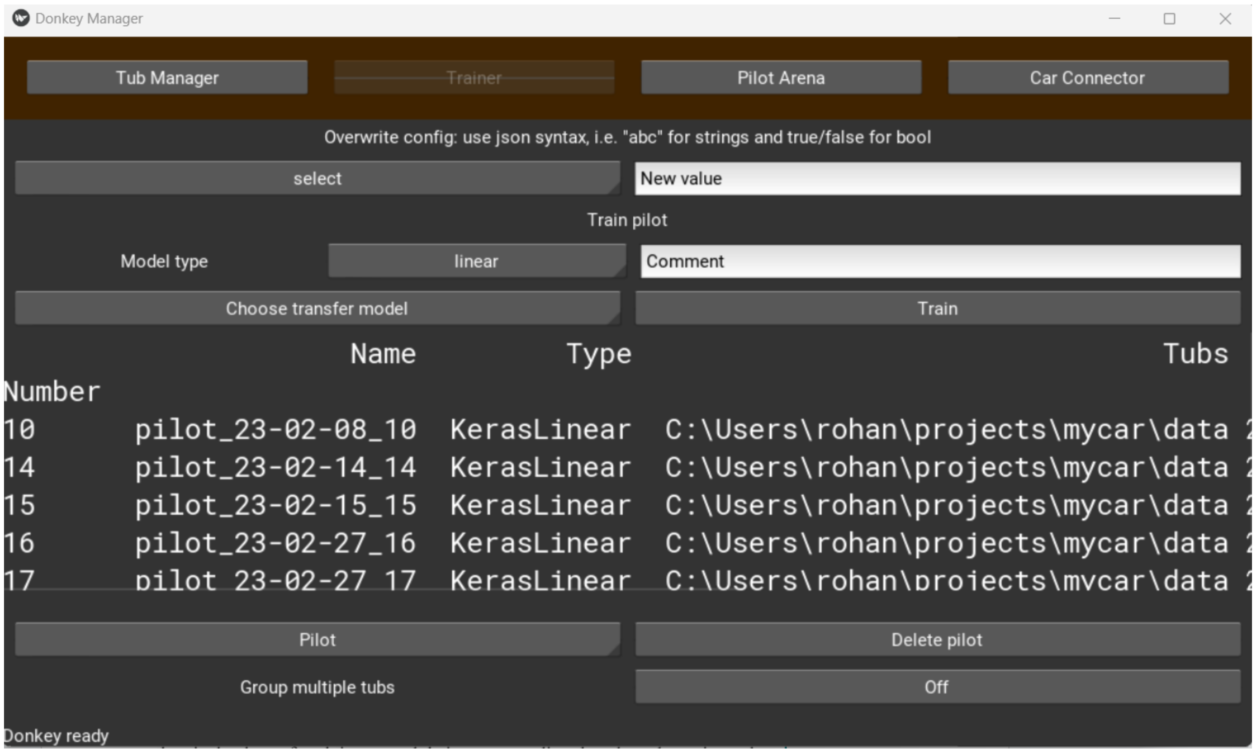
5.2 B Term

The two subteams continued to work in their respective areas throughout B term, and a new teammate, Allison Colon, was brought on to help with hardware issues.

5.2.1 Donkey Car

The Donkey Car team started sorting through numerous version compatibility errors until they got Donkey Car to work on their computers and Raspberry Pis. Once the software was usable, they read through the documentation on creating an autonomous driving model and recorded the necessary data. In order to capture this data, the vehicle first needed to be connected to a Local Area Network (LAN) and have its servos calibrated. The team would now use their cell phone hotspots as the LAN with no other options. The servos were live-adjusted by typing random values into a script and recording their ranges. For instance, the steering servo might turn the vehicle left at value 250, center at 350, and right at 450. These values were put into a configuration file the car would use to convert joystick inputs later. With the car ready to control, the team tested driving from their phones by going to the Donkey Car Dashboard on their phone's browser and using the screen as a joystick. It should be noted here that the motors behind any vehicle can be different, and one should start with a low maximum throttle percentage. The team ran test laps and found the vehicle somewhat challenging to control using a phone screen as the interface. Other control methods, such as radio and gamepad via a web interface, were investigated but proved less valuable. Being directly connected to the servos, the radio control interfered with data collection, and the gamepad, although not suffering from this problem, produced an unwanted input signal, making training impossible.

For the actual training of the model, the team would record manually driven laps around the track. Once the record button was hit, the images would be stored in a database and associated with the steering and throttle values. The data collection rate was constant with respect to time, so the amount of data collected per lap depended on the vehicle's speed. Some cars being faster than others had less data recorded per lap than slower ones. The team found that the average vehicle would record about 1,000 frames of data per lap (~75 feet) on the track. Once the data had been collected, the team would transfer it to a laptop, which would be trained by the Donkey Car User Interface (shown below in Figure 5.2.1).



Overwrite config: use json syntax, i.e. "abc" for strings and true/false for bool

New value

Model type: linear

Comment

```
                    Name             Type                         Tubs
Number
10      pilot_23-02-08_10  KerasLinear  C:\Users\rohan\projects\mycar\data
14      pilot_23-02-14_14  KerasLinear  C:\Users\rohan\projects\mycar\data
15      pilot_23-02-15_15  KerasLinear  C:\Users\rohan\projects\mycar\data
16      pilot_23-02-27_16  KerasLinear  C:\Users\rohan\projects\mycar\data
17      pilot_23-02-27_17  KerasLinear  C:\Users\rohan\projects\mycar\data
```

Group multiple tubs: Off

Donkey ready

Figure 5.2.1: Training model process of Donkey Car with saved models

The team would select the linear regression model, and patterns would be drawn between the pixel values of each image and their corresponding throttle and steering values. The team found that if the model were trained in only six epochs or less, something was missing from the dataset, such as the steering and throttle data, and the model would not perform. Once the team realized how to ensure this data was included during training, they found that the number of required epochs hovered around 20. Upon completion of training, the model would be copied back to the vehicle and could be tested via the web dashboard. The team continued this cycle of training and testing until they found what they thought were the optimal conditions for a model. This considered laps, frames collected, speed, vehicle type, and driver. Even at its most robust, however, the model was very susceptible to visual noise and changes in lighting. Waiting so much as a few hours after training would result in lighting conditions different enough to throw the model off completely. The team considered this and proposed incorporating image preprocessing next term.

5.2.2 mPAD

Concurrently, the mPAD team focused on using a self-driving algorithm based on HuskyLens. HuskyLens is a camera with built-in machine-learning capabilities. The main idea for this path was to utilize the camera to identify lines on the track and, based on where the lines were in relation to the car, calculate steering and throttle values.

The subteam first converted code from C++ libraries to Python to achieve this. Huskylens was initially built on a C++ code frame. To optimize workflow and integrate with other aspects of the project, the team had to manually convert all the code to Python and create libraries for huskylens in Python.

With the newly built libraries, the team could train the HuskyLens on line tracks and create an algorithm that enabled them to implement the desired workflow. The team's issue, however, was the camera's field of view. With the HuskyLens camera only being around 72^{o}, the camera had issues detecting lines to both sides of the car. With these flaws, this implementation route was ultimately dropped.

5.2.3 Car Repair

The new term brought new teammates, the newest addition to the team being a mechanical engineer, to focus on repairing the crucial hardware flaws found and recorded in A term. The first order of business regarding repairing the cars was to compile and organize all available previous documentation. Some cars had more documentation than others. The twins

PARV and PrePARV had the most extensive and detailed documentation, including component lists and step-by-step building instructions. Using this documentation, the first car could be repaired fairly quickly; PrePARV, the only functional car and the one that had been used for testing, received a repaired axle and new ball bearings. Unfortunately, the ball bearings were from its companion, and needing new ones would set PARV far back in the repair order.

The next cars to be selected for repair were the first three Amazon Cars. With intact car beds, most of the work was towards testing functionality and adding all necessary electronics onto the beds. Unfortunately, the immediate problem of a lack of manual controllers and accompanying receivers showed itself. This slowed development, as after enough of both were located in the dedicated MQP closet, LiPo batteries were also acquired to power the motors. Upon closer inspection, many of the LiPo batteries that had been in use had become inflated, meaning the electrolytes in the battery were breaking down, and the batteries themselves were fire hazards. This meant the old batteries needed safe disposal, and new batteries had to be ordered. However, the opportunity was used to order more electronic speed controllers, as they were in limited supply, and more would need to be ordered eventually.

It was during manual testing that Amazon Car 1's faulty servo was identified. It was not fully nonfunctional, as it would only turn right, but that only made the issue tougher to spot. The compact nature of the cars did not assist in the disassembly and reassembly process, which was required for testing. This also applied to the new camera mounts, designed by the team and 3D printed at a personal residence; both printing and installing were also a time sink; the compact car did not allow for screwdrivers, and Allen wrenches to fit, and all components had to be screwed in manually.

Repairing the Amazon Cars took the majority of B Term, but DonkeyCar was able to be completed before the end of the term. It was more structurally and electronically sound than the Amazon Cars; all required for functionality was a new Electronic Speed Controller (ESC), a camera mount, and a secure top bed. By the time the term ended, the number of workable cars had gone from one to five.

5.3 C Term

In C term, the Donkey Car subteam, now known as the preprocessing and object detection team, worked on making the autonomous driving model trained by Donkey Car less vulnerable to noise and changes in lighting. The Car Repair team repaired and refurbished two cars, PARV and Cleveland. The Dashboard team added some helpful new features to help operate the vehicle.

5.3.1 Image Preprocessing

It had been noted in B term that the car was easily thrown off by noise, and the team first investigated any possible built-in solutions. They found that the Donkey Car UI had two main categories of image modifications, which could be applied called augmentations and transformations. Image augmentations were only applied to data and could only affect a model's training. The team attempted to apply these to training but found little to no improvement in the performance of any vehicles they were applied to. On the other hand, image transformations were always applied to the Donkey Car pipeline and had a constant effect. The built-in library for these modifications was also limited, containing a cropping effect and a trapezoidal filter that would mask most of the image, leaving only the area where the lane lines were most likely to be found. The team tried the trapezoidal filter but found little improvement in the vehicle's performance. For now, the team decided to leave these features alone and try other methods of limiting unnecessary information.

The team looked at the camera.py file to try and edit the image before the Donkey Car pipeline received it. Here the team attempted to implement a simple grayscale filter on the image by multiplying red, green, and blue channels by 0.216, 0.587, and 0.144, respectively. Despite adding a for loop, which iterated over the entire image, this modification has a limited effect on the response time of Donkey Car while improving the overall driving performance, completing more laps than its predecessor at different times of the day. The team continued to test these transformations by adding Gaussian blur to the transformation, further obscuring unnecessary information and improving the vehicle's performance.

With the camera.py file now acting as a platform for custom image transformations, the team decided to try and remove nearly all unnecessary information by applying a line detection transformation. In order to accomplish this, the team used the OpenCV library, allowing them to apply Canny edge detection and then a Hough line transformation. These modifications combined transformed the image into a mostly black screen, with one of the color channels being used to represent any edges detected in the image over a certain length. This solution encountered many errors at first. The model’s performance began worse than without image preprocessing, and it was unclear why. After much testing, the team hypothesized that the model found patterns in the negative space (black pixels where no lines were detected) and made incorrect decisions. In order to test this hypothesis, the team created a random noise filter that could be applied to the image in the preprocessing phase. The filter would iterate over each pixel in the image, and if it were black, meaning there was no edge there, it would randomly apply a small brightness. This effect would not apply to pixels that were already bright. This filter would be invisible to anyone looking at the image on the screen. However, now when the model was trained, it would not be able to draw any correlations between the user input and non-edge pixels, leaving the lane lines and other edges, such as walls, as the only information to draw patterns. This proved to be one of the most crucial features included in preprocessing as it allowed a vehicle to be trained and run at any time, providing sufficient lighting for the camera sensor to see. Once this was complete, other transformations, such as a custom cropping filter, were applied to the image for fine-tuning (as shown below in Figure 5.3.1).

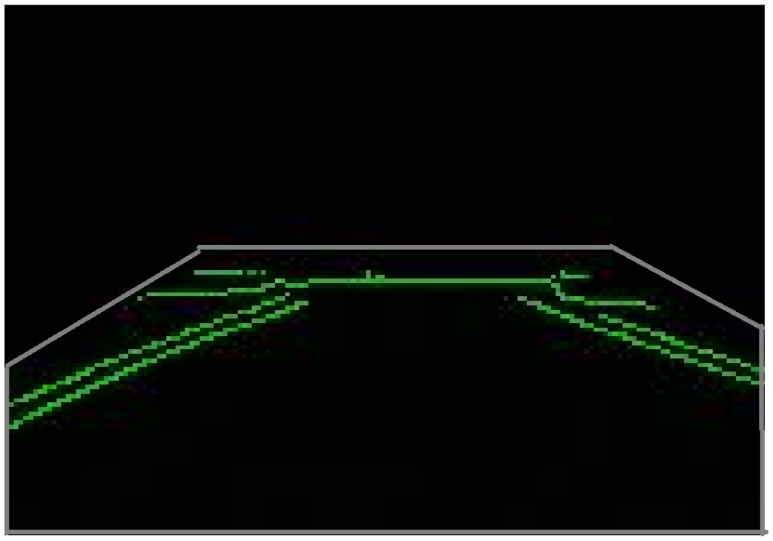


Figure 5.3.1: Trapezoidal cropping on pre-processed image to minimize data seen

Once preprocessing was complete, the team noted that it did not solve all problems involved with training a model. Driving all the time perfectly while training would result in a perfect model, but when transferred to the real world, the car would gradually but inevitably drift into a situation it had never seen before and not know how to react, usually resulting in a crash. To account for this, the team found it best to train some laps perfectly and others where the driver would intentionally correct mistakes, such as drifting outside of the lines and then quickly steering back in. The best ratio of “ideal” laps to “corrective” laps is hard to know, but the team successfully gave each category equal amounts of data.

5.3.2 Object Detection

Having fine-tuned the autonomous driving portion of the project to a good point, the preprocessing subteam turned to object detection, for which some work had already been done. The team had originally been trying to use the TensorFlow object detection API but could not achieve a usable framerate, only receiving around four frames per second. Even with a gap as small as ~250 milliseconds, the vehicle would always move too quickly to have a significant chance of recognizing signage. To try and increase framerate, the team installed Tensorflow onto the Raspberry Pi so that it could run itself. However, this did not perform much better, only coming in at around five frames per second, which still proved too slow for practical use. The team then decided to use TensorFlow lite instead, which, while capping the maximum potential of image detection, might run fast enough to be useful. Unfortunately, this model could not achieve usable framerates, hitting around 7 on average. It should be noted that these numbers were all averages and that the framerate was not uniform throughout any drive. The installed (non-API) object detection methods regularly oscillate between framerate spikes and dips with a wide range. Some of these values were in the range of practical use but could not be relied on as they did not occur frequently enough. At this point, having tried all existing models, the team decided to create a custom convolutional neural network that would be compact and only be trained to find stop signs. While limited in capability, this minimal architecture would prove that object detection could be integrated into the autonomous driving model and be the first model practical enough to work. The team’s custom model could run consistently at 20 frames per second and detect stop signs with a success rate of around 84%.

Integrating this model into the autonomous driving model proved to be an issue of equal size. Even when running on the most powerful Raspberry Pi 4 model, the autonomous driving model and object detection model would not run simultaneously, using more than 94% of the CPU processing power at any given point. Here, the team decided that object detection could run simultaneously if a vehicle were allowed to run with two Raspberry Pis. While a less-than-ideal solution, it was known that a Raspberry Pi could handle each task separately, so it was set up to be tested. The team attached a second camera to one of the vehicles and connected it to the Raspberry Pi running object detection. When this Pi detected a stop sign, it would use a GPIO pin and common ground connection to send a signal to the Pi running Donkey Car. When this signal was received, the throttle would be overridden, and the vehicle would stop. The model would only resume once the signal was gone. This setup allowed one vehicle to autonomously drive around a track and stop when it encountered a stop sign. This concept was also used to test multiple cars traveling in the same direction by attaching a stop sign to a slower car and placing a faster car equipped with object detection behind it. This configuration was considered satisfactory until more processing power could be acquired later this year or for a future team.

5.3.3 Dashboard Work

To start the term, the goal for the mPAD V3 dashboard was to update Donkey Car's current dashboard to include a modern and simplistic design while adding specific features from the previous team's dashboard. When first working on the dashboard, the color palette was unoriginal and basic, using a white background with blue highlights for the buttons. The team identified this as a problem that needed to be fixed with a new color palette. The next step was updating the dashboard layout with a more camera-focused design. This design was important due to the camera's large size and its being the main focus for a user. The new layout was oriented with the camera in the center and the joystick container below. It makes it easier for training as the user can aim the joystick in the direction they want on the camera display. Next, all vehicle controls were relocated to the left side of the screen, with steering modes on the top and the throttle modes on the bottom. This separates each control based on the type of mode they control. Finally, the right side of the dashboard was reserved for graphs to display the sensor data gathered by the car's ultrasonics, IMU, and TF Luna. When making this column, displaying all graphs made the page unnecessarily long. Therefore, a drop-down was made to select the graph the user wants to display, thereby shortening the dashboard to one page without needing to scroll. A problem when testing was that the team needed exact values for the throttle, as 20% was too slow, but 30% was too fast. In order to solve this problem, a slider for the throttle was added instead of the original drop-down. The dashboard layout was finalized at this time, besides getting sensor data to update the graphs in real-time.

5.3.4 Car Repair

During this time, repairs had begun on PARV and Cleveland. These two cars required much more work than those from B Term, as they had missing electronics and broken frames, gearboxes, and steering linkages. Missing pieces must be ordered or looked for in the designated MQP closet, starting with PARV's ball bearings. Taken for PrePARV in B Term, the large, 3D-printed car also suffered from a missing camber link and slipping gears. As the new mounts and frame were printed, the gear connecting the motor to the gearbox was tightened, and the steering linkage was repaired. Afterward came Cleveland, which posed a design challenge in the form of the unsecured steering linkage. The bolt hole for the wheel was within the car base, carved into a curved section of the car. All previous cars had adequate documentation to find old model files for reprinting. However, Cleveland lacked said documentation, and with the complexity and size of the piece, 3D printing a new base was not worth it. Instead, a new piece was designed to be slotted around the fracture. Wrapping around the bolt hole, the piece was drilled into place and secured to the axle.

5.4 D Term

With the bulk of the work for the main project complete, the work in D term was primarily focused on adding sensors to vehicles, integrating their data into the dashboard, and preparing the project’s poster and final report.

The sensor package to be included on every vehicle must contain one TF Luna LiDAR distance sensor, one MPU 6050 inertial measurement unit (IMU), and four ultrasonic distance sensors. While most vehicles were equipped with ultrasonic distance sensors, none had TF Lunas, and many had either damaged IMUs, Arduino, or both. The team ordered components to compensate and completed the packages for the vehicles over the rest of the term.

The TF Lunas required a 6-pin connector to communicate but did not have a connector that could be directly connected to the Arduino. To avoid waiting, the team cut the connectors and soldered the four wires used to jumper cables that could be directly plugged into the Arduino. The connections were shielded with electrical tape, and the remaining two unnecessary wires were taped off in a loop to not be in the way.

When the new IMUs arrived, they needed to be assembled to test them on the breadboard. The team soldered on the headers to each IMU and tested their ability to communicate through the I2C bus on an Arduino known to be functional. Once their functionality was tested, they were placed on the breadboard of each vehicle.

Many of the old Arduino Mega 2560 Rev3 used by the team were functional, except for the I2C bus. However, this was a critical feature as the IMUs could only communicate using I2C.

Upon arrival, the functionality of the I2C buses and serial ports of each Arduino was tested before it was swapped with the older Arduino on each vehicle.

