A logo for a college of engineering

Description automatically generated

Remote Pilotless Vehicle

**Researchers:**

Gabriel Gourmet-Razungles

Electrical Engineering

California State Polytechnic University, Pomona

[gabrielg@cpp.edu](mailto:gabrielg@cpp.edu)

Benjamin Hong

Computer Engineering

California State Polytechnic University, Pomona

[bjhong@cpp.edu](mailto:bjhong@cpp.edu)

**Advisor:**

Dr. Tamer Omar

Department of Electrical and

Computer Engineering

California State Polytechnic

University, Pomona

[tromar@cpp.edu](mailto:tromar@cpp.edu)

Thursday, December 14, 2023

Harutun Minasyan

Computer Engineering

California State Polytechnic University, Pomona

hminasyan@cpp.edu

# Abstract

Today, the term “Autonomy” has grown in popularity as an increasing number of companies have been investing time, research, and resources to provide customers with a driverless experience in their cars. Although some companies have developed working prototypes that are already being sold on the market, these autonomous systems are not yet fully optimized for every situation possible. Drivers that have such autonomy in their cars are still required to be vigilant towards the road and other drivers. This report will focus on the implementation of an autonomous system mainly using a Raspberry Pi single board computer and Arduino microcontroller paired with a low cost go-kart. Lane and traffic light detection are some of the initial goals leading into a fully autonomous vehicle that can navigate marked roads and react to traffic signals. For safety and flexible options, the vehicle will also be implemented with a remote control option by connecting the Raspberry Pi to a VRX Racing Simulator, offering the possibility of human control. The Raspberry Pi will be the main computer on this vehicle, performing the most important tasks. An Arduino will also be connected and be used for less important, but still essential purposes such as battery percentage display, lighting systems, or safety systems. It will also be responsible for communicating with the motor drivers and braking system. The versatility of the Raspberry Pi makes it the computer of choice for implementing various systems. This project’s current goal is to understand the benefits of autonomy and to construct a prototype that can be used as a platform for future expansions such as a robotic arm, remote sensing, and a more complex self-driving system.

Table of Contents

[Abstract 2](#_Toc153464003)

[Introduction 5](#_Toc153464004)

[Project Planning and Task Definition 7](#_Toc153464005)

[Methodology 8](#_Toc153464006)

[Network Model 8](#_Toc153464007)

[Wireless Connection 8](#_Toc153464008)

[VPN Tunnel 9](#_Toc153464009)

[Onboard Router 9](#_Toc153464010)

[System Model 9](#_Toc153464011)

[Main Controller 9](#_Toc153464012)

[Video Streaming 10](#_Toc153464013)

[Motor Drivers 11](#_Toc153464014)

[Brake Servo 12](#_Toc153464015)

[Client Software 13](#_Toc153464016)

[pigpio Daemon 13](#_Toc153464017)

[Main Control Program 14](#_Toc153464018)

[Startup Options 14](#_Toc153464019)

[Control String 15](#_Toc153464020)

[Program Flow 15](#_Toc153464021)

[Steering Algorithm 16](#_Toc153464022)

[Active Safety State Machine 16](#_Toc153464023)

[Bus Model 17](#_Toc153464024)

[Subsystems 19](#_Toc153464025)

[Telemetry 19](#_Toc153464026)

[Lighting 19](#_Toc153464027)

[Ultrasonic 21](#_Toc153464028)

[Battery Monitoring 23](#_Toc153464029)

[OpenCV 25](#_Toc153464030)

[Standards and Constraints 29](#_Toc153464031)

[Results 30](#_Toc153464032)

[Video Streaming Latency 30](#_Toc153464033)

[Network Latency (Video Only) 31](#_Toc153464034)

[Network Latency (Video + Controls) 32](#_Toc153464035)

[Network Throughput 33](#_Toc153464036)

[Graphs and Discussion of Findings 35](#_Toc153464037)

[Conclusions 37](#_Toc153464038)

[References 39](#_Toc153464039)

[Appendix 41](#_Toc153464040)

# Introduction

Technological advancements have brought about a revolution in everyday life, work, and travel. One area that has seen significant growth is the development of autonomous vehicles. This design project aims to create a powerful and efficient pilotless vehicle that can also be remotely controlled. Starting from a base model of a cart, multiple features will be implemented onto the chassis ultimately creating a vehicle that supports multiple functions and has features that allow it to perform various tasks. The vehicle will be user-friendly and safe, with a focus on a smooth and swift motor system that can handle different terrains and environments. It will also have sensors and cameras to detect obstacles and navigate through different environments without the need for human intervention. Users will be able to monitor the battery life of the vehicle with a feature that shows the battery conditions and voltage. To enhance the vehicle's efficiency, a solar panel will also be integrated into the design. This will be a significant step forward in the development of autonomous vehicles and will help revolutionize the way we travel.

Researchers at Philadelphia University have designed and assembled an unmanned ground vehicle like this project. The vehicle is capable of streaming real time video and can be controlled manually via a mobile application through the cloud. It also has an autonomous mode where it ingests information from ultrasonic sensors and GPS to avoid obstacles. To offload the work needed for these operations, an Arduino Mega manages the inputs from all sensors and outputs signals to the motor driver. The Raspberry Pi handles high-level software such as video streaming, cloud connection, and communication to the Arduino Mega [1]. Another paper had a similar goal but instead used the ArduPilot system, which is an open-source autopilot system capable of controlling air or ground unmanned vehicles. The APM controller has been used, which is a microcontroller that runs the ArduRover firmware [2]. This is a possible avenue for integrating GPS and vehicle control.