The sensor data for each vehicle, except the webcam, was collected by the Arduino and sent to the Raspberry Pi via serial communication. In order to deliver data from several sensors in the same line, the team created a script that, in its loop, collected the data from each sensor and loaded it into a formatted string. The string contained variable names followed by a semicolon and the sensor's value for the given cycle. Below is an example of one line that might be sent via serial. It shows the values of four distance sensors (dist1-4), linear accelerations from the three sensors on the IMU (ax, ay, az), angular accelerations from the other three sensors on the IMU (gx, gy, gz), and the distance read by the TF Luna (luna).

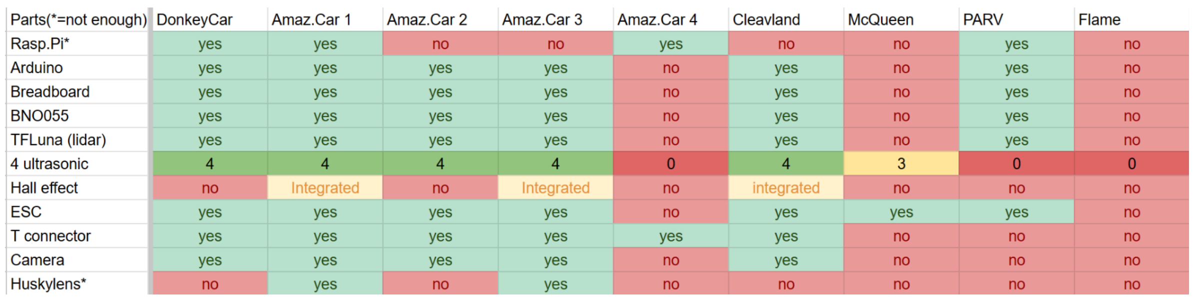
dist1:43 dist2:12 dist3:9 dist4:2 ax:2 ay:1 az:9 gx:3 gy:0 gz:1 luna:72

This format would be easily parsed and converted into a Python dictionary to be used by the Raspberry Pi. Distance units were sent in centimeters, linear acceleration in meters per second squared, and angular acceleration in degrees per second. This information was taken in during Donkey Car's main loop and used to update the dashboard. At this point, the sensor package was fully integrated and could be seen using the web dashboard. At the same time, Amazon Car 4's repairs were underway, adding the new car bed, sensors, and a brand new, hand-soldered ESC.

Once this work was completed, the team considered the bulk of the physical work finished and moved on to making the poster and presentation for project presentation day. New sensor housings and camera mounts were printed to prepare DonkeyCar and Amazon Car 2 for presentations. The poster and presentation are in appendices 14.5 and 14.6, respectively.

6. Mechanical Changes

This section focuses on the changes and fixes made to each car in the mPAD project fleet. Each subsection will detail the steps taken to repair each car into functionality.



Parts(*=not enough)	DonkeyCar	Amaz.Car 1	Amaz.Car 2	Amaz.Car 3	Amaz.Car 4	Cleavland	McQueen	PARV	Flame
Rasp.Pi*	yes	yes	no	no	yes	no	no	yes	no
Arduino	yes	yes	yes	yes	no	yes	no	yes	no
Breadboard	yes	yes	yes	yes	no	yes	no	yes	no
BNO055	yes	yes	yes	yes	no	yes	no	yes	no
TFLuna (lidar)	yes	yes	yes	yes	no	yes	no	yes	no
4 ultrasonic	4	4	4	4	0	4	3	0	0
Hall effect	no	Integrated	no	Integrated	no	integrated	no	no	no
ESC	yes	yes	yes	yes	no	yes	yes	yes	no
T connector	yes	yes	yes	yes	yes	yes	no	no	no
Camera	yes	yes	yes	yes	no	yes	no	no	no
Huskylens*	no	yes	no	yes	no	no	no	no	no

Figure 6.1: A screenshot detailing the original state reproduced as is from Appendix [15.1.1]

Figure 6.1 details a spreadsheet of electronics each car was already equipped with at the beginning of the project. As each car was repaired and refurbished, it would be subject to manual testing. This testing would involve connecting the cars' ESC to an Ikonnik ET3 3-Channel receiver and accompanying transmitter, ensuring that the car could properly receive steering and throttle signals.

To ascertain the functional efficacy of the vehicle and its compatibility with the Donkey Car program, the team performed a calibration sequence to test the steering and throttle functionalities. This entailed sending pulse width modulation (PWM) signals to the motors to precisely control the steering and throttle response of the vehicle. Through this process, the team gained a comprehensive understanding of the communication between the Donkey Car code and the physical motors of the vehicle, thus affirming its readiness for operation.

6.1 Highlights

The final fleet consisted of eight cars. To start, PARV and PrePARV were two cars named, designed, and created by a previous project group. They are student-made, with 3D-printed bases, axles, and supports. They are both very sturdy and due to their large size, they are able to comfortably hold all required components, as seen in Figure 6.1.1.

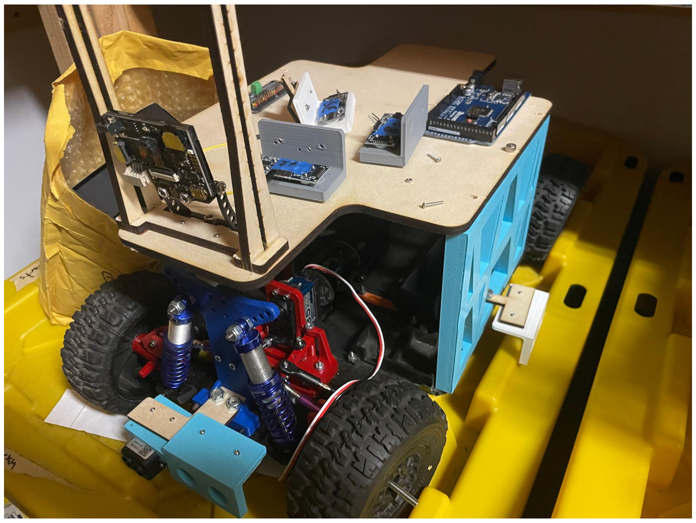


Figure 6.1.1: PrePARV in its original state

Next were the four Amazon Cars. They are named as such because the original base, along with the motor and steering servo, was purchased from the online retailer of the same name. With premade bases and base components needed for driving, they were outfitted with expanded car beds to hold the electrical components necessary for autonomous driving. Figure 6.1.2 shows Amazon Car 3, the first to be repaired, which became the blueprint for the other Amazon Cars.

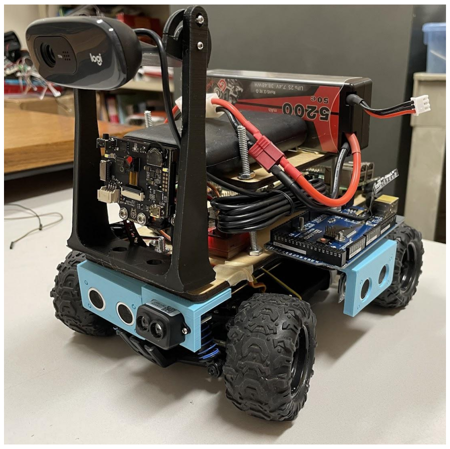


Figure 6.1.2: Amazon Car 3 after obtaining all necessary components

DonkeyCar is another pre-made car developed by the same creators of the autonomous driving software of the same name. It sports a much lower center of mass and a wider car bed, which required new securing wires. This can be seen in Figure 6.1.3, with the car bed being uneven. There were also issues with the ESC, which would need to be replaced.

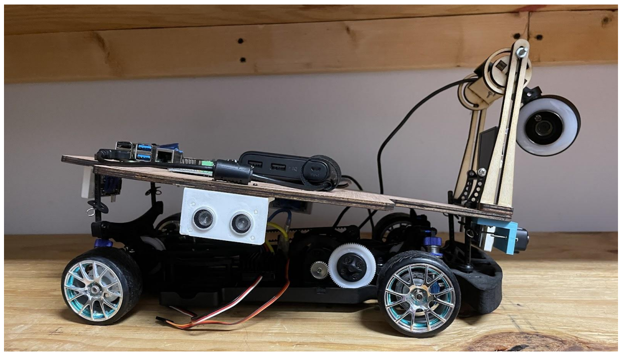


Figure 6.1.3: Donkey Car in its original state

Finally, Cleveland is another student-made car, differing from PARV and PrePARV with its large, student-designed gearbox and a much longer design to compensate. It was sounded electronically, but the connection between its steering linkage and car bed was severed, as seen in Figure 6.1.4. A new, supplemental piece would be required to fix this issue.

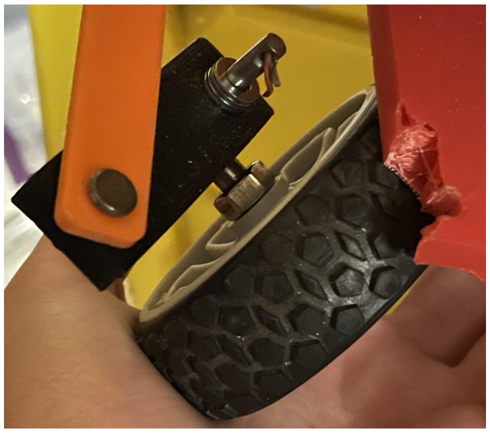


Figure 6.1.4: The original state of Cleveland's axle

In preparation for the repairs, a new camera mount was designed to replace the current ones on the cars, as seen in Figure 6.1.5. The team designed these new mounts, which were 3D printed out of PLA filament and would be installed on each car after printing. Large screws adhere the stand to the base, and smaller screws adhere the mount to the stand and the camera to the mount.

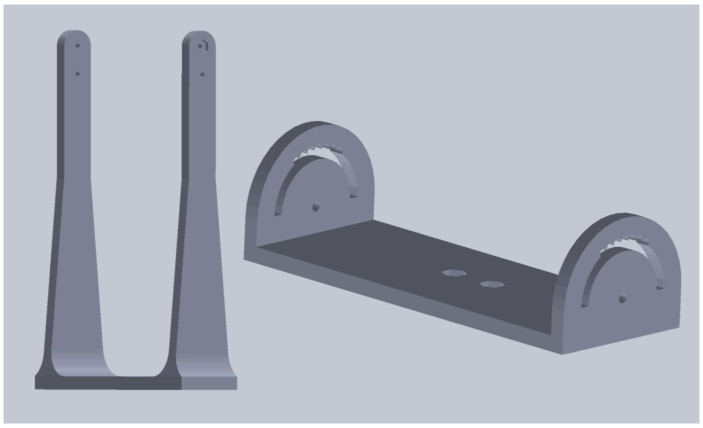


Figure 6.1.5: The models of the camera stand a mount, respectively

6.2 PrePARV and Amazon Cars 1-3

PrePARV was the most intact car out of the fleet, with wiring and batteries connected from the previous year. It worked well enough to begin autonomous testing, and after recognizing that its counterpart PARV displayed similar issues, PARV's singular ball-bearing wheel was taken and used for PrePARV. Figure 6.2.1 shows PrePARV with its repaired wheel.

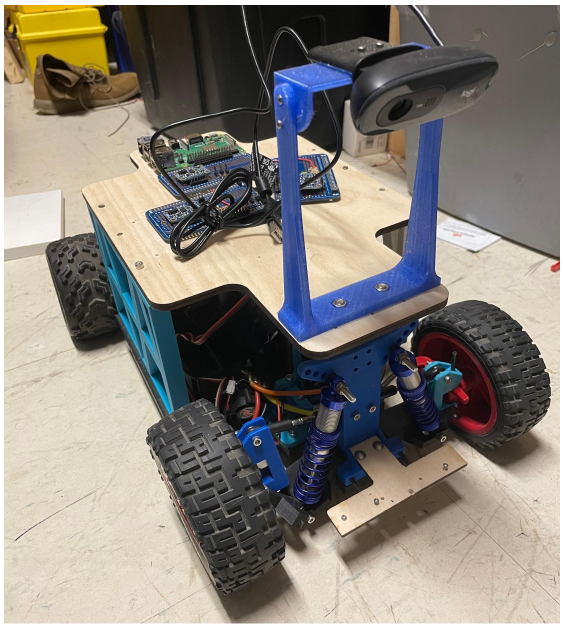


Figure 6.2.1: A furbished PrePARV

The Amazon Cars, being in better shape than the others, were focused on next. Their main hindrance was the lack of RC car receivers to control their steering, mainly for troubleshooting and field testing. The team had access to a storage closet that stored miscellaneous pieces, electronics, cables, and even the large sections of components of other cars. Eventually, five RC receivers and compatible controllers were located, which required some tweaking to connect the cars' electronic speed controllers. However, as soon as they could be driven for testing, other problems began showing themselves. Not only were many of the old lipo batteries used to power the cars inflated and therefore unsafe, but Amazon Car 1's steering servo was nonfunctional, keeping it from turning while driving. Some cars were improperly wired, connecting the lipo batteries and the motor with exposed wires and duct tape. The required T-Plug to Tamiya connectors were located in the storage closet and replaced before troubleshooting resumed, and replacement lipo batteries were also acquired. A new SG90 steering servo for Amazon Car 1 was installed, and as the cars began to enter a state where they were suitable for field tests, new cameras and sensor mounts were 3D printed and installed on all four cars.

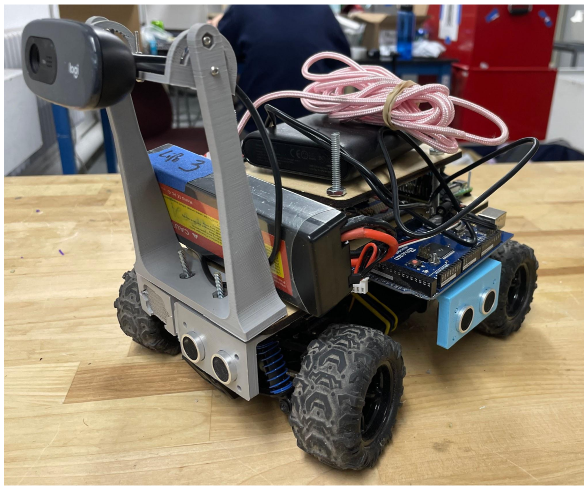


Figure 6.2.2: A completed Amazon Car 2

The new camera mounts were designed by the current team, with the mounts for the ultrasonic sensors and lidar sensor designed by a team that worked on the project in the summer of 2022. Figure 6.2.2 shows the new mounts installed on Amazon Car 2, 3D printed in gray PLA. Figure 6.2.3 shows the models of the ultrasonic and lidar mounts within the modeling program SolidWorks.

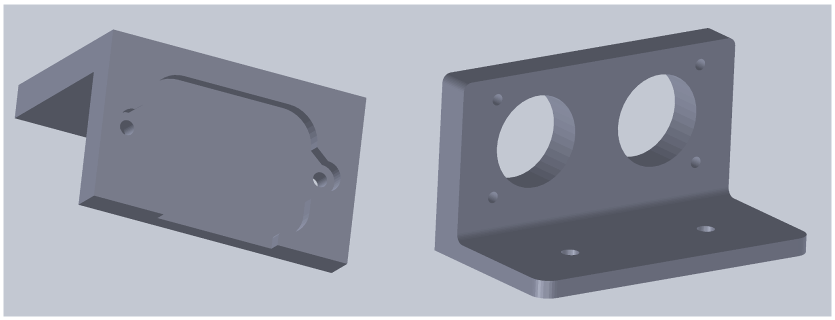


Figure 6.2.3: The LiDAR and ultrasonic sensor models

6.3 Amazon Car 4

Amazon Car 4 was the pre-built car that required the most components. The other three cars had sensors, a breadboard, and old camera mounts. This car required essential components; a new car bed, an Arduino and Raspberry Pi, and every sensor required to run the software package. Its Electronic Steering Controller (ESC) was also nonfunctional, rendering it unable to drive. Due to the motor being brushless rather than brushed, the new ESCs ordered for the other cars were incompatible, and a new ESC had to be installed. Due to the differing connection lengths, the old ESC wires were soldered onto the new controller.

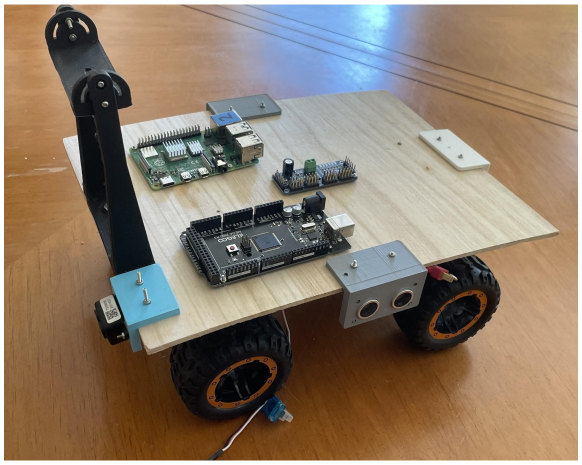


Figure 6.3.1: A refurbished Amazon Car 4

The new car bed was made out of balsa wood, a new Raspberry Pi, Arduino, and a servo shield installed alongside newly printed ultrasonic sensor holders. The new pieces can be seen in Figure 6.3.1, along with a new, 3D-printed camera mount.

6.4 Donkey Car

DonkeyCar was a more straightforward fix due to the main complaint being regarding its lack of securing mechanism for the top bed. The bed was kept from sliding down with specialty metal spokes, but the same effect could be achieved with study paperclips, as they were the same thickness and not much material was required to keep the bed from sliding back up, mainly when picked up. Unfortunately, after some testing, issues with its ESC also arose; it would not connect to any receiver and be firmly adhered to the car bed. Due to being unable to be removed, a second brushed ESC was placed in the car bed, and the first's wires were wrapped and concealed to avoid confusion (as shown below in Figure 6.4.1).

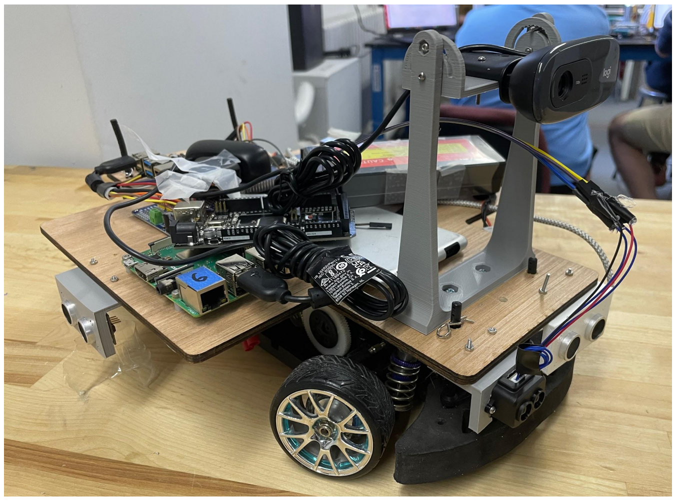


Figure 6.4.1: A completed DonkeyCar

6.5 PARV

PARV had its good wheel taken (Figure 6.4.1) and used for its twin PrePARV; repairs began when the new components arrived. Ball bearings, an axle, and a new camber link were installed.

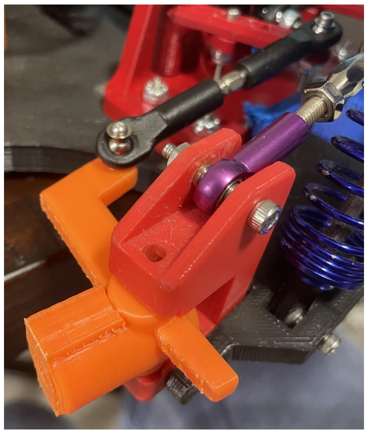


Figure 6.5.1: PARV's new camber link and axle

The cracked frame was replaced, and a camera mount was installed along with the ultrasonic sensors, as seen in Figure 6.5.2.

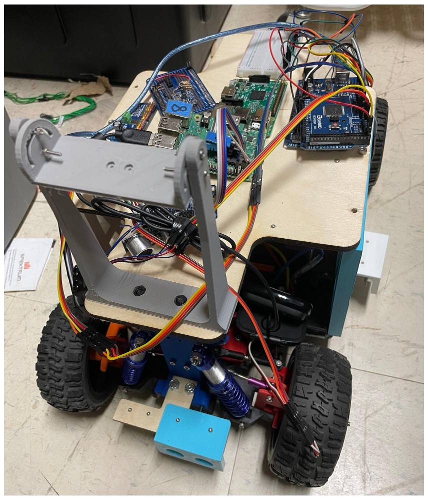


Figure 6.5.2: A refurbished PARV

6.6 Cleveland

Cleveland was another student-made 3D printed car, its issue being a broken frame, breaking the connection to the steering linkage. This breakage left its front left wheel sliding in the open air, unable to rotate properly. Due to the damage relative to the size of the car's base, reprinting the entire piece could have been more practical. Instead, a supplemental piece was designed and printed to be screwed into the base, wrapping around the wheel axle to keep it in place.

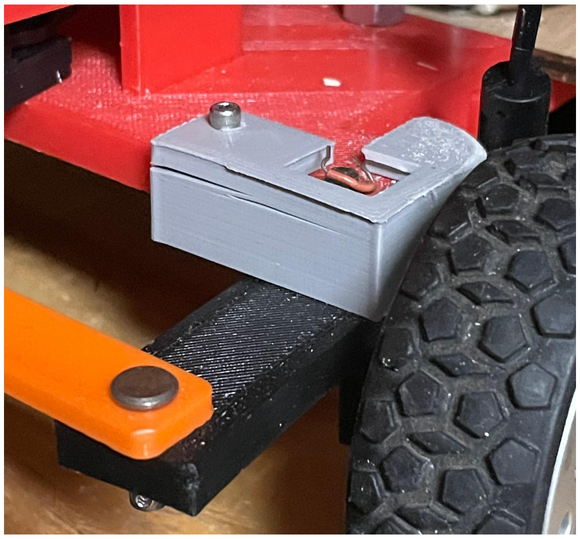


Figure 6.6.1: Cleveland’s supplemental axle piece

The new piece can be seen in Figure 6.6.1, secured to the car base. Additionally, a Raspberry Pi and camera mount were installed to ensure full functionality (Figure 6.6.2).

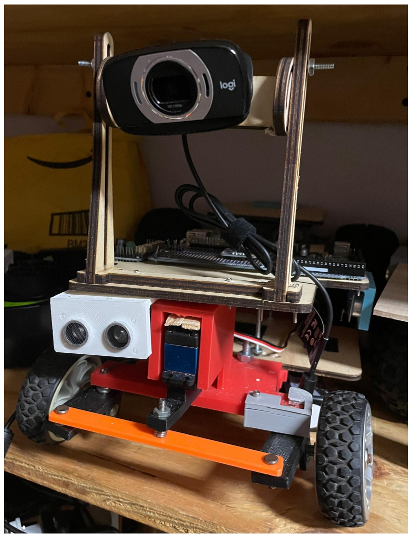


Figure 6.6.2: A completed Cleveland

6.7 Sensor Implementation

This section talks about all sensors that are being used in this project. It concludes various components sent to the dashboard for users to see and understand.

6.7.1 Arduino Mega

Each car has one Arduino Mega (Figure 6.7.1.1) as it is necessary to gather sensor data. All wires are plugged into the Arduino to send data from the sensors to the Arduino. There is code written on the Arduino that is used to pull the values from each pin and send these values to the Raspberry Pi to display that data on the dashboard.

Figure 6.7.1.1: Arduino Mega, Reproduced as is from [19]

6.7.2 Ultrasonic

There is four ultrasonics (Figure 6.7.2.1) on each car, one on each side of the chassis. These measure the distance to a target by measuring the time between emissions and receptions. Each ultrasonic produces a single value used to display on the dashboard. There is only one graph for all of the ultrasonic sensors, so each has its line on the graph.

Figure 6.7.2.1: Ultrasonic sensor, reproduced as is from [20]

6.7.3 TF Luna

There is one TF Luna (Figure 6.7.3.1) in the front of the car. A TF Luna is a small, light, and low-power LIDAR (Light Detection and Ranging) sensor that uses time-of-flight (TOF) technology to detect distances to objects. It emits a pulsed laser beam and measures the time taken for the beam to return after reflecting off an object. The distance to the object can be calculated based on the time taken. These values are calculated every second and displayed on the dashboard.

Figure 6.7.3.1: TF Luna sensor, reproduced as is from [21]

6.7.4 IMU

There is one IMU (Figure 6.7.4.1) sensor on each car, placed on the breadboard. These sensors measure and report a body's specific force, angular velocity, and sometimes magnetic field using a combination of accelerometers, gyroscopes, and magnetometers. This project only uses the accelerometer and gyroscope values to display on the dashboard. There are two graphs for the IMU Data, one for the accelerometer values and one for the gyroscope.

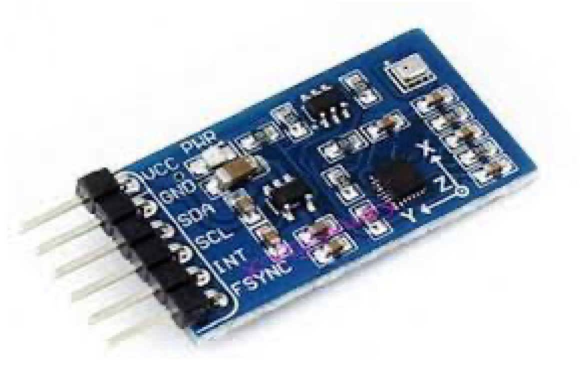


Figure 6.7.4.1: Inertial Measurement Unit(IMU) sensor, reproduced as is from [22]

6.7.5 Conclusion

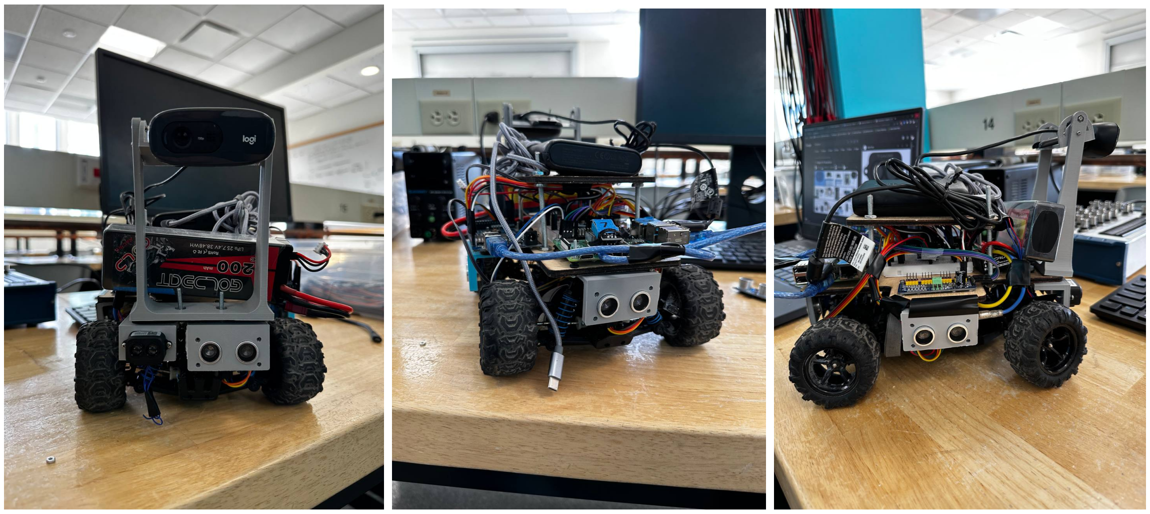


Figure 6.7.5.1: Amazon Car 4 with all sensors around the car

Overall, all these sensors allow further information on the driving process of the car that is displayed on the dashboard. All these sensors are connected to the Arduino Mega, which sends signals to the Raspberry Pi. The Pi will then parse this information and seamlessly upload it to the dashboard throughout the driving process. Figure 6.7.5.1 shows the car's final look with all the sensors mounted along with the finished wiring.

7. Test Track

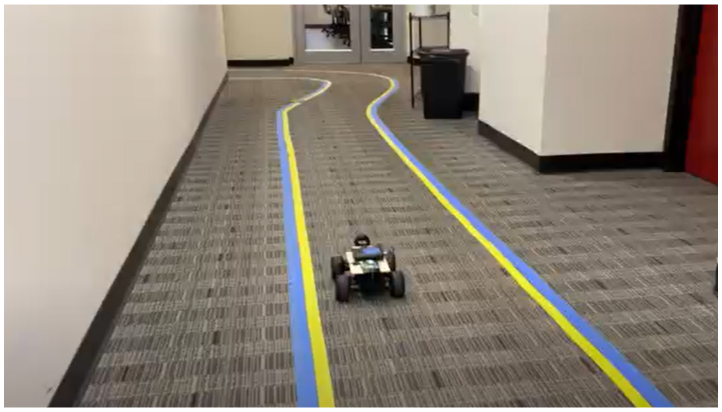


Figure 7.1: Taping of the track at the start of the term

At the start of the term, the track had a taping of blue and yellow on the sides (as seen in Figure 7.1). This taping has been the same since the start of 2022 and needs a significant change. The tape started slowly falling apart, and the track had holes and spaces. This affected the driving process of the car quite a bit as the car did not have a sense of continuation of the sides of the track thus, at times, the car did not know where the edge of a track was. The original dimensions for the width of the track were about 50 - 75cm [17].

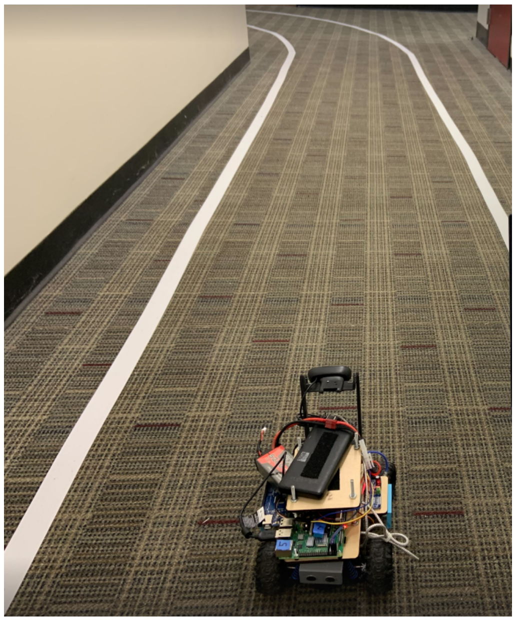


Figure 7.2: New taping was done to make a continuous lane

As seen in Figure 7.2, the team had decided to change the taping to one concise color, white. This would make the track look much neater and more evident to the camera. It could easily be more distinguished from the carpet color or even the walls color due to the saturation of the color itself. Using this taping was a significant investment as now the car had a much better understanding of the flow of the track and could detect the lines more clearly overall.

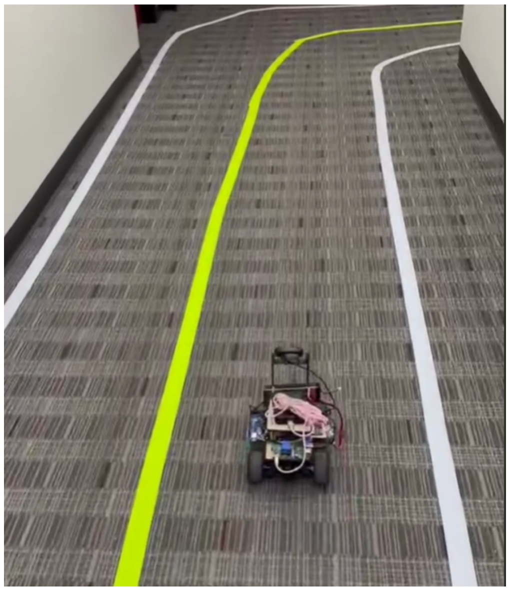


Figure 7.3: Final look of the track

As the team slowly progressed through the objectives, a more extensive track was demanded. Having two cars simultaneously on the same track would be very hard with the initial track. Both cars would fight for space and mess up the driving process for both models. Thus, the team extended the track, even more, to give a more extensive surface area overall. Including more space, a divider was placed to represent a real-world road situation for the cars to follow (as seen in Figure 7.3). This allowed for two different traffic flows to drive simultaneously and not interfere with one another. The dimensions of this new track are about 127 - 160 cm long. This is the dimension for the entire track. Remember that two cars run simultaneously on the track thus, approximately each car gets about 63 - 80 cm for itself to drive on.

8. Self-Driving Implementation

This chapter will comprehensively examine the implementation of various self-driving algorithm concepts. These concepts include video capture, lane following, and lane detection techniques. As autonomous driving is a fundamental aspect of this project, the implementation must be clearly articulated and readily testable. Two different approaches were used and tested to see the advantages and disadvantages. The driving code has been exclusively developed using Python3 programming language.

8.1 Components Used

The following items are employed in implementing autonomous driving. Please refer to the Setup Guide in Appendix to see the circuit and wiring:

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

8.2 Driving Algorithm

In this section, two different approaches are talked about to approach the self-driving algorithm. The first is Husky Rover, while the next is an open-source program called Donkey Car. Ultimately, the team chose Donkey Car due to its efficiency.

8.2.1 Husky Rover Approach

HuskyRover is an autonomous RC car that uses HuskyLens to drive around a track. The track, in this case, is a circle defined by a singular line in which the car tracks to stay on course. Unlike DonkeyCar and last year's mPAD, this system only uses Arduino instead of Raspberry Pi or both (as seen below in Figure 8.2.1.1). The main asset of the car is the HuskyLens which is used to follow the line around the track. Due to the use of HuskyLens on this car, it can also follow behind an object without crashing into it. In addition to the HuskyLens, the car uses an RC controller, motor controller, and a steering servo. The system can experience 300+ frames per second for line detection, allowing quick responses when taking turns.

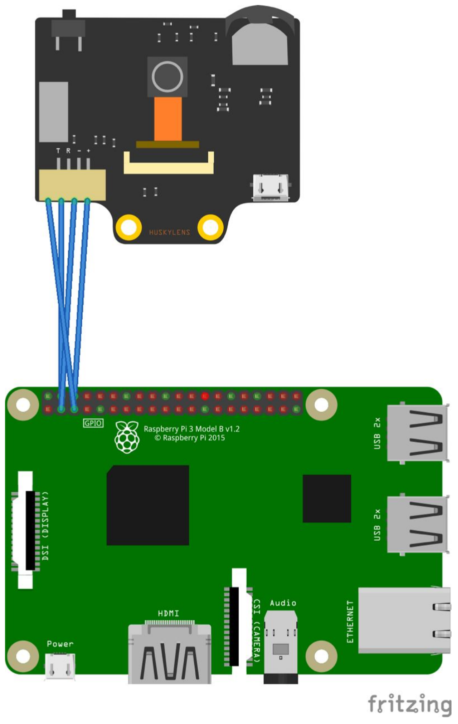


Figure 8.2.1.1. HuskyLens connection to Raspberry Pi

The unfortunate thing about HuskyLens is that it can not be directly programmed. This means there is no perspective correction or the ability to track two different colored lines.

However, it can easily interface with Arduino or a Teensy-based board to do work like reading RC or driving servos. It also has built-in neural network AI programs for object detection, color detection, tags, faces, and gestures. Something else to note is that this system does not consume as much power as OpenCV does.

8.2.1.1 Video Capture

The HuskyLens visual AI sensor is equipped with a built-in camera capable of capturing video through a command sent to the sensor via its microcontroller interfaces, such as an Arduino or Raspberry Pi. Upon receipt of this command, the sensor initiates the video capture process at a specified frame rate, typically at 30 frames per second (as seen below in Figure 8.2.1.1.1). Figure 8.2.1.1.1 shows one of the Amazon Cars using HuskyLens, in which the image is displayed on a back-facing screen.

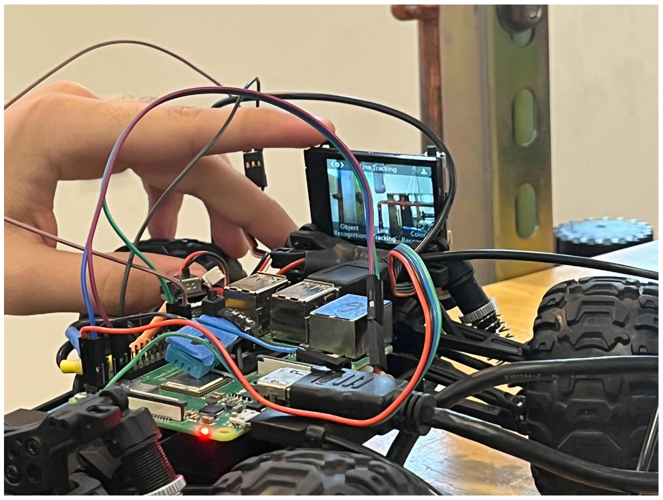


Figure 8.2.1.1.1: Amazon Car using the HuskyLens camera

The captured video frames are then processed by the sensor's onboard image processing algorithms, which are responsible for detecting and recognizing objects, facial expressions, hand gestures, and speech commands within the video frames. Additionally, image enhancement techniques, such as color correction or noise reduction, may be applied to improve the quality of the video frames. The processed video frames can be transmitted to the microcontroller or to the cloud for further analysis or storage. Additionally, the sensor can be configured to send notifications or trigger actions based on detecting specific objects, facial expressions, hand gestures, or speech commands.

In conclusion, the HuskyLens visual AI sensor has a built-in camera to capture video through a command sent via its microcontroller interface. The captured video frames are processed by the sensor's onboard image processing algorithms that process the captured video frames to detect and recognize objects, facial expressions, hand gestures, and speech commands. The processed video frames can be sent to the microcontroller or cloud for further analysis or storage. Additionally, the sensor can be configured to send notifications or trigger actions based on detecting specific objects, facial expressions, hand gestures, or speech commands.

8.2.1.2 Lane Detection

HuskyLens is equipped with a built-in camera that allows it to capture video and process it to detect lanes on the road. The sensor's onboard image processing algorithms perform the lane detection process. These algorithms use edge detection, color space transformation, and Hough transform to identify the lanes in the video frames.

The first step in lane detection is to convert the video frames from the RGB color space to a different color space, such as HLS or HSV, that separates the color information from the brightness and saturation information. This step allows the sensor to more easily identify the lanes based on their color, as they are typically painted in white or yellow. Next, the sensor applies edge detection algorithms to identify the edges of the lanes in the video frames. The sharp contrast between the color of the lanes and the road typically defines the edges. The sensor uses a Canny edge detection algorithm to identify these edges. Once the edges are identified, the sensor applies the Hough transform to detect the lines corresponding to the lanes. The Hough transform converts the edge points into a parameter space where the lines are represented as points. By finding the intersection of the lines in this parameter space, the sensor can identify the lanes. Finally, the sensor uses the information about the lanes to create a road map and calculate the vehicle's position in relation to the lanes. The vehicle's control system can use this information to plan and execute safe and efficient trajectories (as seen below in Figure 8.2.1.2.1). Figure 8.2.1.2.1 shows HuskyLens doing line tracking, as seen by the blue arrow following the white tape line of the track.

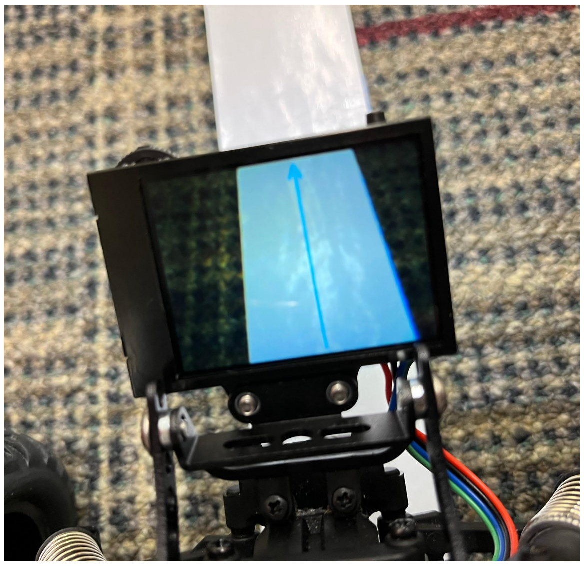


Figure 8.2.1.2.1: HuskyLens line detection

In summary, HuskyLens uses its built-in camera to capture video and its onboard image processing algorithms to perform lane detection by first converting the video frames to a different color space, then applying edge detection algorithms, and finally using the Hough transform to detect the lines that correspond to the lanes. The sensor then uses this information to create a road map and calculate the vehicle's position in relation to the lanes, which the vehicle’s control system can use to plan and execute safe and efficient trajectories. This lane detection feature is crucial for autonomous navigation and helps the vehicle to stay on the correct path and avoid collisions.