Whether it involves autonomous driving or remote piloting from a human, the vehicle requires some kind of sight. Different options are available from the use of a lidar system to a simple camera attached to the body of the vehicle. Current vision is achieved by connecting a Microsoft LifeCam Studio webcam to the USB port of the Raspberry Pi. A more advanced device will be implemented for permanent use in the form of a Raspberry Pi V2 Camera (also known as the IMX 219 camera). The future implementation will likely utilize both cameras, one for traffic sign recognition and lane detection, and another for surveying the environment which will be mounted on a panning platform, using a servo motor, allowing the user to look around. With basic functions such as live streaming feed and image capturing, added to its compact size, the IMX219 is the ideal camera for the job. Furthermore, traffic lights and lane detection are capable through processing signals with the Raspberry-Pi with an easy ribbon connection, leaving one extra USB port available for other ideas [3].

Two researchers at a university in Thailand have achieved lane detection and lane keeping of a small vehicle using a Raspberry Pi 3B and a Raspberry Pi Camera similar to the one currently available. The paper discussed the possibility of LIDAR and GIS systems for lane and road detection, but ultimately the vision approach was chosen. OpenCV was utilized to do Canny Edge detection on lanes and an algorithm was used to determine the line angle which was used to determine the turning angle of the vehicle to follow the lane [4]. Another team of researchers from Thailand have used a Pi and OpenCV to achieve the same function. Canny Edge was also used, but the team used the Hough transform to detect straight lines. A similar algorithm was also used that involved linear slopes and tangent to calculate steering angle. The paper had also discussed the hardware system where an L298N driver was used to drive two motors [5]. It is also possible to achieve this without the use of OpenCV, as two researchers from Shenzhen University were able to use a CCD camera and fuzzy logic. The vehicle was able to follow a line at a speed of 2 m/s [6]. While the paper is somewhat outdated, the fuzzy rule table it discussed will be useful for the software that will run on the Raspberry Pi.

Continuing with computer vision, another function that is desired is the detection and recognition of traffic lights. A roadside unit will control traffic lights as well as collect telemetry data from vehicles passing by. The vehicle is expected to obey the lights on the traffic lights by observing the colors of the lights. Researchers from the University of Bahrain have used a model created using OpenCV that was trained using YOLO on a dataset of traffic light images. Using this method, they were able to achieve 76% accuracy [7]. The OpenCV script shall provide steering angle and traffic light status to the main program. Due to the computationally demanding nature of OpenCV, it may require a dedicated Pi in which the output will be sent over a serial bus to the master Pi.

To allow the pilot to view the environment the vehicle is driving in, there must be a real time video stream available to the pilot. A team of three researchers from the Kielce University of Technology have used the WebRTC protocol to enable real time streaming of video from a drone for the purpose of monitoring parking lots. WebRTC is designed for real time communication between web browsers and allows two way video and audio, a feature that is also desired for this vehicle. The authors have also written that WebRTC is an engine that operates within a browser like Google Chrome and that both the sender and receiver shall both be using the same browser versions. Data from sensors had also been transmitted via the data channel on the WebRTC session, suggesting a good outline of a possible system to build [8]. Given these requirements, a good communication channel must be established between the vehicle and the pilot.

Typical modems and routers have port forwarding in which a port on a computer may be exposed to the wide area network so that it can be accessed from anywhere in the world. Unfortunately, access to a modem with this capability is limited, so the next best option is the data connection provided by a mobile phone. However, since most mobile operators have clients behind a Network Address Translation layer, classic port forwarding is not an option. Therefore, a client-server-client model should be considered. Peter J. Burke, IEEE, has written about a self-flying plane, and one of its features is of particular relevance to this project. The video connection had been achieved using an Amazon Web Services instance and a Socks 5 proxy SSH tunnel [9]. Another paper described the network teleoperation of a wireless mobile robot where TCP/IP sockets were utilized to establish communication between the user and the robot [10]. These will be explored in the development of this project.

Although the vehicle will have a live feed from a camera, with the right LiDAR system there will be little need to look through its eyes. There are two- and three-dimensional LiDAR systems and they both have their advantages. The two-dimensional LiDAR system will be more cost effective; however it loses its perception of depth but has a faster response time due to it reporting back only a two dimensional object. The three-dimensional LiDAR system will be more powerful and provide a better set of eyes for the vehicle. It will be able to see depth and calculate if the object in front of it is tall or short and have a better understanding of what is in front of it. [11] The vehicle will be able to come to a better conclusion of what is surrounding the vehicle with a three-dimensional LiDAR system, and it will be more useful for scenarios where it would help others.

With a more powerful LiDAR system comes the issue of the requirements for the LiDAR-based object detection algorithms. According to the KITTI Dataset, Karlsruhe Institute of Technology and Toyota Technology Institute, there is a correlation between different open-source LiDAR algorithms and their precision and reaction time. In the study, they had 4 types of LiDAR algorithms they used, and they all have high precision to detect cars. However, 2 of them have lower precision while detecting pedestrians and they seem to have the same precision for bicyclists. From their results, they determined that for a LiDAR system to be considered for a vehicle, the algorithm needs to archive a frame rate over 25 Hz and only one of their algorithms that was included in the test achieved 35 Hz deeming it to be the only algorithm to be useful for real-time detection. [12] With this being said the algorithm by PointsPillars was the only algorithm in their study to achieve 35 Hz frame rate and they also have the lowest response time which is imperative to having a well-functioning LiDAR system for a vehicle.