8.2.1.3 Lane Following

The lane detection feature of the HuskyLens visual AI sensor can be utilized to implement lane following functionality within autonomous vehicles. This process involves the utilization of the sensor's lane detection capabilities to identify the lanes on the road, as well as the position of the vehicle in relation to those lanes. The vehicle's control system can then utilize this information to plan and execute safe and efficient trajectories.

The initial step in implementing lane-following functionality is integrating the HuskyLens sensor into the vehicle's control system. This integration process involves connecting the sensor to a microcontroller, such as an Arduino or a Raspberry Pi, which can be utilized to control the vehicle's movement. Once the sensor is integrated, it can capture video of the vehicle's environment and process it to detect the lanes on the road. The sensor's onboard image processing algorithms can then be used to calculate the vehicle's position in relation to the lanes, providing the necessary information for the control system to plan and execute safe and efficient trajectories. Additionally, the sensor can send notifications or trigger actions to detect specific lane conditions, such as a lane departure or a traffic sign, allowing the control system to respond accordingly. Utilizing the HuskyLens sensor's lane detection capabilities allows for the implementation of lane-following functionality within autonomous vehicles. When integrated into the vehicle's control system, this feature enables the vehicle to navigate safely and efficiently by providing real-time information about the vehicle's position in relation to the lanes. The control system can then use this information to plan and execute safe and efficient trajectories, thus enhancing the overall performance and safety of the autonomous vehicle. Furthermore, the sensor's capability to send notifications or trigger actions based on detecting specific lane conditions, such as a lane departure or a traffic sign, allows for an additional layer of safety and adaptability within the autonomous vehicle's navigation system.

Unfortunately, the team could not successfully implement and fully test the Husky Rover. The main issue encountered was the limited field of view of the camera lens, which was only about 72 degrees. As a result, the camera had difficulty detecting lines on both sides of the vehicle. Due to this limitation, the team decided to abandon this implementation approach.

8.2.2 Donkey Car Approach

As previously mentioned, Donkey Car is an open-sourced self-driving platform for small-scale robotic vehicles. It works by combining hardware and software components to create a self-driving vehicle. This is built on Python and uses deep neural networks to train an RC car autonomously. Users manually drive the vehicle around a track, collecting data from the camera and sensors to train the neural network. The web-based interface allows users to deploy the model to the car for autonomous driving. There are many steps regarding collecting this data and processing the camera information along with speed/throttle values (as seen below in Figure 8.2.2.1). Each step is essential overall to create an ideal model with the functionality of staying between 2 lines (acting like lanes) and following them appropriately. Once the model is trained and the car starts to drive autonomously, each image captured is sent to the preprocessing model where hough line transformation is applied, the throttle and steering are calculated from the model, and then the throttle and steering are sent back to the car so that the model can update.

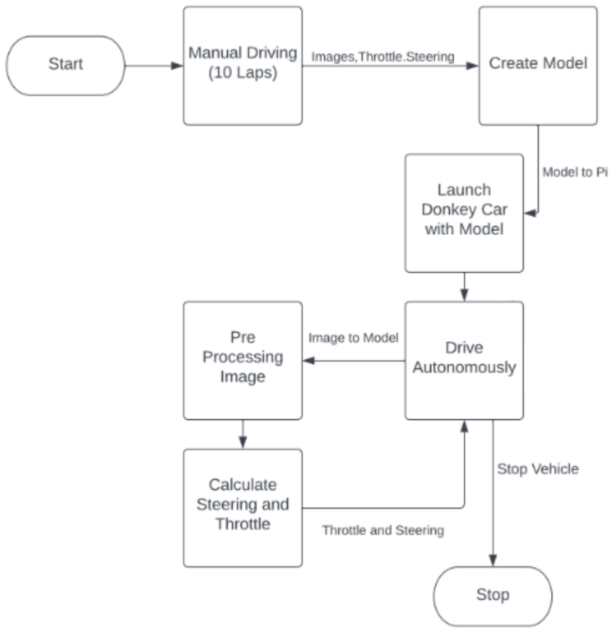


Figure 8.2.2.1: Donkey Car and pre-processing flow chart

The following sections expound upon the intricacies of Figure 8.2.2.1, which portrays the sequential progression of the Donkey Car's underlying mechanics during its autonomous operation. This flowchart elucidates the various processes that occur during the car's autonomous driving, providing a comprehensive overview of the system's functionality.

8.2.2.1 Video Capture

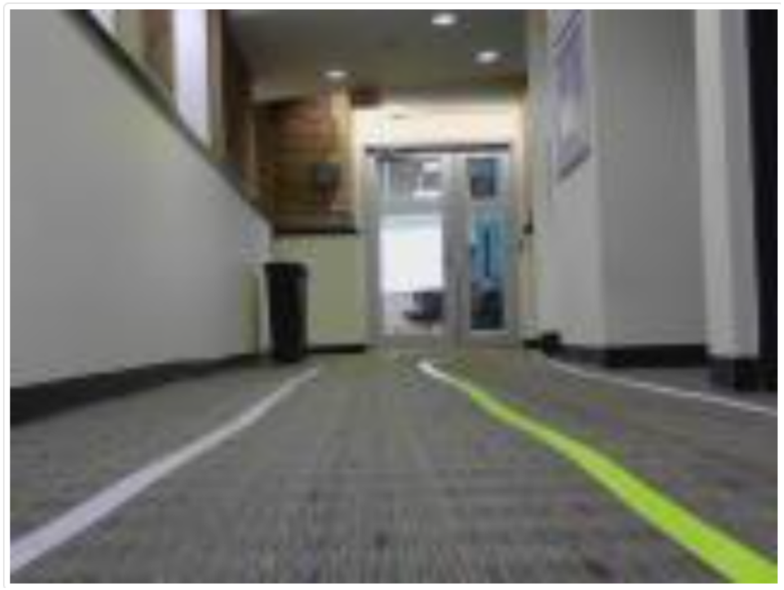


Figure 8.2.2.1.1: Webcam image of what the car perceives

Once the webcam is successfully initialized, the DonkeyCar platform command line continuously captures images of what the car sees in real-time (as seen in Figure 8.2.2.1.1). This loop is crucial as it allows the autonomous vehicle to update its understanding of the environment and react accordingly constantly. In each iteration of the loop, the platform retrieves a frame from the webcam using the Pygame camera object, which is then preprocessed to ensure compatibility with the subsequent stages of the pipeline. Preprocessing may include resizing the image, converting the color space, or applying normalization techniques to improve the system's performance.

After a standard image has been captured, it is passed on to the subsequent processes. This typically involves feeding the image to a machine learning model, such as a convolutional neural network (CNN), responsible for detecting and recognizing various features and objects within the frame. The autonomous driving system then uses this information to make decisions based on the current environment, such as steering angle adjustments, acceleration, and braking.

In conclusion, the video capture mechanism in the DonkeyCar platform plays a critical role in the overall functionality of the autonomous driving system. By leveraging the Python programming language and the Pygame library, the platform can efficiently access the webcam, capture images, preprocess them, and pass them on to the processing pipeline. This streamlined process enables the autonomous vehicle to continuously update its understanding of the environment and make informed decisions, ultimately contributing to the system's successful operation.

8.2.2.2 Image Pre-processing

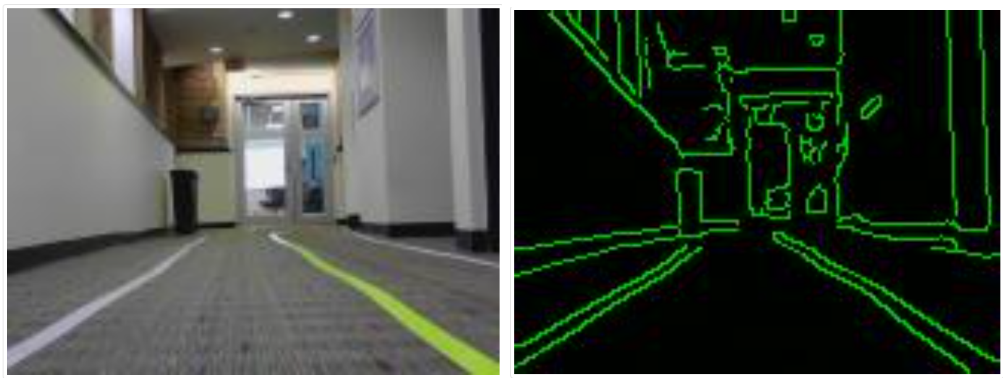


Figure 8.2.2.2.1: Before and after images of pre-processing

In the development of autonomous driving systems, the quality of the input data is of utmost importance, as it directly impacts the performance and accuracy of the decision-making process. In the context of the DonkeyCar platform, pre-processing techniques are applied to the captured images to enhance their features and improve the system's overall functionality. This section will discuss the pre-processing steps implemented in the DonkeyCar platform, focusing on the grayscale conversion, Gaussian blur application, Hough line transform, and the addition of noise. Figure 8.2.2.2.1 presents a comparative analysis between an unprocessed image and its counterpart subjected to a preprocessing algorithm. The image demonstrates a notable enhancement in the visual clarity of the borders surrounding the objects in the processed image. This effect allows for easier and more precise identification and analysis of the objects within the image. Such preprocessing algorithms are valuable tools for various fields, including computer vision, machine learning, and image processing.

Initially, the DonkeyCar platform did not include pre-processing steps in its pipeline. However, recognizing the potential benefits of applying image pre-processing techniques, the platform has since incorporated enhancements to the captured images. The first of these enhancements involves converting the images to grayscale. This simplifies the image data by reducing the color channels to a single channel, allowing the machine-learning algorithm to focus on essential features rather than color information.

Subsequently, a Gaussian blur is applied to the grayscale image to reduce high-frequency noise and smooth the image. This technique helps suppress unwanted details and highlights the more prominent features, such as edges and lines, which are crucial for the machine learning algorithm to make accurate predictions.

Following the Gaussian blur, a Hough line transform is applied to extract lines from the image, changing the rest of the image to black and making the lines green (as shown in the Figure above). This transformation is beneficial in scenarios where the autonomous vehicle needs to detect and follow lane markings or other linear structures in the environment. The machine learning algorithm can better recognize and interpret these features by isolating and emphasizing these lines. Finally, to further improve the machine learning algorithm's performance, noise with pixel values between 1 and 5 is added to the image to improve the machine learning algorithm's performance further. Although these pixels help the machine learning algorithm avoid learning incorrect patterns, they are naked to the human eye. Introducing this noise can help the algorithm become more robust and generalize better. It encourages the model to focus on detecting patterns in the data rather than relying on pixel-perfect representations.

Implementing pre-processing techniques in the DonkeyCar platform has significantly enhanced the input data quality for the machine learning algorithm. By converting images to grayscale, applying a Gaussian blur, utilizing the Hough line transform, and adding noise, the platform ensures that the captured images are better suited for pattern detection and feature extraction. These enhancements ultimately contribute to the improved performance and accuracy of the autonomous driving system.

8.2.2.3 Lane Detection

Autonomous driving systems have experienced significant advancements in recent years, with the ability to perceive and navigate their environment being of paramount importance accurately. Lane detection is a key aspect of this navigation process, which helps ensure the vehicle remains within designated lanes and maintains safe distances from surrounding obstacles. This section of the research paper will discuss the lane detection mechanism utilized in the DonkeyCar platform, specifically focusing on how the linear neural network (LNN) identifies lines on both sides of the car.

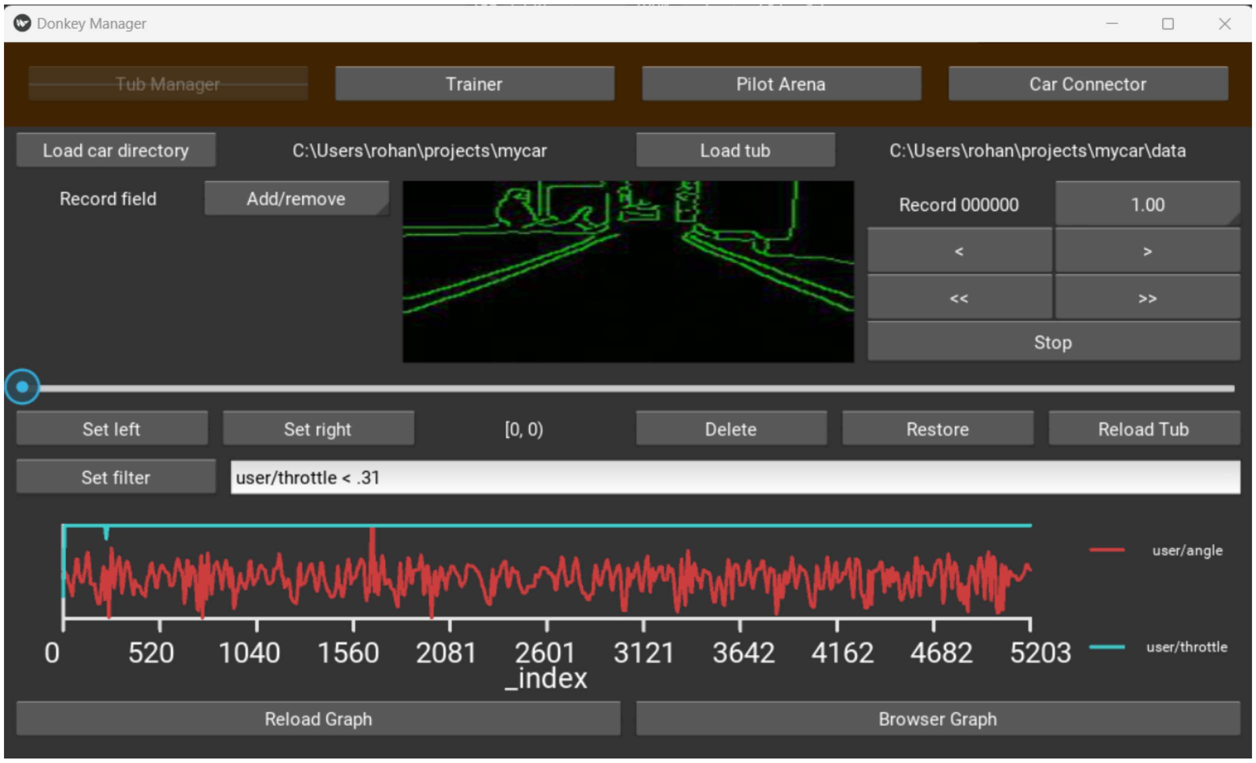


Figure 8.2.2.3.1: Donkey car model training ui

After training for a set amount of laps, the user generates a model using the command line tools of DonkeyCar. This model (as seen in Figure 8.2.2.3.1) then trains on the records or images and the throttle and steering values for each frame. The model learns patterns and, from these patterns, can drive autonomously based on past historical data. The model finds patterns by identifying lanes and calculates the steering and throttle needed to continue driving autonomously. This graphical interface was discussed earlier in the background information.

Lane detection is a critical component of autonomous driving systems. It enables the vehicle to navigate accurately within its environment and maintain its position within the designated lanes. In the DonkeyCar platform, a linear neural network (LNN) is employed to identify the lines on both sides of the car, ensuring safe and controlled navigation. The LNN operates by processing the pre-processed images and identifying patterns and features corresponding to the lane markings. This is achieved through multiple layers of convolutional and pooling operations, which successively reduce the spatial dimensions of the input image while increasing its depth (i.e., the number of feature maps). As the image passes through the network, the LNN learns to recognize and differentiate between various line shapes, orientations, and curvatures, ultimately identifying the left and right lane markings.

Once the LNN has detected the lane markings, the autonomous driving system can use this information to determine the vehicle's position within the lanes and adjust as needed. For instance, the system may adjust the steering angle to maintain the vehicle's position within the center of the lane or apply the brakes if it detects a potential collision risk. By effectively detecting and tracking the lanes, the LNN-based lane detection system in the DonkeyCar platform plays a crucial role in ensuring the safety and efficiency of the autonomous vehicle's navigation.

In summary, the lane detection mechanism employed by the DonkeyCar platform is instrumental in facilitating the safe and accurate navigation of the autonomous vehicle. Utilizing a convolutional neural network, the platform can effectively identify and track the left and right lane markings, allowing the vehicle to make informed decisions based on its position within the designated lanes. As a result, the LNN-based lane detection system contributes to the overall functionality of the DonkeyCar platform but also serves as a vital component in developing safe and reliable autonomous driving technologies.

8.2.2.4 Lane Following

The effective integration of the components and processes discussed in previous sections is critical for successfully operating an autonomous driving system, such as the DonkeyCar platform. In this section, the team will explore how the captured and pre-processed images and the lane detection mechanism based on the convolutional neural network (CNN) combine to enable the autonomous vehicle to follow its designated lane accurately and safely.

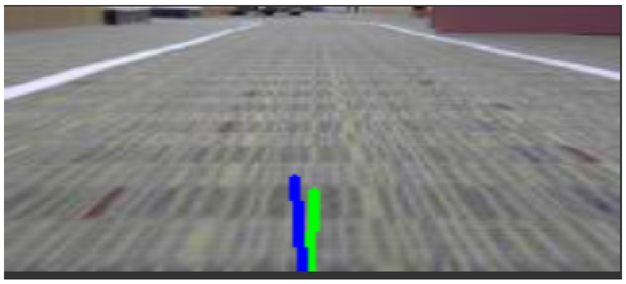


Figure 8.2.2.4.1: Shows the model's reaction to how it will drive on the track

Once the pre-processed images have been fed through the linear regression model with convolution layers and the lane markings have been identified, the system proceeds to extract essential information about the vehicle's position within the lanes and the curvature and orientation of the lane markings. This data is then used to compute the necessary steering angle adjustments and speed changes required to maintain the vehicle's position within the center of the lane while adhering to any changes in the road's curvature.

The steering angle adjustments and speed changes are determined by a control algorithm, which considers factors such as the distance between the detected lane markings, the vehicle's position relative to the center of the lane, and the curvature of the lane. Figure 8.2.2.4.1 depicts two color-coded lines, with each color representing either a human control or an AI control. The green line indicates the manual driving process of a human compared to the blue line, which signifies the model's understanding of how to drive on the track. In the Figure above, it can be seen that the blue line (the AI) is tilting a bit to the left on a straight path compared to the green line (human driving) staying at the center of the track. By analyzing these factors, the control algorithm can generate appropriate steering commands and throttle adjustments, which are then passed on to the vehicle's actuators.

8.3 Conclusion

To ensure a smooth and reliable operation, the entire process of capturing images, pre-processing, lane detection, and control command generation occurs in a continuous loop, with the system constantly updating its understanding of the environment and adjusting its actions accordingly. This real-time responsiveness allows the autonomous vehicle to adapt to changes in the driving environment, such as new lane markings, road curvature, and obstacles, resulting in a safer and more efficient driving experience.

In conclusion, integrating image capture, pre-processing, lane detection, and control algorithms is crucial for successfully operating the DonkeyCar platform's autonomous driving system. By effectively combining these components, the system can accurately follow its designated lane, adapt to changing road conditions, and ensure the safety and efficiency of the vehicle's navigation. This holistic approach highlights the importance of considering each aspect of the autonomous driving pipeline and serves as a valuable reference for the ongoing development of autonomous driving technologies.

9. Object Detection

Object detection is a crucial aspect of autonomous driving systems. It enables vehicles to perceive and understand their surroundings, ultimately making informed decisions based on the identified objects. Various methods and optimizations have been researched and implemented to achieve the highest possible frame rate for real-time object detection. Specific research incorporating object detection on raspberry pi has been more of interest in real years for various purposes such as biomedical devices [2]. This section will discuss the different object detection methods and optimizations, including using TensorFlow API, TensorFlow Lite, custom TensorFlow models, parallel processing, and multi-device processing (as shown in the flowchart in Figure 9.1).

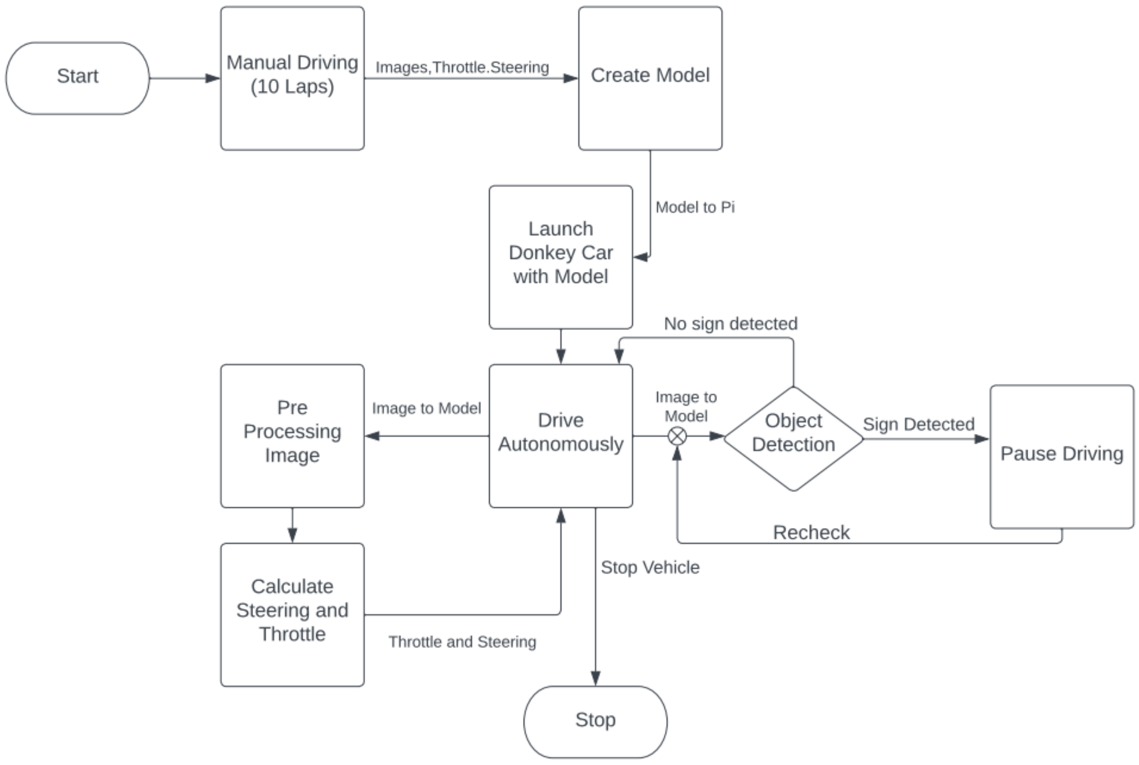


Figure 9.1: Driving logic flowchart with stop sign detection

Figure 9.1 depicts the flow chart illustrating the implementation of stop sign detection, which was incorporated into the overall backend of the code. Prior to this, the flowchart detailing the driving functionality was presented, highlighting the intersection between the stop sign detection logic and the rest of the system. Subsequent sections elaborated on the interconnections between the various components.

9.1 TensorFlow API

TensorFlow

Figure 9.1.1: TensorFlow logo, reproduced as is from [10]

The TensorFlow API (Figure 9.1.1) was initially utilized for object detection, achieving an average frame rate of 4 FPS. While the API offered a reliable and accessible method for detecting objects, the frame rate was considered suboptimal for real-time applications [5]. The TensorFlow API provides pre-trained models that can detect various objects [6]. However, the complexity and size of these models can limit the processing speed, especially on devices with limited computational resources.

9.2 TensorFlow Lite

Figure 9.2.1: TensorFlow lite logo, reproduced as is from [10]

To improve the frame rate, TensorFlow Lite (Figure 9.2.1) was employed as a lighter and faster alternative to the TensorFlow API [6]. The use of TensorFlow Lite resulted in an increased frame rate of five FPS, demonstrating a modest improvement in object detection performance. TensorFlow Lite is explicitly designed for mobile and edge devices, offering smaller models and faster processing times. This performance improvement is achieved by utilizing efficient quantization techniques and model optimization methods that reduce the model size and increase the processing speed without significantly affecting accuracy.

9.3 Custom TensorFlow Model

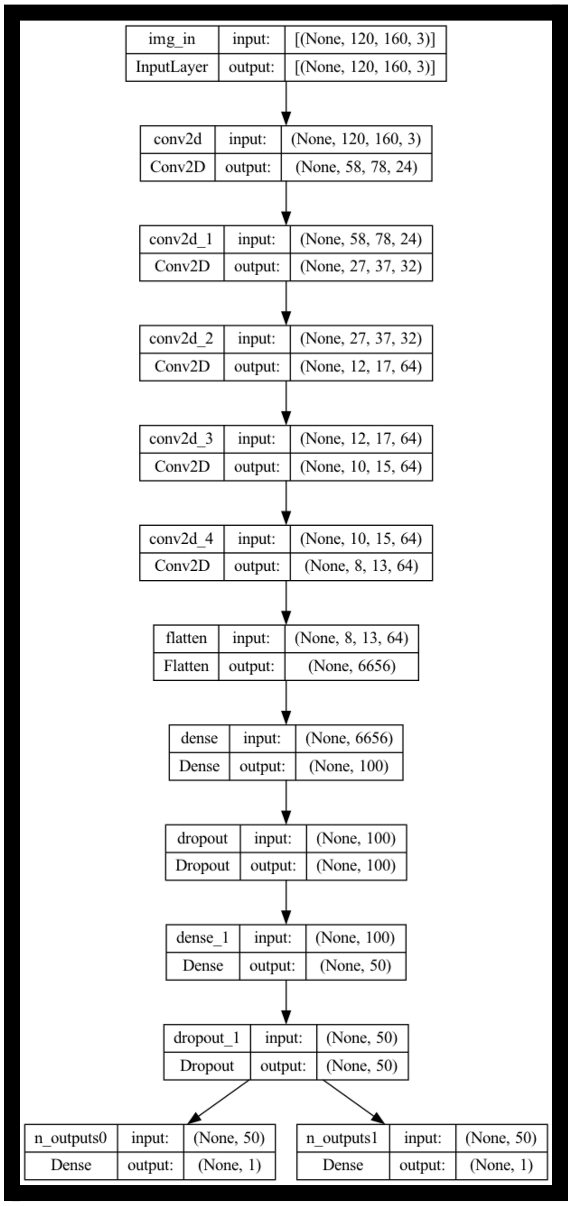

img_in | input: | [(None, 120, 160, 3)]
InputLayer | output: | [(None, 120, 160, 3)]
conv2d | input: | (None, 120, 160, 3)
Conv2D | output: | (None, 58, 78, 24)
conv2d_1 | input: | (None, 58, 78, 24)
Conv2D | output: | (None, 27, 37, 32)
conv2d_2 | input: | (None, 27, 37, 32)
Conv2D | output: | (None, 12, 17, 64)
conv2d_3 | input: | (None, 12, 17, 64)
Conv2D | output: | (None, 10, 15, 64)
conv2d_4 | input: | (None, 10, 15, 64)
Conv2D | output: | (None, 8, 13, 64)
flatten | input: | (None, 8, 13, 64)
Flatten | output: | (None, 6656)
dense | input: | (None, 6656)
Dense | output: | (None, 100)
dropout | input: | (None, 100)
Dropout | output: | (None, 100)
dense_1 | input: | (None, 100)
Dense | output: | (None, 50)
dropout_1 | input: | (None, 50)
Dropout | output: | (None, 50)
n_outputs0 | input: | (None, 50)
Dense | output: | (None, 1)
n_outputs1 | input: | (None, 50)
Dense | output: | (None, 1)

Figure 9.3.1: Custom cnn model of stop sign detection

To further enhance the frame rate, a custom TensorFlow model was developed and implemented. Through multiple iterations, the team tailored a model that fulfills the requirements of being at least 84% accurate and has the capabilities of running at least 5 frames a second.

Figure 9.3.1 shows the final tailored model was a multi-layer CNN model with several convolution layers. This custom model was tailored to the specific requirements of the autonomous driving system and achieved a higher frame rate of 6 FPS, indicating a more suitable solution for real-time object detection. The custom model focused on streamlining the architecture and reducing the number of layers and parameters, allowing for faster processing while maintaining acceptable levels of accuracy. The custom model provided an optimal balance between performance and complexity by focusing on detecting objects particularly relevant to the autonomous driving context.

9.4 Parallel Processing

Significant improvements in the frame rate were achieved by incorporating parallel processing techniques. Parallel processing was implemented using multi-threading or multi-core processing which allowed the main core program and object detection to run concurrently. This system reached an impressive frame rate of 20 FPS, substantially increasing the efficiency and responsiveness of the object detection process. This approach allowed the system to fully utilize the computational capabilities of the hardware, distributing the workload evenly across multiple processing units and effectively reducing the time required for processing individual frames.

9.5 Optimizations

Several optimizations were applied to the existing TensorFlow model to improve its performance. The model was optimized using Numpy, a popular Python library for numerical operations and data manipulation. Replacing the original operations with more efficient numpy functions reduced the model's processing time, leading to improved performance.

Additionally, research was conducted to explore more memory-optimized methods for saving images, such as using more efficient compression techniques or discarding irrelevant data. These optimizations further enhanced the system's overall performance, enabling it to process and store images more quickly and with reduced memory usage.

9.6 Multi-Device Processing

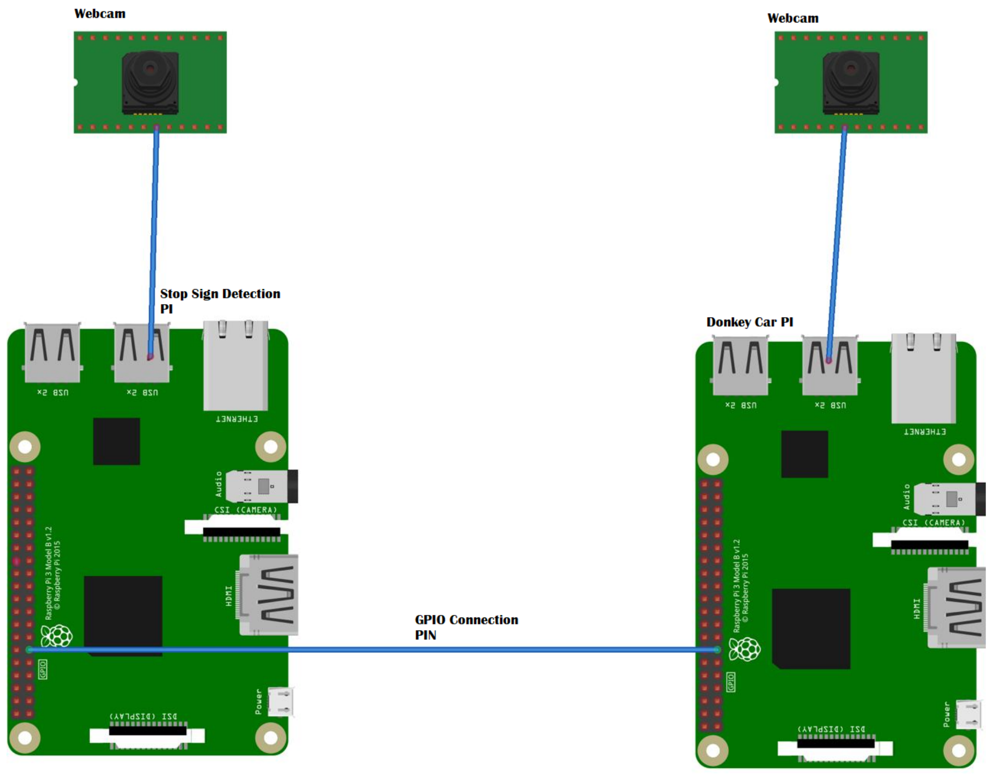


Figure 9.6.1: Double raspberry pi to show parallel processing

To overcome the limitations of a single device's CPU capabilities, an interface was created between two Raspberry Pi devices (as seen in Figure 9.6.1). This was necessary since the cpu load of the pi with both object detection and donkey car was causing a constant crash of the system.. This configuration allowed one device to focus solely on stop sign detection while the other managed the core DonkeyCar operations. This multi-device processing approach maximized the available computational resources and enabled a more efficient and responsive object detection process. This brought down each pi’s cpu load to under 60% while maintaining the 20fps for object detection. This was a much more reliable solution to ensure the car would stop when a stop sign is present. By offloading the computationally intensive tasks to a dedicated device, each Raspberry Pi could focus on its specific responsibilities, improving overall system performance and responsiveness.

9.7 Conclusion

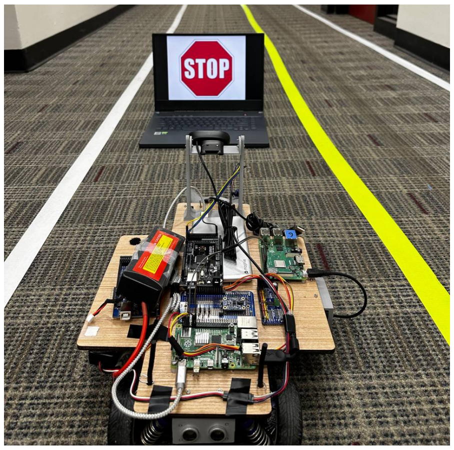


Figure 9.7.1: Donkey car stopped in front of a stop sign

The exploration and implementation of various object detection methods and optimizations have significantly improved the autonomous driving system's real-time responsiveness and efficiency performance. A robust and efficient object detection system has been developed by employing different TensorFlow models, parallel processing techniques, and multi-device processing (Figure 9.7.1).

This implementation was tested multiple times on different stop signs configuration. After the test results, the team concluded that the accuracy of this model was roughly 84%. This means the model will detect a stop sign and stop correctly 84% of the time. Videos of testing are shown in the Appendix.

This accomplishment demonstrates the importance of continuous research, experimentation, and optimization in developing advanced autonomous driving technologies. As autonomous driving advances, further improvements in object detection methods and technologies can be expected, ultimately contributing to safer and more efficient autonomous vehicles.

10. Dashboard Changes

In this section, the topic of implementing a new dashboard is introduced and discussed. The team reinvents the dashboard from the previous mPAD version and Donkey Car open source to create a united web page that allows new users a better experience with the interactive UI.

10.1 mPAD V2 Dashboard

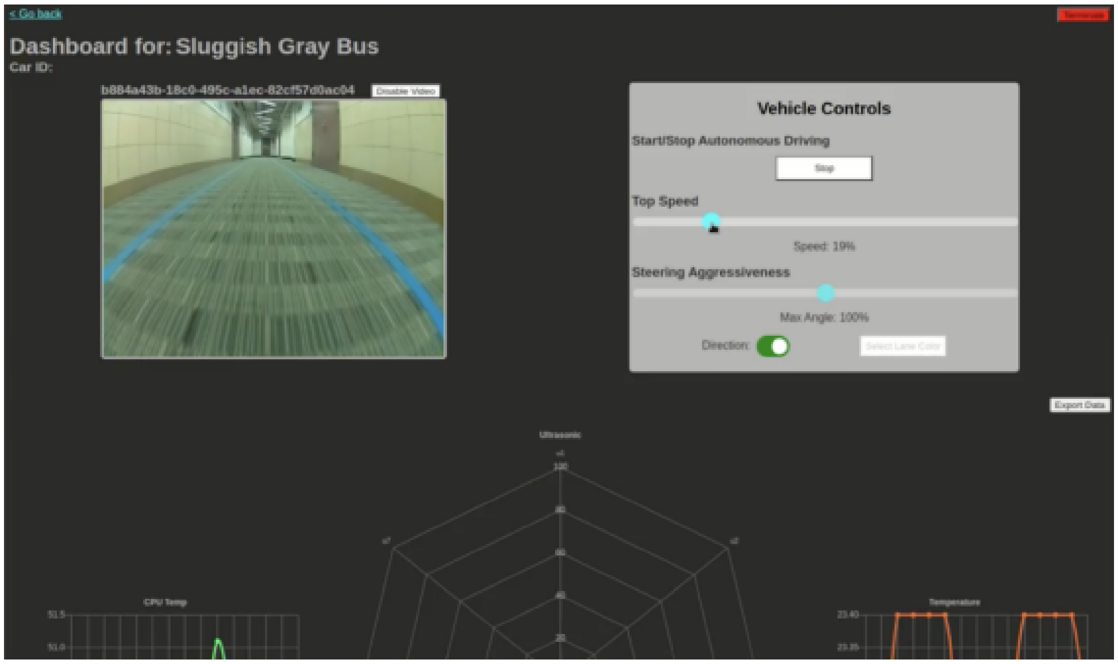


Figure 10.1.1: The mPAD V2 dashboard, reproduced as is from [17]

The initial dashboard for mPAD V2 can be seen in Figure 10.1.1. This dashboard was a custom dashboard created by last year's team. A few features they added for this dashboard include servo angle adjustment, servo/motor channel selection, a comprehensive tutorial walkthrough, manual driving, hardware component test scripts, inverse steering, throttle switches, LiPo battery level indicator, and several UI/UX updates. The dashboard displays the image from the camera on the left side, and the vehicle controls on the right. Vehicle controls included starting and stopping autonomous driving, sliders for top speed and steering aggressiveness, and selecting the lane color. The selection was used by clicking on the image in the camera display to select which color was desired for the lane. Sliders were deemed important to implement as they helped get exact values for both throttle and steering.

A key component that previous teams did was include graphs to display sensor data on their dashboards. These would take sensor data from the Arduino, send it to the Raspberry Pi, and then the Raspberry Pi would send that information to the dashboard to be displayed on the graphs. The dashboard had graphs for the data collected by Hall-effect, IMU, ultrasonics, battery percentage, humidity, and temperature sensors. The user can display or hide these graphs depending on whether or not they want to see their graphs. This is a feature that the team chose to implement in the new dashboard for mPADv3.

10.2 Donkey Car Dashboard

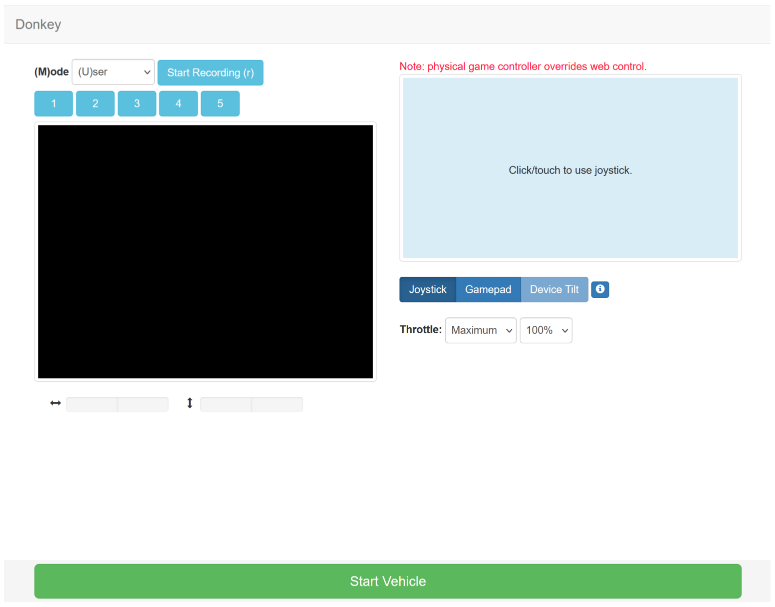


Figure 10.2.1: The original dashboard of Donkey Car, Reproduced as is from [18]

Figure 10.2.1 shows the original dashboard that the creators of Donkey Car designed. This dashboard consists of many functionalities designed for autonomous and manual driving of Donkey Car for gathering training data. The manual driving for Donkey Car is done through three different styles of controllers; Joystick, Gamepad, and Device Tilt. As seen in Figure 10.2.1, buttons for selecting a manual driving method are below the joystick container. The primary method of manual driving is through the joystick, as when the website is loaded, the joystick container is shown. When either of the other methods is selected, the joystick disappears. Gamepad is used when using a controller such as an Xbox or PlayStation controller. Device Tilt can be used with a mobile device to drive by tilting the device in any direction. The team discovered it was not correctly implemented as during testing, the team typically used a phone while manually driving, and when they tried using the device tilt, the car did not react. The small button beside the manual driving is an information button explaining the webpage's functionality. This explains how to use each button and dropdown to make the car drive as intended.