With all these features that will be implemented, the battery will have a strenuous load on it while it operates. To monitor the life of the battery and the amount of charge it has, a voltage/current monitoring system will be implemented to allow the users to see the current state of the battery. With the use of an Arduino system that is programmed to display the current voltage state, the user will have a visual while operating the vehicle to know how much charge is left on the battery to plan out the venture for the vehicle. To create the voltage monitoring system, the current status of the battery will have to be measured and through calculations Arduino will be able to determine the SOC and SOH for the battery. [13]

In addition to monitoring the battery levels, there is another feature to be implemented to not only monitor the status of the battery but to charge it through a solar panel. The solar panel will also be implemented through the Arduino to provide charge to one of the battery cells and allow the car to charge while it is in motion to retain the battery or slow down the discharge of the battery. The solar panel will require a few modifications to the Arduino system using an MPPT, Maximum Power Point Tracking Controller to reduce the amount of voltage being generated and to convert excess voltage to current and allow for a more efficient charger. [14] The solar panel will allow for the battery to not only retain some of its charge, but it will also be able to keep the latency low due to it providing charge to the batter. This will work due to all the components requiring power and the solar panel will provide charge to keep the components working optimally and thus reducing the latency.

# Project Planning and Task Definition

Tackling an autonomous vehicle project is no easy feat as there are many different tasks to take on and everything must work well together in the end. For a project such as this one, organization is the key ingredient for continuous progress, thus different tasks will take place in a defined order that has been determined to be most efficient.

Lane detection, lidar vision and autonomy are exciting features to work with, nevertheless the fundamentals cannot be left behind. The vehicle needs to have its main functions working properly including movements forward, backwards, left turning, right turning, and finally braking. Following basic movement, control of the cart is also very important. Current control happens through the keyboard of the computer that is connected to the Raspberry Pi running the code. The immediate goal is to connect the Raspberry Pi, and therefore the entire vehicle, to a VRX racing simulator using a cloud connection using a mobile phone, thus providing a better driving experience for remote operations. The VRX racing simulator will also remain as a long-term option to drive the vehicle when a human is necessary.

The next most crucial part is getting power to all the components on this vehicle, and for that, a proper battery is needed. In addition to upgrading the current battery and displaying its values, making the circuits more efficient remains a goal.

Finally, camera improvements and field of view flexibility will be implemented onto the cart, hence opening the door to not only vision but the ability to use OpenCV and work with lane detection and traffic light detection. Such applications will set the vehicle closer to having full autonomy.

# Methodology

## Network Model

A diagram of a computer system

Description automatically generated

Figure 1: Network Model

### Wireless Connection

Ensuring a stable and secure connection between the vehicle and the pilot is critical to the vehicle’s viability out in the field. Previous iterations of this project have utilized a Sierra Wireless 4G LTE modem to provide Internet connectivity to the vehicle, however due to availability, the modem could not be used for this project. The closest available alternative to a modem was a smartphone as it utilizes the same technology to access the Internet. This project utilizes a Samsung Galaxy Note 9 for its data connection. It contains a 4G LTE modem that connects to the AT&T cellular network. Results of testing this network connection are shown in the results section of this report. This solution mostly works, however due to the nature of Carrier-Grade Network Address Translation (CG-NAT) that many carriers use today, it is difficult to allow incoming connections from the Internet to access devices behind a smartphone’s data connection. As mentioned in the introduction, there are a few options to get around this constraint.

### VPN Tunnel

The first solution uses a Virtual Private Network (VPN) to establish a tunnel between the pilot and the vehicle, and the second uses a Socks 5 proxy to forward the traffic. For ease of use and increased security the VPN method was selected. To facilitate this VPN tunnel, an OpenVPN server was set up on an Amazon Web Services T2 micro instance and TCP port 443 was opened to allow the vehicle to connect to the server. For the Linux systems onboard the vehicle and the pilot’s client software, the official OpenVPN client was used to establish the VPN connection. This creates a star topology shown in Figure 1: Network Model, where each computer connects to a central server at which traffic can be routed between any computer. Furthermore, because the OpenVPN tunnel is encrypted, the connection between the pilot and vehicle is secured.

### Onboard Router

In addition, a GL.iNet GL-AR300M16 router was used to bridge the smartphone with the Raspberry Pi and mini PC which is shown in the dotted box in Figure 1: Network Model. The motivation to use a router between the two was to reduce any latency or bottlenecks imposed by a wireless connection. This provides a central point for managing additional networked systems connected to the router. The router supports both Android and iOS smartphones under any wireless carrier, which allows for high compatibility whenever the vehicle needs Internet connectivity. The router also hosts a short-range wireless network for easy access to the onboard computers for debugging or in environments where there is no cellular reception.

## System Model

### Main Controller

The heart of the vehicle is the Raspberry Pi 3 shown in Figure 2: System Model. It runs C++ code that is discussed in a later section. The Raspberry Pi features General Purpose Input Output (GPIO) pins and other serial interfaces such as the Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Camera Serial Interface (CSI), and Universal Asynchronous Receiver-Transmitter (UART) protocol. These interfaces are used for peripherals such as the analog to digital converter (ADC), the various subsystems, the motor drivers, and the Pi camera used for live remote viewing. Since the Raspberry Pi runs the Raspbian operating system, a derivative of Debian Linux, it supports many applications and programming languages making it a powerful platform for embedded devices. Figure 3: Physical Implementation shows the physical implementation of the system. Other components such as the ADC and the Mini PC will be discussed in later sections of this report.

A diagram of a computer system

Description automatically generated

Figure 2: System Model

A machine with wires and wires on a red cart

Description automatically generated with medium confidence

Figure 3: Physical Implementation

### Video Streaming

To provide the pilot with live video, the WebRTC extension for the UV4L streaming server was chosen for the task. It utilizes a Raspberry Pi camera module using the CSI interface and streams video to any web browser with low latency and low bandwidth. It can achieve a video resolution of 1280x720 with an average latency of 360 milliseconds and an average throughput of 1.081 Mbps. These metrics were under an AT&T connection, and more information about the performance under a different carrier is discussed in the results section of this report. Shown in Figure 4: Panning Camera is the camera module mounted on a pan-tilt servo assembly. This implementation only utilizes the panning servo since the camera has an adequate vertical field of view to not require more tilting. This gives the pilot the ability to see beyond the field of view of the camera for any obstacles before moving.

A close up of a circuit board

Description automatically generated

Figure 4: Panning Camera

### Motor Drivers

To send power to the driver and steering motor, a BTS 7960 H-bridge motor driver was provided. This gives the Raspberry Pi the ability to control the speed and direction of these motors which, in turn, control the vehicle’s movement. Initially, the “WiringPi” library was used to manipulate the GPIO pins. The library had functioned as expected and was used for a large part of the development of the main software. However, when the servo motors were added to provide panning movement of the camera, it behaved very sporadically and appeared to be caused by the software implementation of PWM that the library had used. An alternative library called “pigpio” was found to be the perfect replacement as it implemented hardware timed PWM as well as PWM meant for servos. This eliminated all glitching and has been providing very stable signals. Figure 5: Motor Driver Schematic shows the connection of the motor drivers with the Pi while Figure 6: Motor Drivers shows the physical implantation of the drivers.

A diagram of a car

Description automatically generated

Figure 5: Motor Driver Schematic

A close up of a circuit board

Description automatically generated

Figure 6: Motor Drivers

### Brake Servo

Initially, the go-kart platform had a Progressive Automations PA-07-2-5 linear actuator to engage the brake calipers. While the actuator provided enough force for the task, it did not provide a feedback signal. This meant that the software was not aware of where the actuator was in space, and it also meant that the actuator position could only be controlled using preset delays. This became an issue when attempting to map the VRX simulator’s brake pedal with this actuator. It was desired to have the brake caliper match the exact position of the brake pedal, just like in a real vehicle. Without a feedback signal, the software cannot determine where the actuator is and when to stop moving it. The solution that came about was to use a normal servo motor but one that has a high torque rating. A 20kg servo motor was selected and mounted as shown in Figure 7: Brake Servo Implementation. This servo provides the necessary force to stop the vehicle and because it is controlled by a PWM signal, its position can be precisely set. Therefore, the brake caliper now mimics the movement of the brake pedal, giving the driver granular control over the vehicle’s brakes.

A close up of a machine

Description automatically generated

Figure 7: Brake Servo Implementation

## Client Software

The pilot has access to three methods of input to control the vehicle: the Thrustmaster racing wheel, an Xbox 360 controller, or a standard computer keyboard. All three client programs operate similarly in that they connect to the vehicle’s TCP server using a socket and send the same formatted string as discussed later in the control string section. Figure 8: VRX Simulator Client shows the terminal window of the client that acquires inputs from the VRX simulator. Figure 9: Keyboard Client shows the terminal window that acquires inputs from a keyboard. All three clients ask the user what address to connect to, whether that be the local IP of the vehicle (if connected to its wireless network) or the VPN IP address (if connected to the VPN). In the case of the keyboard client in Figure 10: Keyboard Client User Input, it asks for the acceleration value to use when the up or down arrows are pressed, since the input is either pressed or not pressed.

A screenshot of a computer

Description automatically generated

Figure 8: VRX Simulator Client

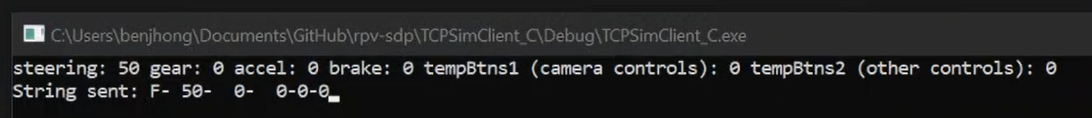


Figure 9: Keyboard Client

A computer screen with white text

Description automatically generated

Figure 10: Keyboard Client User Input

## pigpio Daemon

One issue that arose during the development of this project was how to share the I/O exposed by pigpio with different, independent programs on the Raspberry Pi. Initially, the main control program used the “pigpio” library to get access to and manipulate the various I/O. This was fine, and recommended for projects where only one program would need to access it. However, it meant that one, and only one, program could access the I/O. When it came to developing additional applications that needed access to the I2C bus, it became apparent that only one program would be allowed to run. The solution was to use the “pigpiod” library, where the “d” stands for daemon. Instead, this library connects to the daemon socket which is directly connected to the Pi’s I/O. It acts like a middleman between a program and the I/O. This allows more than one application to connect to that socket to gain access to the I/O. Figure 11: Daemon shows this stack, from I/O to the different programs.

A diagram of a program

Description automatically generated

Figure 11: Daemon

## Main Control Program

### Startup Options

The main control program is responsible for all functions of the vehicle. When the program starts, it initializes all I/O and displays the result of the initialization shown below in Figure 12: Initialization Results. If the pigpio library successfully connects to the daemon, it provides a handle that must be used by all its functions to point it to that socket connection.