On the left side of the dashboard is a display for the video feed, which helps the user see where it is going while manually driving. Below the camera display are two bars that display what direction the car's direction while manually driving. It displays the direction of left and right and forward and backward in decimal values out of one. These were used to send values to the car to update the throttle and steering.

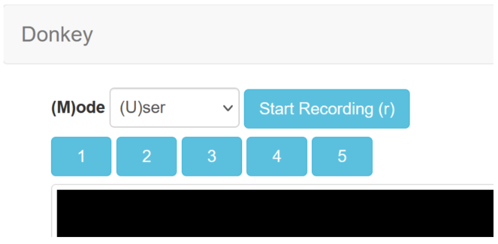


Figure 10.2.2: Drop-down for driving mode, Reproduced as is from [18]

As shown in Figure 10.2.2, there is a drop-down and six buttons above the camera display. The drop-down selects the type of driving mode the user wants: manual, autonomous, or auto steer. The manual trains the vehicle and gathers data to create a model. Autonomous is used to use the model created from the training data gathered, including the images from the camera and their associated values for the throttle and steering. Autosteer uses the model as well. However, it only uses the values for steering and allows the user to manually change the throttle in real time if they want to accelerate or decelerate. The button next to the dropdown records the images from the camera to be used when creating a training model, which automatically starts when the car is started. The five numbered buttons below allow switching between cameras if your car has multiple cameras. Finally, the button at the bottom starts the car's driving. The car will automatically start when manual driving starts but can be used when starting autonomous driving.

The team decided that the dashboard needed an update to feel more modern and make the dashboard look more eye-pleasing for the user. This was a goal of this project that was outlined earlier in the report.

10.3 mPAD V3 Dashboard

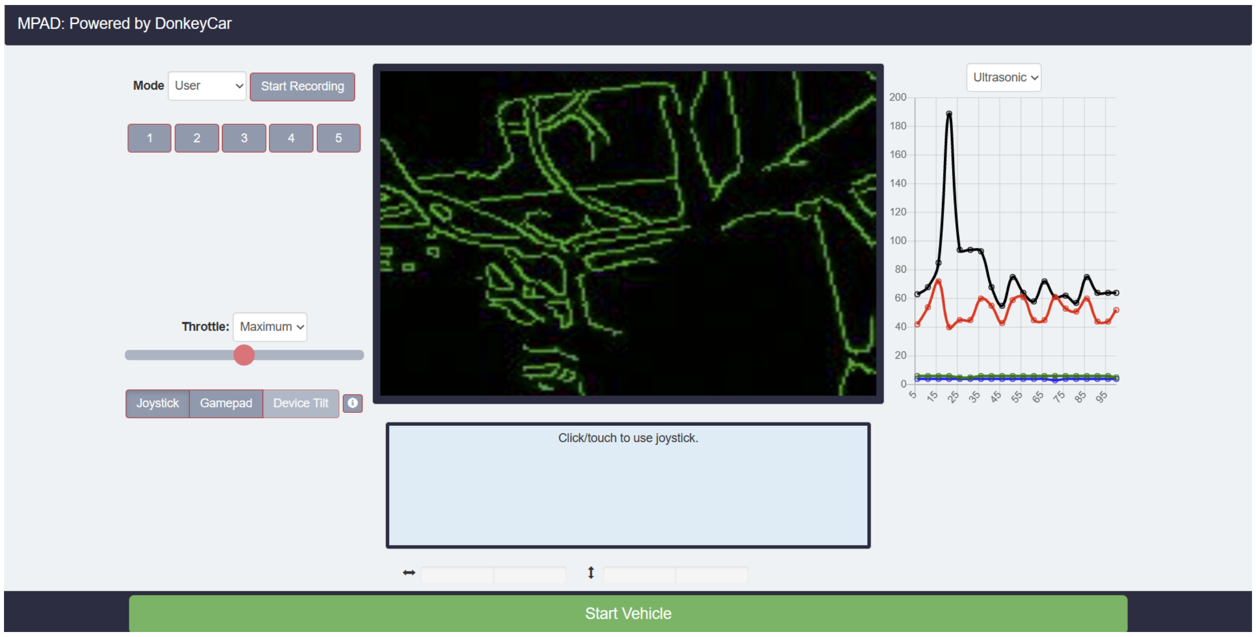


Figure 10.3.1: mPAD V3 dashboard updated

Here is the current dashboard for mPAD V3. This is a complete redesign of the original dashboard that Donkey Car used. The new dashboard still has most of the original functionality of the dashboard, as a lot of the functionality is still relevant and is used by the Donkey Car code. As seen in Figure 10.3.1, the dashboard still has the start vehicle button, camera display, joystick, directional display, selection dropdowns for the mode of driving and the throttle, and buttons for recording, camera selection, and manual driving modes. The main changes that this dashboard received with this update were a change in color palette and layout and adding a slider for the throttle and graphs to display sensor data.

Two of the major updates were updating the color palette and the layout of the dashboard. First, with the color palette, the original dashboard had a very simplistic design, with the primary color being a white background with a few blue accents, as seen on the buttons and joystick. It was decided that this needed to be updated regarding the background color as, at times, it was very hard to distinguish the differences in the UI with someone with a disability (colorblind) and overall more pleasing to the eye for a user. This palette utilizes more shades of blue with red and gray accents. Next, with the layout, the previous design seemed awkward for the user, with the juxtaposition of the camera display right next to the joystick. Therefore, the layout was updated by putting the camera display above the joystick. This will help when the user is manually driving using the joystick as they can aim the joystick towards the direction they want to go in based on the camera they see above. There is also more of an emphasis on the camera on this dashboard, as it is believed that the camera should be the main focus of the dashboard. This was done by centering the camera display and making it the largest part of the dashboard. The other changes were relocating the buttons to a column on the side of the camera. This puts all the dashboard functionality into one area, making it simple for the user to utilize.

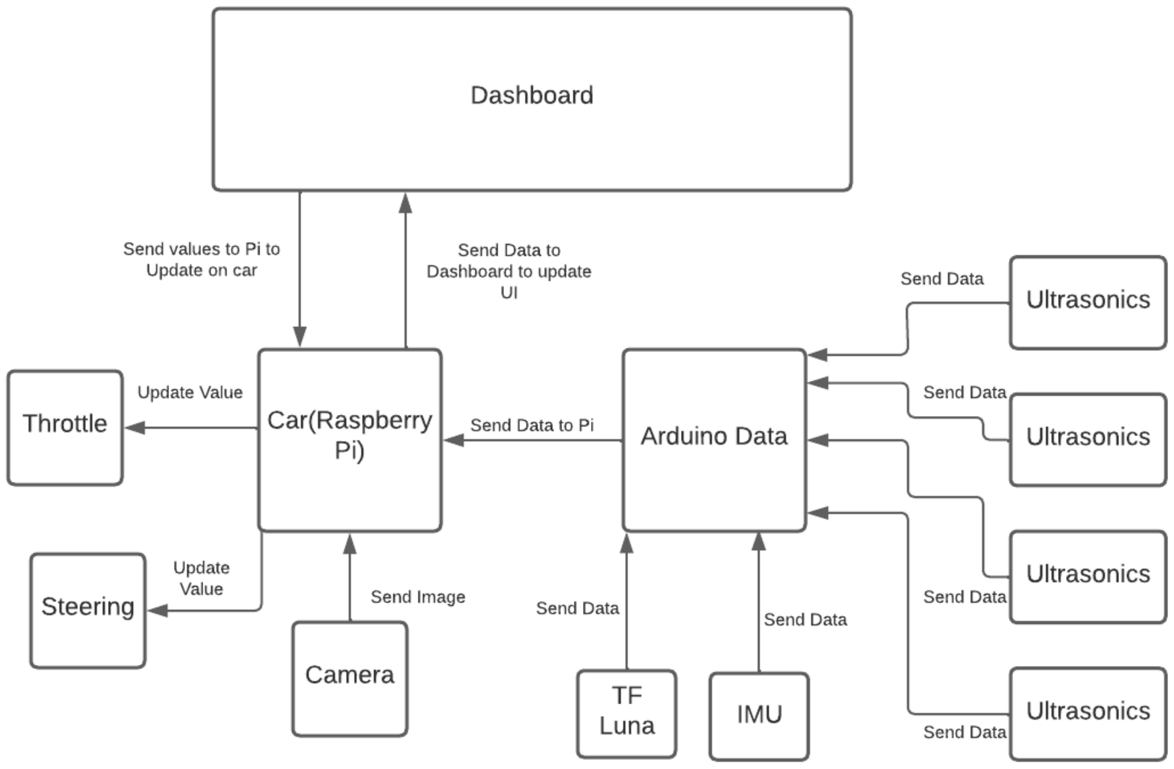


Figure 10.3.2: Flowchart on how the dashboard works behind the scenes

The addition of the graphs on the right side of the dashboard was the biggest change in comparison to the original dashboard. This was something that was introduced during the mPAD V1 project, as there were a few sensors that were added to each of the cars. These included sensors such as IMU, Ultrasonics, and TF Luna. With the data gathered from these sensors, there were graphs on the dashboard to display this information. This was something that the team wanted to bring over to the Donkey Car dashboard in order to allow the user to analyze this data. This was done by sending the sensor data to an Arduino to send that data to the Raspberry Pi (as seen in Figure 10.3.2). From there, the data that the Raspberry Pi received is displayed on the dashboard. A drop-down placed above the graphs allows the user to select which graphs the user wants to display.

The slider was also an important update for this dashboard, as the group struggled to get the right throttle percentage for each car. Some cars needed a higher throttle, and others needed a lower throttle. The difference between the previous dropdown is that the user could not get exact values for the throttle. Specifically, a 25% throttle was needed for one of the cars, but the only values the throttle dropdown had were 20% and 30%.

This dashboard was common ground between the mPADV2 and Donkey Car dashboards. This dashboard used all the functionality of the Donkey Car dashboard, as everything there was already implemented through the open-source code that the team inherited. However, this was updated with a new modern design that users will likely enjoy. The new layout and color palette help this. One thing that was taken from the mPAD V2 dashboard was the graphs for sensor data. The graphs will help the user analyze how they struggled with their models. Overall, by taking different parts from each of the dashboards, the mPAD V3 dashboard has greater functionality and design, making it easier to use.

10.4 Conclusion

The mPAD V3 dashboard presents a comprehensive overhaul of the original Donkey Car dashboard while leveraging the salient features of its predecessor, mPAD V2. Retaining the core functionality, this latest iteration integrates novel design elements and user-friendly attributes for an enhanced user experience. The revamped layout and color scheme are aesthetically pleasing, while the graphical representation of sensor data enables effortless analysis of model performance. The mPAD V3 dashboard represents a marked improvement over previous versions, and is an optimal choice for users seeking a streamlined and efficient Donkey Car dashboard.

11 Testing

The team employed a rigorous testing methodology to ensure the car's autonomous driving capability at each stage of development. The testing procedure involved generating a model using the Donkey UI and running it on the track. The primary objective was verifying the car's ability to stay within the designated lanes and complete a minimum of three laps independently. Three primary cars were used throughout the testing process, with Amazon Car 1 serving as the primary testbed for pre-processing testing. Following this, DonkeyCar was employed to test stop sign detection on two separate PIs. Finally, PreParv was utilized to test dashboard changes while updates were being made. Following the individual testing of each implementation, the team integrated all three solutions and conducted the final round of testing on Amazon Car 1 and DonkeyCar. The team also documented their testing process in detail, with videos available for review in the Appendix section.

11.1 Object Detection

Objective detection was evaluated in this study by employing the Raspberry Pi to identify stop signs of varying sizes. The stop signs were strategically positioned in different locations relative to the car. The system was rigorously tested to verify that the car would come to a complete halt as long as a stop sign was detected. Initially, stop sign detection was directly performed on the Raspberry Pi to assess the model's capability in recognizing the stop sign. Figure 11.1.1, presented below, displays the results of stop sign detection on the Raspberry Pi.

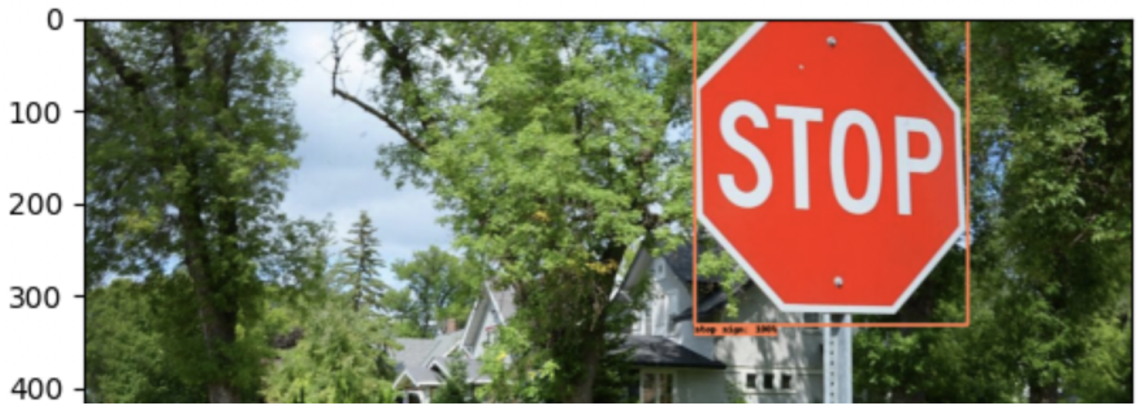


Figure 11.1.1: Stop sign detection on a raspberry pi

Subsequently, a comprehensive assessment was conducted to ensure that once a stop sign was detected, the car would indeed come to a complete stop. This evaluation involved a team member manually lifting the car while another student displayed a stop sign on their mobile device. Detailed findings from this evaluation can be found in the appendix.

Finally, the testing phase concluded with a verification process to ensure the car would stop at the appropriate moment when a stop sign was on the track. Additional information regarding this assessment can be found in the appendix.

11.2 User Interface

The testing phase for the updated dashboard primarily focused on ensuring the smooth and accurate integration of sensor data into the dashboard. The process involved rigorous testing of various methodologies to determine the most efficient and effective approach. The testing team successfully identified the ideal solution to create a text file for each sensor reading, which was then processed using an XMLHttpReceiver to transfer the values to the relevant graphs on the dashboard.

Additionally, testing was conducted on various aspects of the dashboard's front end, including HTML, CSS, and JavaScript files. The team tested the slider for the throttle and verified that the updated value was accurately reflected on the dashboard and within the Javascript file where the throttle is updated. Similarly, the drop-down for graph selection was thoroughly tested to ensure that the correct graph containers were displayed while hiding the others. Throughout the testing phase, the team maintained high professionalism, adhering to industry best practices and standards. The team diligently addressed all issues and worked tirelessly to ensure the updated dashboard was thoroughly tested and fully functional. Overall, the team's efforts have resulted in a robust and reliable dashboard that meets the highest standards of quality and performance.

11.3 Mechanical Repairs

As each car was repaired and refurbished, it would be subject to manual testing. This testing would involve connecting the cars' ESC to an Ikonnik ET3 3-Channel receiver and accompanying transmitter, ensuring that the car could properly receive steering and throttle signals. After the cars were determined to be able to run for at least a few minutes, they would be deemed suitable to hand off to the rest of the team.

While testing the autonomous driving software on the cars, some of the cars would run into mechanical issues and be brought back in for repairs. Sometimes, the mounts holding the ultrasonic sensors would unscrew themselves through repeated jostling, whether through driving or manual transportation. A simple reattachment would rectify this issue.

There were instances where the ESCs would stop connecting to either the car or would no longer accept inputs from a controller. This would result in a complete replacement of the ESC, which would resolve the issue.

The camera mounts were light and thin, designed to keep the cameras at the appropriate height but not shift the car's center of gravity. Unfortunately, due to the design and material, the mounts would occasionally snap open, preventing angle change and, at times, falling off. Due to printing times, a full reprinting and replacing would be conducted, normally the next day.

As the software developed, it was noticed that the more laps the car did, the more likely it was to drift off the track. Occasionally, the car would stray too far and collide with the wall or a nearby object. Due to this, along with the small, short screws used to attach components, items would fall off the car and onto the car during driving, jostled by movement or impacts. Reattachment would usually fix the issue, with other solutions such as velcro dots also being implemented.

11.4 Donkey Car

During the testing phase of the Donkey Car project, the team implemented a rigorous approach to assessing the performance of the AI model. Following the generation of the model via Donkey UI, the team would conduct autonomous driving tests on the field. This method involved placing the RC car on the field and observing its behavior while driving autonomously. The primary objectives of this test were to ensure that the car could move forward and drive on straightaways while staying within the lane lines. The latter was particularly crucial, as the car's ability to navigate turns while staying within the lane lines was essential for successful autonomous driving. The team emphasized testing the car's ability to navigate turns. This was an area of concern, as the car's ability to take turns within the lane lines would determine its ability to navigate the track effectively. When the car took too wide or too sharp a turn, it would go off the track, highlighting the importance of this aspect of testing.

Lighting was also a significant consideration in autonomous driving tests. The camera needed to identify the lanes to enable the car to navigate the track successfully. However, implementing the new pre-processing feature mitigated the impact of lighting conditions on the car's performance. The environment was considered suitable for autonomous driving as long as the camera could detect the two lanes. The team considered a model successful if the car could complete three or more laps autonomously without any issues or disturbances. This rigorous testing ensured that the AI model generated through Donkey UI was high quality and capable of navigating the track effectively. It should be noted that visual aids in the form of videos are available for reference in the Appendix [15.3], depicting the rigorous testing process undertaken to evaluate the performance of the Donkey Car AI model. These videos provide a comprehensive overview of the testing process, enabling interested parties to gain further insights into the approach taken by the team to ensure the effectiveness of the model.

12. Discussion

The development of the third version of the mPAD aims to achieve various technical design objectives, including improving the reliability of self-driving, integrating object detection, enhancing the user-friendly dashboard, and improving modularity and scalability. This selection highlights the process of achieving these objectives and the broader impact of our project.

12.1 Improving Self-Driving Reliability

The decision to abandon the previous year's progress and start anew was primarily motivated by the unreliable nature of the autonomous driving algorithm. Rather than attempting to debug the problematic code, the team concluded that a fresh start would be more advantageous. An open-source platform with multiple developers was chosen to ensure the vehicle's reliability to facilitate prompt bug fixes. Compared to the previous year's team, our team produced a significant amount of documentation, guides, and codebase to facilitate a smooth transition to the following year's team.

The team established specific requirements that had to be met, the first of which was the car's ability to remain within its designated lane, interpret its environment, and identify the lane's position relative to the car. This requirement was successfully achieved using the Donkey Car platform, as evidenced by the data collected by the functional autonomous driving algorithm. The second requirement was that the program performs effectively in varying lighting conditions. Implementing a pre-processing pipeline that eliminated external factors, such as lighting conditions, met this condition. The final requirement was that the car must stay within its designated lane to avoid accidents. By training the model through a driver, we ensured that the car always remained in its lane. This requirement is validated by observing the demo video in the appendix. All requirements have been met, and the team has successfully achieved the objectives outlined in this paper.