A screenshot of a computer

Description automatically generated

Figure 12: Initialization Results

If any one of these fails to initialize, the program will show which I/O device did not initialize and will continue running. Next, it asks the user the same question asked in the client program, which is to specify the IP address to listen on. Then it asks whether to enable the serial port to receive movement commands from the lane detection system powered by the mini PC. Lastly, it asks whether to enable the active safety system which is discussed later in this section. These questions are shown below in Figure 13: User Options:

A screenshot of a computer

Description automatically generated

Figure 13: User Options

### Control String

Before discussing the main program, the control string must first be introduced. As mentioned above in the client software section, there is a standardized format that the control string must follow. The string must be composed of the following values separated by delimiters shown in Figure 14: Control String Format:

A computer code with text

Description automatically generated with medium confidence

Figure 14: Control String Format

The parsing logic expects this format only. Steering, acceleration, and braking values are supplied to the vehicle using the three fields shown in Figure 14: Control String Format. Shown below in Table 1: Button Set 1 Codes and Table 2: Button Set 2 Codes are the meanings of the different button set codes.

Table 1: Button Set 1 Codes

|  |  |
| --- | --- |
| Code | Command |
| 0 | Nothing/default |
| 1 | Pan camera left |
| 2 | Pan camera right |
| 3 | Center camera |

Table 2: Button Set 2 Codes

|  |  |
| --- | --- |
| Code | Command |
| 0 | Nothing/default |
| 1 | Signal right |
| 2 | Signal left |
| 3 | CV control |
| 4 | Headlights |

### Program Flow

After the user answers the startup questions, the program opens the socket and waits for a connection. When a connection gets established, the program enters a loop and first checks if serial communication was enabled. If it was, it checks the number of bytes available at the serial buffer. If it is greater than or equal to 17 bytes, it gets copied into a character array. Next, the program checks if the socket’s receive buffer is empty. If it is, it means that there is no data being sent and thus is assumed to be a connection fault. The program waits here until it starts getting data. When it does, it grabs the computer vision (CV) control flag from the socket buffer and sets the CV status LED on the vehicle accordingly. This LED is visible from the live streaming camera and tells the pilot if the vehicle is currently being driven by themselves or the computer vision system. At this point, there are two control strings present at both buffers. Both are formatted identically, except one came from the pilot’s client program and the other from the computer vision system. The CV control flag determines which one enters the parsing function. The function parses out the values shown in Figure 14: Control String Format and passes them to various “run” functions such as *run\_steering* and *run\_acceleration* to carry out different actions. The entire loop is delayed by 50 milliseconds to allow the two buffers to fill up without getting interrupted.

### Steering Algorithm

One challenge was getting *absolute* steering positioning to function correctly. Since it was desired to have the steering wheel mirror the vehicle’s wheels just like how the braking servo functioned, the steering motor needed a feedback signal. Luckily, the linear actuator that was already on the go-kart had a feedback signal provided by an internal potentiometer. Since the Raspberry Pi did not have a native ADC, an MCP3008 was used to capture the analog signal from the potentiometer. The MCP3008 operates using the SPI protocol and because the pigpiod library does not have functions that can directly talk to it, a direct bit-level approach was needed. The implementation is discussed in Appendix A. Unfortunately, due to the nature of analog signals, simply reading this value and using it to determine which direction to move the actuator until it reached the target position did not work. Since the ADC was picking up noise and giving fluctuating values it caused the actuator to violently oscillate around the target position. There were many attempts to reduce the noise in hardware by using RC filters or even software-based filters using moving averages, but they did not yield any satisfactory results. The solution was simple; implement a dead band in which to ignore changes less than a set value. The function that implements this is shown in Figure 15: Steering Algorithm:

A computer screen with text and images

Description automatically generated

Figure 15: Steering Algorithm

### Active Safety State Machine

The Active Safety (AS) system is a safety system designed to prevent the vehicle from crashing into obstacles. The inputs to the state machine come from the ultrasonic subsystem via I2C where two sensors placed in the front and rear provide distance data to the main program. Whenever the vehicle senses an obstacle closer than a set threshold, the drive motor is deactivated, and the brakes engage. It does not leave this state until the pilot lets go of the accelerator pedal. When the pilot releases the pedal, the vehicle continues halting. If the pilot depresses the pedal again, the vehicle re-engages the drive motor and relaxes the brakes. It allows the pilot to move away from the obstacle for a brief period. If an obstacle is still detected after this brief period, the vehicle halts again and repeats the same process. If there are no obstacles, the vehicle is allowed to move normally. A state machine was modeled after this and implemented as shown in Figure 16: Active Safety State Machine.

A diagram of a flowchart

Description automatically generated

Figure 16: Active Safety State Machine

The AS flag represents the return value of the function *run\_active\_safety()* which returns 0 if no obstacle is detected and 1 if either the front or rear ultrasonic sensor detects an object within 35 centimeters. The HF gets asserted whenever the state machine decides that the vehicle needs to come to a stop. If it is asserted, the program overrides the acceleration value to 0 and brake value to 100. If not, it passes the two values through.

## Bus Model

When the team was developing new features such as the ultrasonic subsystem, it was clear that certain features of the vehicle should be made modular and separate from the main program. For example, the measurement program requires delay functions to wait for a signal from the ultrasonic sensors. This would add harmful delay to the main program. Furthermore, it was difficult to write code in one large program and collaborate. To distribute the load of writing code and to enable modularity in the system architecture, the team decided to offload these new features onto separate Arduino microcontrollers and have them communicate with the master Raspberry Pi using a serial bus. Initially, the Controller Area Network or CAN bus was chosen for its robustness and the fact that the bus consisted of two wires. While this would have been a perfect solution, CAN bus requires more hardware and software overhead to operate. There was not enough documentation on CAN bus implementation on Raspberry Pi to make it practical. In the search for a new protocol, I2C became more and more practical for this application. Since it also featured a two-wire implementation and the master/slave principle, it was a good alternative to CAN bus. Figure 17: I2C Bus Model shows the bus model and how each subsystem is connected to the bus.

A diagram of a computer

Description automatically generated

Figure 17: I2C Bus Model

Since the Raspberry Pi operates on 3.3V logic levels and the Arduino on 5V logic levels, a level shifter was needed to prevent the Arduino’s I2C port from damaging the I2C port on the Raspberry Pi. The bus begins after the level shifter and any device can be added by connecting it to the SCL and SDA wires. The level shifter is shown below in Figure 18: Level Shifter:

A close up of wires

Description automatically generated

Figure 18: Level Shifter

When the main program starts, every subsystem is initialized using *i2c\_open()* which takes in the pigpio handle as well as the hardware address of the subsystem. The address is shown in brackets for every subsystem in Figure 17: I2C Bus Model. The function will return the handle of the subsystem which gets used by subsequent functions that interact with it.

## Subsystems

### Telemetry

The telemetry subsystem is responsible for collecting the vehicle’s location, wheel speed data, and sending it to the main program whenever requested. It is powered by an Arduino Uno where the vehicle location is collected using the NEO-6M GPS module and the wheel speed is measured by a Taiss E38S6-600-24G rotary encoder. This data is requested by a Python script which runs separately from the main C++ program. The Python script utilizes the Extensible Messaging and Presence Protocol (XMPP) to send this data to the Roadside Unit (RSU) for it to capture, process, and store. The flow of this script is depicted below in Figure 19: Telemetry Acquisition Script:

A diagram of a process flow

Description automatically generated

Figure 19: Telemetry Acquisition Script

Since the telemetry subsystem is an integral part of the Road Side Unit (RSU), more detailed information about this particular subsystem is available in the report for that project. The Python script is the only piece of software that runs on the vehicle’s Raspberry Pi that pertains to this project.

### Lighting

The lighting subsystem gives the vehicle lights that a standard vehicle has such as brake lights, turn signals, and headlights. These have been implemented onto the vehicle to improve pedestrian safety for those walking near the vehicle. The brake lights have been configured to turn on whenever the brake servo is engaged. Turn signals are initiated using the second button set shown in Table 2: Button Set 2 Codes. The turn signals blink with a fixed interval set by the delay() function. An additional light that was added was the CV control LED. It is placed at the front of the vehicle in view of the camera. Whenever the CV control flag is set, the LED is turned on to tell the pilot that the vehicle is taking commands from the OpenCV subsystem, rather than the pilot’s controls.

Whenever the Raspberry Pi commands the lighting subsystem to turn on a light, the Arduino Nano receives the I2C command and manipulates the 6-channel relay board that is connected to its digital pins. The relay board is shown below in Figure 20: 6-Channel Relay Board:

A close-up of a circuit board

Description automatically generated

Figure 20: 6-Channel Relay Board

The lighting subsystem is wired as shown below in Figure 21: Lighting Subsystem Schematic. The relay board acts as an electronic switch that turns a light on or off based on the input at the INX pins. Since the relays require 12 volts to actuate, the board is supplied with the main 12V rail that runs throughout the vehicle. The voltage the relays are switching is 5 volts, as per the LED requirements.

A diagram of a circuit

Description automatically generated

Figure 21: Lighting Subsystem Schematic

### Ultrasonic

An objective we had for this project was to implement an autonomous mode in which the user would be able to put a path via GPS and the kart would drive itself to the end point. For any autonomous mode in any vehicle, the use of an ultrasonic sensor is key to its success. In our case we were weighing the options of using a 2-D LiDAR sensor or using the HS-04 ultrasonic sensors which are relatively inexpensive and rather easy to implement. Another issue with the 2-D LiDAR sensor would be the range of operation is only what is in its 2-dimensional plane. We opted to use the HS-04 ultrasonic sensors instead because they are smaller in size, perform the same task as the LiDAR and it has a 15 degree of a cone angle which would allow us to receive data from a 3-dimensional view compared to the limited 2-dimensional LiDAR sensor. [15] With our choice of the sensor complete, we move onto the implementation of the sensors themselves and the mounting. We started off with 4 sensors, one in the front, one on each side and then we had one in the back just in case if we want to reverse and we do not have a reverse camera so that one sensor helps with the reversing. When we first created the system, it worked well. However, under load we noticed noise was being introduced by the motor drivers to the sensors. To combat this, we moved the motor drivers away from the sensors and then we also used a thicker 14-gauge wire that has shielding on it to protect from noise interference for the sensors. Once we confirmed the noise interference was gone, we started implementing it with I2C to send the data over to the raspberry pi through an I2C serial bus and integrated it with our Active Safety System. When the kart is in its autonomous mode, the Active Safety System can be activated and once active, the kart will use the data coming from the sensors to determine where the closest object is in its path. The kart will then come to a stop if the sensor reports that an object is getting closer to the kart than our allotted threshold, it will apply the brakes of the kart and come to a stop before coming in contact with whatever object is in its path.