12.2 Integrating Object Detection

The integration of object detection was considered an important goal of this year's MQP team. Incorporating this technology would have accomplished something all three previous years' MQP teams wanted to accomplish for the future. To ensure we met this objective, a few requirements were set beforehand.

The integration of object detection was a significant objective of this year's MQP team, which aimed to accomplish a goal desired by the last three MQP teams. To achieve this goal, the team established several requirements, including the need for the car to interpret data from sensors accurately and identify the stop sign's location. The program must also make reliable decisions to ensure the car stops safely at the sign. The demo video in the appendix demonstrates that all requirements were met, with the car detecting a stop sign, identifying its location, and remaining stationary until the sign is no longer visible. The team has thus achieved the objectives outlined in this appendix.

12.3 Enhancing the Dashboard

The dashboard plays a crucial role in the overall package of this project as it serves as the interface between the user and the RC car.

Therefore, the team established specific requirements to enhance the dashboard's user experience. The first requirement was redesigning the dashboard's UI to create a visually appealing and intuitive user experience. The second requirement was to improve the UX by optimizing interactions and ensuring compatibility with the Donkey Car Open Source program.

The third requirement was to incorporate sensor data into the dashboard in an easy-to-understand graph format, which can be downloaded upon request. Lastly, the team adhered to established design principles to ensure the dashboard was functional and aesthetically pleasing.

Although user studies could not be conducted due to time constraints, the team believes these requirements were met based on the demonstration of the dashboard in the appendix.

12.4 Incorporating Modularity and Scalability

The goal of modularity and scalability in this MQP project is crucial for its success. It enables the platform to apply to various automobile types and can be scaled up for more powerful vehicles.

A set of requirements was established to ensure that this objective is achieved. The first requirement is to develop a modular package adaptable to different car sizes and configurations, which was demonstrated in the appendix using various-sized RC cars. The second requirement is to redefine the software program to allow for customization based on vehicle specifications, which unfortunately could not be accomplished due to time constraints. The third requirement is investigating a modular box with sensors for easy deployment across multiple vehicles. While a modular box was not designed, the required sensors could be integrated into multiple vehicles. The fourth requirement is creating a comprehensive user guide for easy installation and system maintenance, as demonstrated in the appendix. Lastly, documentation of design, development, testing results, and user guide updates will be included for future maintenance and upgrades, available on the team's GitHub repo and the donkeycar wiki.

While not all requirements were met, the team believes the objective has been mostly fulfilled.

12.5 Conclusion

The team is pleased to report that the objectives for mPAD V3 have succeeded. Through a range of changes and enhancements, including the implementation of a new camera for object detection, improvements to the self-driving code to compute pre-processing, expanded sensor support to include ultrasonics, and more, testing on vehicles of varying sizes, the addition of dashboard quality-of-life features, and a comprehensive setup guide, the system has been made more robust, scalable, and intuitive.

The team acknowledges that further efforts are required to enhance the system's overall performance, incorporate collision avoidance capabilities, and detect other signs apart from stop signs into the self-driving logic, which are desirable features yet to be implemented. Furthermore, the team understands that the modular and scalability objective may not be completely met as the team did not have time to design a physical modular package to store all components. In addition, in the software created, there is currently no way a user can enter their vehicle constraints, as mentioned in the requirements. While the team claims the dashboard requirements are met, it was subjective depending on the audience. The team did not have time to study how well users received the dashboard. Nevertheless, in the team's eyes, this was an overall successful approach to the dashboard due to how efficient the dashboard was during the testing process of this project.

Despite these limitations, the team acknowledges the substantial improvements made in mPAD V3 compared to its predecessor and confidently asserts that the project goals have largely been achieved.

13. Conclusion

Overall, the project can be deemed a success as the team successfully delivered a functioning product amidst numerous changes over the given timeline. At the outset, the team faced the vehicles' mechanical faults and the software program's inability to launch correctly, let alone connect with the car. However, the team adeptly disassembled the given code base and restructured it using the open-source platform Donkey Car. This allowed for better organization of the program and enhanced its operational efficiency.

Furthermore, the team proficiently implemented a pre-processing algorithm, the Hough line transformation, which proved highly effective in removing noise from images and improving the linear regression model's performance. This resulted in the model focusing solely on the primary objective of driving between the lanes without being distracted by extraneous factors such as lightning. In addition, the team developed a stop sign detection program that enabled the car to halt upon detecting a stop sign. This marked a significant milestone in object detection and the model's interaction with the Donkey Car model.

In pursuit of a better user experience, the team successfully designed and implemented a new UI that visually presented sensor data using graphs and charts. Additionally, the team created a new color palette to enhance the website's aesthetics and appeal to a broader range of users. These enhancements ultimately contributed to the project's success, resulting in a fully functioning product that provides a step-by-step guide for users to create their AI model to drive autonomously on a customized track, including the ability to stop at stop signs. In conclusion, this codebase's modular structure allows for incorporating additional implementations in the future, thus paving the way for new and exciting possibilities.

13.1 Future work

As the mPAD project continues to evolve and advance, there are several focus areas for future work. These efforts aim to improve the system's performance, scalability, and flexibility and ultimately bring us closer to achieving the goal of a fully autonomous driving package.

13.1.1 Upgrading Computing Power

As the mPAD project moves forward, one of the main areas of focus for future work is to upgrade the computing power used in the system. Currently, the system uses Raspberry Pis for both driving and object detection. While this works, it is limited by the computing power of the Raspberry Pi, which can become a bottleneck as the system becomes more complex. Therefore, the team plans to upgrade to a more powerful, Linux-based microcomputer and connect it to Arduino and cameras. This will provide more processing power and improve the system's overall performance.

13.1.2 Custom Boards and Casings

Another area of focus for future work is creating a custom board to connect sensors and servos. This board will be designed to meet the specific needs of the project and will allow for better integration of the different components. Additionally, future teams can plan to create a custom casing to house all components and make it modular with plugins for sensors and servos. This will allow for easy customization and upgradeability of the system.

13.1.3 Working with Larger Vehicles

One of the objectives of mPAD was to create a package that unified all components needed to make an RC car run autonomously. The platform is built around a core set of components, including a Raspberry Pi computer, a motor controller, a servo shield, and a camera, that can be easily customized and extended using a range of plug-and-play modules.

One of the main advantages of the modular design of the mPAD (powered by Donkey Car) platform is that it allows users to quickly and easily add new features and functionality to their vehicles without redesigning the entire system. For example, if users want to add sensors for obstacle avoidance, they could simply add a new module that integrates with the existing hardware and software stack. Thus implementing object detection using ultrasonics or TF Luna would be very straightforward.

Another benefit of modular design is that it allows for easy experimentation and prototyping. Users can try out different components and configurations to see what works best for their use case without committing to a specific design upfront. This allows easy access to add to previous work and keep updating the project as time passes.

Figure 13.1.3.1: Children's electric car along with a golf cart reproduced as is from [1,3]

Donkey Car’s platform's modularity enables easy customization and scalability. Appropriate modifications allow the platform's modular architecture to work with various vehicles, from kids' electric cars to golf carts (Figure 13.1.3.1). This will require significant modifications to the current system and involve working with larger and more complex systems. However, this would continue pushing the limits of what is possible with autonomous driving technology, especially the power of mPAD.

Overall, the modularity of this platform makes it a powerful tool for robotics enthusiasts and developers who want to build and experiment with autonomous vehicles. By providing a flexible and extensible framework, the platform enables users to focus on building the features and functionality they need without getting bogged down in the details of hardware and software integration.

13.1.4 Integrating Sensors Logic to Driving Logic

In addition to upgrading the computing power and creating custom boards and casings, future mPAD teams should also focus on adding sensors to enhance the capabilities of the autonomous driving system. Specifically, to correctly utilize ultrasonic sensors, Time-of-Flight (ToF) sensors like the TF Luna, and Inertial Measurement Units (IMUs). These sensors will enable the system to detect and avoid obstacles and adjust steering and throttle controls to accommodate uneven terrain and inclines. For example, ultrasonic sensors can detect objects in the vehicle's path and trigger evasive actions. In contrast, ToF sensors can provide more accurate distance measurements and help avoid collisions. IMUs can provide precise orientation and motion tracking data to help the vehicle adjust its trajectory on ramps and other uneven surfaces.

By incorporating these sensors into the system, the mPAD project will achieve greater safety, stability, and precision in autonomous driving. Furthermore, the team will be able to leverage the data collected by these sensors to train machine-learning models that can improve the system's decision-making capabilities in various scenarios. Adding these sensors represents an important step forward in developing the mPAD project and will help bring us closer to achieving fully autonomous driving.

13.1.5 Summary

In summary, future work for the mPAD project will involve upgrading the computing power, creating custom boards and casings, and working with larger vehicles. These efforts will improve the performance and scalability of the system and ultimately bring us closer to achieving the goal of a fully autonomous driving package. While these goals are ambitious, the team is dedicated to pushing the boundaries of autonomous driving technology and creating a safer, more efficient mode of transportation for everyone.

13.2 Reflections

During the mPAD V3 development, the team members gained valuable lessons individually and collectively. The team possessed diverse personal knowledge, which they could leverage to enhance their overall performance. The following sections provide an account of the team members' backgrounds before working together, the skills they developed during the project, and their reflections on the project experience.

13.2.1 Rohan Anand

As a student majoring in Computer Science and Robotics Engineering, the mPAD project provided an excellent opportunity to showcase the knowledge I acquired at WPI. With a solid background in Python programming and proficiency in electrical engineering(thanks to RBE1001, RBE2001, and RBE2002), specifically in electrical wiring, I was well-equipped to handle the hardware aspects of the project. However, the project presented a significant challenge as it required me to develop an autonomous program from scratch and append it to an existing codebase - the Donkey Car open-source platform, which had undergone extensive development by numerous contributors. I have dealt with autonomous projects before, thanks to RBE3002 which had me use ROS to autonomously navigate through a maze autonomously, thus giving me an edge on what to expect when going into this project.

During the project, I focused on pre-processing as a significant objective, which involved learning how to translate existing images into ones that highlight the key features a linear regression model should focus on. I had some experience in pre-processing thanks to RBE3001, which required me to do pre-processing on colored balls to make them stand out a lot more. The Donkey Car platform primarily relies on machine learning models, such as linear regression and convolutional neural networks (CNNs). As a result, I had to conduct extensive research to understand the main pipeline of Donkey Car and how any modifications would affect the autonomous features of the program.

The project provided me with valuable insights into debugging a large codebase and integrating custom code into the pipeline while ensuring efficiency. Overall, the mPAD project was an excellent opportunity to apply my knowledge and gain valuable experience in robotics engineering.

13.2.2 Martin Bleakley

As a Robotics Engineering major, the Modular Package for Autonomous Driving project was a great opportunity for me to gain experience working on a multidisciplinary project. Throughout the project, I gained valuable experience integrating and modifying existing software onto an existing platform. I also gained the experience of helping to make this decision which significantly impacted more than ¾ of my time working on the project. Working collaboratively with my teammates, I learned how to effectively communicate and collaborate with individuals from different backgrounds and skill sets. Additionally, the project allowed me to put into practice many of the mechanical and electrical skills I had gained throughout my coursework(such as RBE3002), providing me with a deeper understanding of the practical applications of robotics engineering. Overall, the project was a valuable learning experience that has helped me to develop important skills that will be useful in my future endeavors.

13.2.3 Allison Colón-Heyliger

As a Mechanical Engineering major, my most remarkable experiences at WPI were the classes focused on hands-on experiences and learning the core concepts of design work. mPAD required me to use everything I had learned in those classes and more, making it a rewarding and worthwhile experience. I was the sole mechanical engineer, assisting the others through repairing the run-down fleet of cars left behind by other years. It was a task no less important than the software the cars would run and would bring many challenges and learning opportunities as time progressed.

While I had worked on many group projects before this, it was the first time I would work in parallel with the rest of the team. It was an interesting learning experience, meeting with other subgroups and giving updates on my work. It reflected what I would expect in the workforce, and I left the experience feeling better equipped to work in similar dynamics. The mPAD project also introduced me to the necessity of proper documentation and organization. It taught me how to navigate old documentation folders and documents to find the information needed to progress on the cars. Not only did I learn how to deal with the ghosts of previous years, but I also learned how to communicate with the others on the team. Working with scheduling to work on key tasks and communicating clearly when a car needs repairs, I am sure I will be taking these skills into the field in the future. As for the past, classes such as Intro to

CAD (ES 1310) and Modeling And Analysis Of Mechatronic Systems (ME 4322) taught me the necessary skills to bring out the cars' full potential. Intro to CAD, for example, taught me how to utilize SolidWorks to design and export models of 3D printed pieces, as well as presenting the opportunity to become SolidWorks certified.

Nothing was more satisfying than watching the cars drive autonomously along the track, knowing where they started. The mPAD project allowed me to apply what I learned into a practical application, becoming a worthwhile experience. It prepared me to work with others, strengthened my understanding of mechanical and design concepts, and helped me gain insight into the other aspects of engineering often paired alongside mechanical. It was an experience I would not soon forget.

13.2.4 Ian Khung

As a Computer Science major and a Data Science Minor, working on a physical object such as a robot was a first for me. Towards the beginning of the year, I had a lot to catch up on due to a lack of knowledge in hardware and how to code hardware. As the year progressed, I felt like I gained a vast knowledge of how to work with robots and the different vocabulary used with robots.

I was much more comfortable working on this project because the open-source code we worked on for Donkey Car was written in Python, Javascript, HTML, and CSS. Python and Javascript are two languages I have had experience with in previous classes I took at WPI. On the other hand, I was very excited to work with both HTML and CSS as these are two languages that I wanted more experience with. While working in the Software Engineering Class (CS3733), my role on the team was the UI/UX Designer, so during my time on this project, working with the dashboard was a perfect place for me to focus my time. In addition, I worked with Javascript during my Computer Graphics (CS4731) class, which gave me a good background when working with the dashboard and its interactive features. I felt that I got great experience on this project which will help me at the beginning of my career.

13.2.5 Ishan Rathi

During my time at WPI, I undertook an intensive four-year program in Robotics and Computer Science, which required a great deal of dedication and hard work. Working on this project gave me a unique opportunity to apply my computer science and robotics skills to a practical problem. I demonstrated my ability to turn theoretical concepts into real-world solutions.

I gained invaluable experience working in a core research position by taking on this project. I had the opportunity to engage in a rigorous research process, which included collecting and analyzing data, identifying patterns and trends, and developing machine-learning models to address specific problems. This experience was crucial as it helped me understand what it takes to be a successful researcher and prepared me for future academic endeavors.

Moreover, the project enhanced my problem-solving skills and taught me to approach complex problems methodically. The experience taught me to break down challenging tasks into smaller, more manageable pieces, enabling me to tackle them systematically. This approach proved to be invaluable as it not only helped me complete the project successfully but it is also a skill that I continue to apply in my academic and professional endeavors. The classes that helped me learn the necessary skills were Machine Learning (CS4342), Robotics 3002 (RBE3002), and Mobile Computing (CS4518). These classes helped facilitate the necessary skills to implement object detection and help with pre-processing an image.

In conclusion, my work on this project was a significant milestone in my academic journey. It allowed me to demonstrate my computer science and robotics skills, provided valuable research experience, and helped me grow as a student. I am grateful for this opportunity and believe it has prepared me well for the challenges ahead.

14. References

[1] Aliexpress, "6v Children Remote Control RC Ride On Electric Car Four Wheel Double Drive Toy Car Rechargeable Baby Can Sit On|Ride On Cars|," n.d. [Online]. Available: https://www.aliexpress.com/item/2251832658231944.html?src=ibdm_d03p0558e02r02&sk=&aff_platform=&aff_trace_key=&af=&cv=&cn=&dp=. [Accessed April 22, 2023].

[2] A. Gunnarsson, "Real time object detection on a Raspberry Pi," Dissertation, Linnaeus University, Växjö, Sweden, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-89573.

[3] Prime Motorcycles, "Hisun Pulse Golf Cart [Photograph]," 2020. [Online]. Available: https://www.primemotorcycles.com/Golf-Carts-Hisun-Pulse-Golf-Cart-2020-Sanford-FL-f0db44c3-44c0-45b4-94a4-aaf600b2b533.

[4] Z. Jiang, L. Zhao, S. Li, and Y. Jia, "Real-time object detection method based on improved YOLOv4-tiny," 2020.

[5] T. Q. Khoi, N. A. Quang, and N. K. Hieu, "Object detection for drones on Raspberry Pi potentials and challenges," in IOP Conference Series: Materials Science and Engineering, vol. 1109, no. 1, 2021, Art. no. 012033. doi: 10.1088/1757-899x/1109/1/012033.

[6] "TensorFlow," Wikipedia, April 20, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=TensorFlow&oldid=1150793006.

[7] "Tesla Roadster: What We Know So Far," Car and Driver, May 20, 2021. [Online]. Available: https://www.caranddriver.com/tesla/roadster.

[8] GitHub, "Build software better, together," n.d. [Online]. Available: https://github.com/logos. [Accessed April 24, 2023].

[9] DFRobot Wiki, "Gravity: Huskylens AI machine vision sensor," n.d. [Online]. Available: https://wiki.dfrobot.com/HUSKYLENS_V1.0_SKU_SEN0305_SEN0336. [Accessed April 24, 2023].

[10] TensorFlow, "Object detection: TensorFlow Hub," n.d. [Online]. Available: https://www.tensorflow.org/hub/tutorials/object_detection. [Accessed April 24, 2023].

[11] J. Cusack, “How driverless cars will change our world,” BBC Future, 24-Feb-2022. [Online]. Available: https://www.bbc.com/future/article/20211126-how-driverless-cars-will-change-our-world . [Accessed: 26-Apr-2023].

[12] Bill. VISNIC, “SAE revises levels of Driving Automation,” SAE International, 24-Jun-2021. [Online]. Available: https://www.sae.org/news/2021/06/sae-revises-levels-of-driving-automation. [Accessed: 26-Apr-2023].

[13] “Autopilot,” Tesla. [Online]. Available: https://www.tesla.com/autopilot. [Accessed: 26-Apr-2023].

[14]“Make driving chill,” comma.ai - make driving chill. [Online]. Available: https://comma.ai/. [Accessed: 26-Apr-2023].

[15]Kim, T., Marge, A., Rondon, J., Wood, K., (2020). Modular Self-Driving and Sensor Packages for R/C Cars. Unpublished Major Qualifying Project. Worcester Polytechnic Institute.

[16] Azevedo, E. G., Jeanlys, A., Mercer, C., Mugabo, J., Reardon, Eric., Sel. T., (2021). Modular Package for Autonomous Driving (MPAD) (Undergraduate Major Qualifying

Project). Retrieved from Worcester Polytechnic Institute Electronic Projects Collection: https://digital.wpi.edu/concern/student_works/3b591c485?locale=en

[17] Lima. B., LoPreti. A., Riviere. T., Simpson. L., Yang. W., (2022). Modular Package for Autonomous Driving V3(Undergraduate Major Qualifying Project). Retrieved from Worcester Polytechnic Institute Electronic Projects Collection: https://digital.wpi.edu/concern/student_works/4j03d296z?locale=en

[18] "Donkey UI," UI - Donkey Car, [Online]. Available: https://docs.donkeycar.com/utility/ui/. [Accessed: April 26, 2023].

[19] P. Marian, "Arduino Mega 2560 pinout," ElectroSchematics.com, 21-Apr-2014. [Online]. Available: https://www.electroschematics.com/arduino-mega-2560-pinout/. [Accessed: 26-Apr-2023].

[20] Anon, Abba, Dejan, Cristina, Rohitash, D. Nedelkovski, H. Olin, Naja, Matt, Chris, Carmine, B. Burroughs, Gustavo, Brahim, Harshit, A. salim, M. Crawford, N. Puff, Malik, Fatso, Anant, Yoangel, S. Seth, Asda, William, Rakan, Pramod, Ton, Anamul, Vignesh, G. J. Alande, J. Kathryn, Sougata, T. Schulz, Aiden, Sougata, Harvey, Thor, Haravey, Extremeus, Amirul, Ian, T. W. Wize, Tino, Rhydo, Mujadidd, S. shetty, S. Looney, Pradeep, and Pictorobo, "Ultrasonic sensor HC-SR04 and Arduino - Complete Guide," How To Mechatronics, 18-Feb-2022. [Online]. Available: https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/. [Accessed: 26-Apr-2023].