A circuit board with many wires

Description automatically generated

Figure 22: Ultrasonic Subsystem Schematic

In Figure 22: Ultrasonic Subsystem Schematic, we have the implementation of the ultrasonic sensors with the Arduino to create the safety system. We have our sensors connected to the 5-volt power port in parallel, so each sensor receives the 5 volts it needs to power on. With the use of PCB prototype boards and male to female header connectors we can create the parallel connection of the 5-volt power to be provided to each sensor along with the grounds to have them all powered properly. In addition, each echo and trig pin are connected in their own analog inputs to have their own independent data transmission to not bridge with another sensor and send mixed data. From here we also connect the SDA and SCL ports to the I2C serial bus to transmit the data to the pi and be able to receive the data and push it through to the Active Safety System for the user to know the distances for the kart and for the autonomous mode as well.

A screen shot of a computer code

Description automatically generated

Figure 23: Code for Distance Measurement

In Figure 23: Code Distance Measurement, we have the distance measurement function and we used this to format the data we receive through our echo pin which is the distance we get from the sensors to the closest object in its field of view. The formula for calculating the distance is based on the duration of the ping from the echo portion of the sensor to travel to and back from the closest object in its path multiplied by the speed of sound converted to the cm/microsecond. We know the speed of sound is 343 which would convert to 0.0343 cm/microsecond to display the correct distance based on our unit of choice and our delay.

A screen shot of a computer program

Description automatically generated

Figure 24: I2C Request Handler

In Figure 24: I2C Request Handler, we have a snippet of the code used for the ultrasonic sensor system. In the snippet, we have the request function of the code to send the data it receives from the sensor to an array for the I2C bus to have it implemented in the Active Safety System. It reads the data input from the sensors and then gets formatted in a different function and then the formatted data would pass through as a distance in centimeters. At the end we only used the front and back sensor due to some complications and just a reduction of the amount of data we were transmitting from the Pi to save bandwidth.

### Battery Monitoring

The battery monitoring system is designed for the sole purpose of the user knowing the status of the batteries attached to the kart. Previous groups were not able to have a display set to know about the status of the system’s battery and would run into issues where the kart would shut off mid run not knowing the status of their battery. In our iteration of the kart, the benefit of this system would help us due to us having the same issues and the batteries themselves would discharge due to their nature and had a lower battery life due to constant discharge. The battery monitoring system includes an Arduino, I2C display and two voltage dividers to handle the input voltage of our two 12-volt batteries. With the battery monitoring system, we implemented a connection splitter on the kart where the two batteries would attach to power multiple devices on board and the motor for the kart. We connected it to the splitter to receive the voltage for the battery and then send the data to Arduino. The use of the voltage divider is necessary due to the Arduinos nature of only being able to handle 5-volt input and we had two 12-volt batteries which would overload the analog inputs of the Arduino rendering it useless. The system is set up to send the voltage to the divider first and then we send the divided voltage to the Arduino. It is a simple 5x divider, however we noticed that was not the case and we had to reverse engineer the multiplier to get an accurate reading which was 3.25675263. Now, the Arduino is set up to read from both batteries and display each battery status at a 5 second interval giving us enough time to know what the status of our batteries are. The code has been set up with the I2C library to talk to the display and the basic math library for easy functions. The code also has a switch statement to alternate between 12-volt and 24-volt batteries to provide accurate readings for the user of the kart.

A circuit board with wires and a screen

Description automatically generated

Figure 25: Battery Monitoring System Schematic

Figure 25: Battery Monitoring System Schematic encapsulates the diagram for how the battery system is connected and it is rather quite simple. You would take the input voltage from the batteries to the voltage divider to protect the Arduino and then we can connect it to the analog inputs from the Arduino. Then we use the SCL and SDA ports to connect to our I2C display to have a visual output. [13] In hindsight we should have implemented the system with the karts screen using the I2C bus so the user would have the knowledge of the battery status as well. We could have done this by splitting the SDA and SCL signal and taking one of the connections to the display then taking the other to the kart. In addition to this, we were implementing a different way of calculating the percentage where we would find the voltage of the battery once the kart turns off and divide that by the full voltage to get a more accurate percentage. However, each battery would have a different drain due to the system itself. The battery powering the motor would drain quicker and more due to the power draw in comparison to the other battery just powering peripherals and the mini pc and Raspberry Pi.

A screenshot of a computer program

Description automatically generated

Figure 26: Code to Read Voltages

In Figure 26: Code to Read Voltages, we have a snippet of the code for the Arduino battery monitoring system. The code snippet shows the main loop of the program to read the voltages through the analog inputs we have enabled and to apply the correct multiplier before sending the data to the display. The snippet also includes the display function for enabling the screen along with the format of the screen to enable the output to be correctly displayed.

### OpenCV

OpenCV, a free open-source software made of a library of programming functions used for real-time computer vision. In this project, OpenCV’s main purpose is to be used in providing lane detection for the go-kart. OpenCV being a new implementation for the kart, much learning was needed to know how to start and what to implement in the code to get lane detection. As a part of this learning process, an article that was of tremendous importance in shaping OpenCV knowledge was Matt Hardwick’s project on the same subject [16]. Through the information, the rest of the lane detection was made possible.