[21] "Lidar TF luna- laser distance sensor - 8M -," BOTLAND, 17-May-2022. [Online]. Available:

https://botland.store/time-of-flight-sensor/16638-lidar-tf-luna-laser-distance-sensor-8m-uarti2c-5903351249041.html. [Accessed: 26-Apr-2023].

[22] “Amazon.com: 10 DOF IMU Sensor (C) inertial measurement unit motion ...” [Online]. Available: https://www.amazon.com/Inertial-Measurement-Position-Temperature-Consumption/dp/B00VUHIDLA. [Accessed: 26-Apr-2023].

[23] “AWS deepracer - the fastest way to get rolling with machine learning.” [Online]. Available: https://aws.amazon.com/deepracer/. [Accessed: 26-Apr-2023].

[24] freeCodeCamp.org, “An introduction to Q-learning: Reinforcement learning,” freeCodeCamp.org, 09-Aug-2018. [Online]. Available: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/. [Accessed: 26-Apr-2023].

[25] Gallardo. C., (2022). Refurbishment of Cars and Installation of Sensors for Testing Autonomous Driving.(Undergraduate Major Qualifying Project). Retrieved from Worcester Polytechnic Institute Electronic Projects Collection: https://digital.wpi.edu/pdfviewer/6969z401m

[26] “Raspberry PI,” *Google search*. [Online]. Available: https://www.google.com/search?client=safari&rls=en&q=rasberry%2Bpi&ie=UTF-8&oe=UTF-8. [Accessed: 26-Apr-2023].

[27] “Huskylens - an easy-to-use AI camera: Vision Sensor,” *DFRobot*. [Online]. Available: https://www.dfrobot.com/product-1922.html. [Accessed: 26-Apr-2023].

[28] “Robot operating system,” ROS. [Online]. Available: https://www.ros.org/. [Accessed: 26-Apr-2023].

[29] E. Curiosities, “Controlling two ultrasonic sensor with Arduino Uno with code: HC-SR04 module control using Arduino with code,” Controlling two Ultrasonic Sensor with Arduino Uno with code | HC-SR04 module control using Arduino with code, 07-Apr-2023. [Online]. Available: https://www.electronicscuriosities.com/2023/03/controlling-two-ultrasonic-sensor-with.html. [Accessed: 28-Apr-2023].

[30] Vikas, “MPU9250 9-DOF Gyro Accelerometer magnetometer module with Arduino,” Microcontrollers Lab, 04-Jul-2022. [Online]. Available: https://microcontrollerslab.com/mpu9250-9-dof-module-pinout-interfacing-with-arduino-features-specifications/. [Accessed: 28-Apr-2023].

[31] J. Hrisko, “Distance detection with the TF-luna lidar and Raspberry Pi,” Maker Portal, 16-Nov-2022. [Online]. Available: https://makersportal.com/blog/distance-detection-with-the-tf-luna-lidar-and-raspberry-pi. [Accessed: 28-Apr-2023].

15. Appendix

15.1 Hardware Cost and Specifications

Component	Cost	Specifications
Raspberry Pi 3	$105	Raspberry Pi 3
Raspberry Pi 4	$160	Raspberry Pi 4
NETGEARRouter	$120	Netgear Router
MEGA R3 Board	$22.99	MEGA Board
Ultrasonic Sensor	$3.50	HC-SR04 Ultrasonic
IMU	$1.40	IMU
TF Luna	$28	TF Luna
HuskyLens Camera	$54.90	HuskyLens Camera
Logitech Webcam	$23.64	Logitech Webcam
ESC	27.63	ESC
Servo Shield	$9.19	Servo Shield

15.1.1 Hardware Component List

This excel sheet shows the types and number of components already installed on the cars when first received.

mPAD Car Component Spreadsheet

15.2 mPAD V3 Setup

15.2.1 Donkey Car Setup

The following guide provides step-by-step instructions for installing and running the mPAD system on a Raspberry Pi connected to a servo shield, with throttle and steering sensors and a webcam. The guide assumes basic familiarity with programming, Raspberry Pi, and Donkey Car, and it provides links to relevant resources for beginners. The process involves downloading the code from Github, installing the required dependencies, creating a virtual environment, and installing custom Donkey Car Python code. The guide also includes optional steps for installing OpenCV and upgrading to a micro-computer for better performance. After the installation, the user must calibrate the steering and throttle values, edit the configuration file, and launch Donkey Car.

1. Set up your car by connecting a Raspberry Pi to a servo shield.

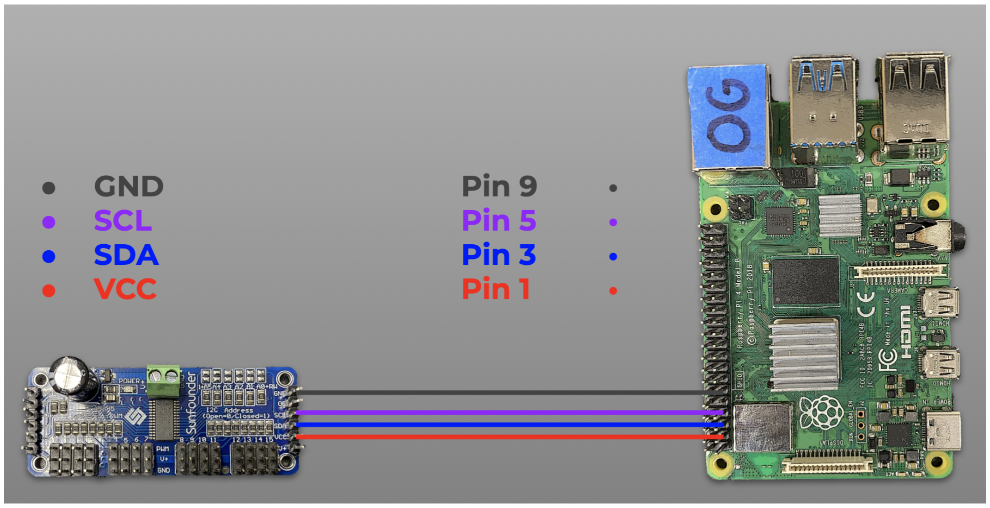


Figure 15.2.1.1: Connection of servo shield to raspberry pi, reproduced as is from [17]

2. Connect the servo shield to the servo controller.

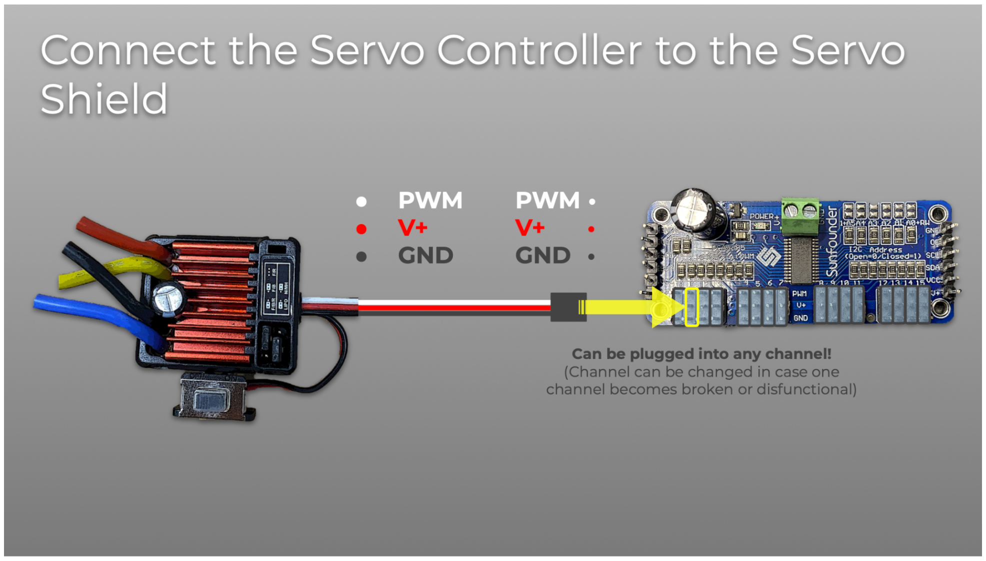

Connect the Servo Controller to the Servo Shield
● PWM
● V+
● GND
PWM •
V+ •
GND •
Can be plugged into any channel!
(Channel can be changed in case one channel becomes broken or disfunctional)

Figure 15.2.1.2: Connection of servo shield to servo controller, reproduced as is from [17]

3. Connect a webcam to the Raspberry Pi.

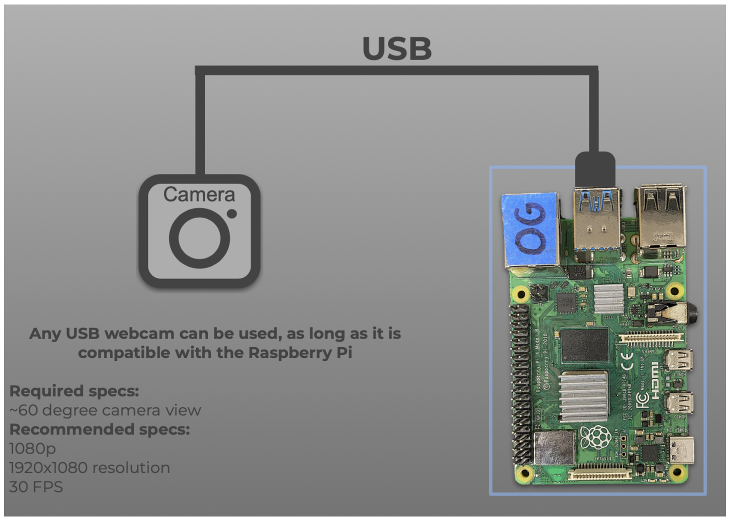


Figure 15.2.1.3: Connection of USB camera to raspberry pi, reproduced as is from [17]

4. Open a terminal and enter the following command

```
wget
https://raw.githubusercontent.com/ishan190425/DonkeyCustom/main/mpad.sh
&& chmod +x mpad.sh && ./mpad.sh
```

This will download the custom donkeycar program, and install it and create a car located in the ~/mycar folder.

5. Calibrate steering, by entering the following command. Find values between 200-1500 for forward values, reverse values and stopped values. For example, on a car 450 would be forward

```
donkey calibrate --pwm-pin=PCA9685.1:40.1
```

6. Calibrate the throttle, by entering the following command. Find values between 200-1500 for forward values, reverse values, and stopped values. For example, on a car 450 would be forward

```
donkey calibrate --pwm-pin=PCA9685.1:40.0
```

7. Start driving by entering the following command.

```
python3 manage.py drive
```

8. Continue to follow the wiki for the remaining steps on how to train using this platform along with the creation of a model: Donkey Car Wiki .

15.2.2 Arduino Sensor Connection

This is the following circuit for plugging in all sensors to the Arduino(see Figure 15.2.2.1). Once all these sensors are plugged in, connect the Arduino Mega and the PI together using a USB 2.0 Cable Type A/B.

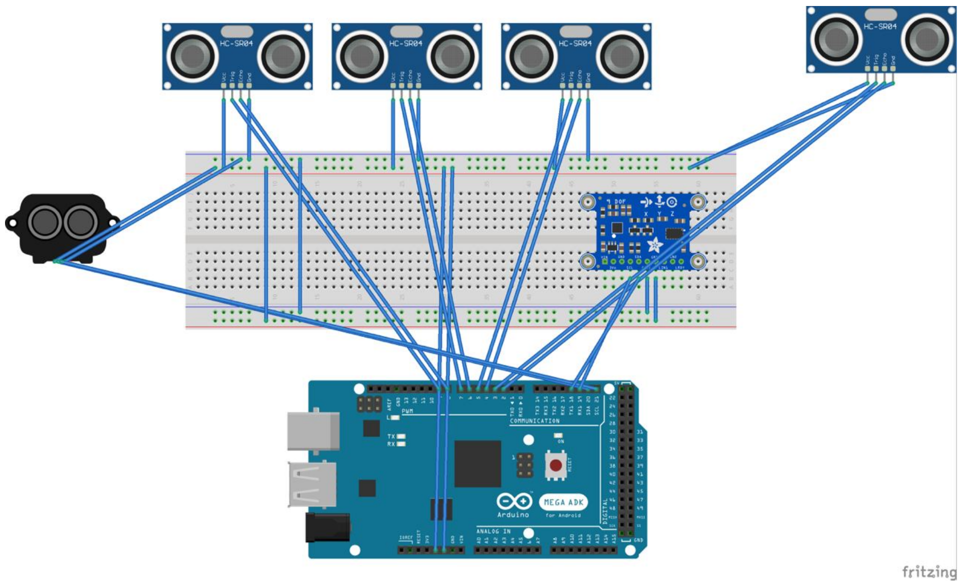


Figure 15.2.2.1: Circuit configuration for Arduino

15.2.2.1 Ultrasonics wiring

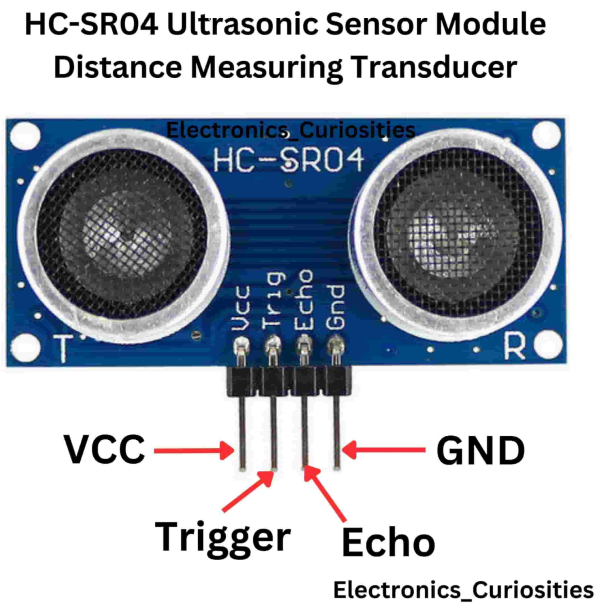


Figure 15.2.2.1.1: Ultrasonics pinouts configuration, reproduced as is from [29]

In Figure 15.2.2.1.1 shows the configuration of the pinouts of a simple ultrasonic. In this project, 4 ultrasonics are used for all sides of the car. Keep in mind, VCC is the voltage provided to the sensor thus requiring 5V. GND goes to GND on the respective device.

Ultrasonic1 - Trigger Pin: 2 || Echo Pin: 3

Ultrasonic2 - Trigger Pin: 4 || Echo Pin: 5

Ultrasonic3 - Trigger Pin: 6 || Echo Pin: 7

Ultrasonic4 - Trigger Pin: 8 || Echo Pin: 9

15.2.2.2 IMU wiring

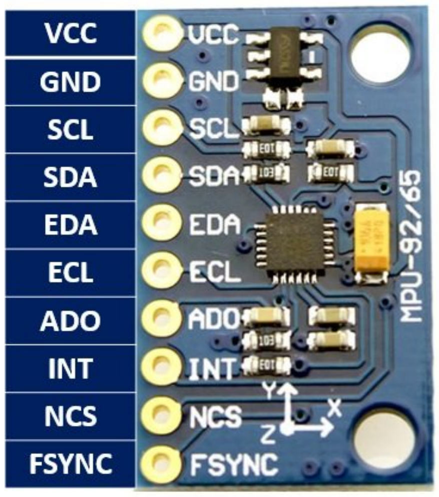


Figure 15.2.2.2.1: IMU pinout configuration, reproduced as is from [30]

In Figure 15.2.2.2.1, it can be seen the IMU pinout configuration onto what it needs to work. In this project, only 1 IMU is required to get readings. Keep in mind, VCC is the voltage provided to the sensor thus requiring 5V. SCL Pin on IMU goes to SCL on Arduino Mega. On top of this SDA pin on IMU goes to SDA on Arduino Mega. GND goes to GND on the respective device.

15.2.2.3 TF Luna wiring

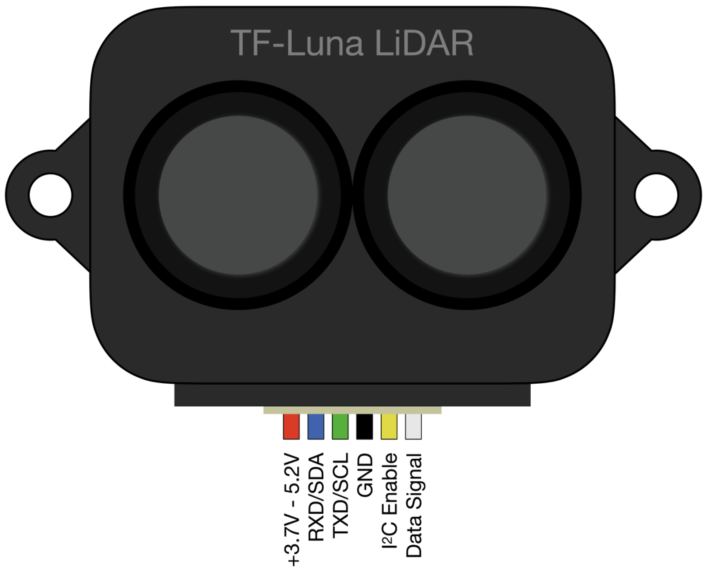


Figure 15.2.2.3.1: TF Luna wiring pinouts, reproduced as is from [31]

In Figure 15.2.2.3.1, it shows the configuration of the pinouts for a TF Luna. In this project, only 1 TF Luna is used and that is at the front of the car itself. Keep in mind, VCC is the voltage provided to the sensor thus requiring 5V. The RXD/SDA pin on the Luna plugs into a 17RX2 pinout on the Arduino Mega. The TXD/SCL pin on the Luna plugs into 16TX2 on the Arduino Mega. GND goes to GND on the respective device.

15.3 Self-Driving Videos

In this section, all the videos throughout the year are shown and talked about. Each video shows the work the team did along the way and how far they came to get the result.

15.3.1 First Trial of Donkey Car Autonomous Driving

This video shows the first trial of running a simple model of Donkey Car on Pre-Parv. The team trained this with about five laps of data around the track.

First Trial of Donkey Car Autonomous Driving

15.2.2 Simulation Trial of Donkey Car Driving

This video shows the use of a simulation model on Donkey Car. The team wanted to see how effective a simulated model would work in the real world.

Simulation Trial of Donkey Car Driving

15.2.3 Amazon Car Driving on New Track

This video shows the driving of Amazon Car on the new white track and how it reacts to the new color of the track.

Amazon Car Driving on New Track

15.2.4 Pre-Processing Video

This video shows the first trial of pre-processing on the new track so lightning will not be an issue along with any noise in the background.

Pre-Processing Video

15.2.5 Stop sign Detection Trial

This video shows the car successfully stopping at a stop sign using object detection.

Stop sign Detection Trial

15.2.6 Final Result of Donkey Car Driving on Track with Stop Sign Detection

This video displays the final result of this MQP. Displayed the two cars going around a track in opposite directions, one with object detection and one without.

Final Result of Donkey Car Driving on Track with Stop Sign Detection

14.2.7 Stop Sign Detection by Manual Testing

This video shows a member holding up the donkey car while another member introduces a stop sign.

Manual Stop Sign Detection Testing

15.4 Code to GitHub

Figure 15.4.1: Github logo, reproduced as is from [8]

To use or analyze the mPAD V3 code base, please head over to the repository where the main production code lies(Figure 15.4.1).

Donkey Custom Github Link

15.5 mPAD V3 Poster Presentation

This poster is the final poster presented during MQP presentations at WPI. It was used for the MME Department along with the CS Department.

mPAD V3 Poster

15.6 mPAD V3 Powerpoint Presentation

This PowerPoint presentation was used for the RBE presentation during MQP presentations at WPI.

mPAD V3 Powerpoint