As a start, there needs to be a way to implement OpenCV onto the cart, using a device with enough computing power to accommodate the frame captures and analysis. Although the onboard Raspberry Pi could be used for such a task, adding more tasks to it could slow down the entire program and hinder the kart’s operability. Subsequently, to undertake the lane detection’s task, an ASUS CN62 Chromebox computer is used shown in Figure 27: ASUS CN62 Chromebox.

A black square device with ports

Description automatically generated

Figure 27: ASUS CN62 Chromebox

The Chromebox is operated by an i5-5500U CPU, 8GB DDR3 of RAM and 32GB of SSD storage. These specifications provide the necessary computation power required. Furthermore, it is running the Ubuntu 22.04.3 LTS operating system which allows access and operation of OpenCV. The small size of this computer makes it easier to implement on board the kart thus allowing for connection to the Raspberry Pi easily. To power the computer, the 12V power supply for the entire cart is connected to a boost converter, stepping the voltage up to 20V.

For this kart, the lane detection works through Python code, written on the integrated development environment PyCharm, which is installed on the machine. PyCharm provides an easy to operate environment for the user and has an abundant number of sources online that can be useful for debugging a code, including the system itself.

For the code itself, there are three main parts that can be distinguished: image capture and processing, communication, and video feedback. Image capture is the main component of lane detection as it is where the whole process starts. The camera used is an Uhuru UW-002, and its sole purpose on this cart is to capture the image for the code to process it. The camera is strategically placed in the front of the cart to best capture the two lanes in front. Upon starting the code, the camera is activated through the command: cv2.VideoCapture(0). Following up, the program will initiate its main function called ‘pipeline’ that will perform the main detection of the lane. Two external functions are used in this code, namely “region\_of\_interest” and ‘draw\_lines.’ Although these function’s names seem self-explanatory, they still require some explanation. The function ‘region\_of\_interest’ essentially extracts a region of interest for an image provided by creating a polygon inside the dimensions of the image and masking out all around it so that the program can focus on the inside of the polygon, which will contain the lanes. ‘Region\_of\_interest’ will be used in the pipeline function as shown in Figure 28: Hexagon Shape Region of Interest:

A screenshot of a video

Description automatically generated

Figure 28: Hexagon Shape Region of Interest

The next essential function “draw\_lines” also seems self explanatory but it is not as simple as it seems. This function essentially creates a copy on the image captured, takes a set of lines that match the location of the lanes detected drawing them on a blank image and merges them with the original capture. This function is essential as the detection of the lines must be accurate for the pipeline to work correctly.

Before moving on to the ‘pipeline’ function, communication must be mentioned first as it is being used in conjunction with the other function. The main way the on-board computer relays information to the Raspberry Pi is through serial communication. The following line of code initializes the serial connection (ser) to communicate with the Pi at a baud rate of 115,200 bits/s:



Figure 29: Serial Object Instantiation

All starting here, computer vision will be able to send commands to the Pi to direct the kart accordingly. Thus, we have the “send\_serial\_command” function that does exactly that, sending a string format message using the ‘str.format()’ method. For this specific case, the message template is the following:



Figure 30: Control Message Format

“F” is to remain constant for communication, the information in the curly brackets is changed depending on the need of the kart to turn left or right, and the second value shown corresponds to the speed of the kart. The speed can also be changed interactively, however for simplicity of testing, it remained the same and the steering remained the focus. The format must remain exactly the same in order for the Raspberry Pi to recognize what is required of it.

The main program’s function is the ‘pipeline’ function, the brains of the lane detection. Its main task is to implement a processing pipeline for live lane-detection combining edge detection, Hough transform and lane tracking. Edge detection is made possible through grayscaling the image provided by the camera, thus creating a distinct two-tone image, and using “Canny Edge Detection” implemented in OpenCV, there is a detection of the change of color shades of the image. The Hough Transform comes in afterwards with the purpose of identifying the lines within the designated region. This is done by representing lines as points in a polar coordinate system with each point corresponding to a line in the Hough space, and with a multitude of points, the actual captured line can be traced. The Hough Transform can also be modified based on the capture conditions/quality by changing the following parameters in Figure 31: Hough Transform.

A screen shot of a computer program

Description automatically generated

Figure 31: Hough Transform

represents the distance resolution of the accumulator in pixels. Decreasing this value would result in a finer detection of lines at the cost or more lengthy computation.

defines the granularity at which of the angles at which the lines are detected. Decreasing would allow for detection of lines at finer angles.

The determines the minimum number of points (Hough lines) to be considered for a valid result. Increasing this value would require stronger evidence for a line to be detected but could lead to fewer lines detected.

The determines how long a line needs to be to be considered helping shorter lines to be excluded.

Finally, determines the maximum gap between detected lines allowed to be considered as a single line. Increasing this value would potentially for longer lines.

The next step is to separate the lines into two groups, one on the left, and one on the right, and this can be done by determining the slopes of the lines detected.

A computer screen shot of a code

Description automatically generated

Figure 32: Slope Calculation

Thanks to the region of interest, any other insignificant line is filtered out of the image, as also shown in Figure 28: Hexagon Shape Region of Interest. The next step in this lane detection is the calculation of the kart’s position relative to the calculated center of the lanes. Such positioning is calculated by finding the start and end point of the left and right lines based on the polynomial fits of the drawn lines.

A screen shot of a computer program

Description automatically generated

Figure 33: Calculated Center

At this point, image processing and kart positioning are set and what’s left is to communicate with the Raspberry Pi what to do if the kart drifts to the left or to the right, and so, using the items above, the position is relayed to the ‘send\_serial\_command’ function which is constantly in contact with the Pi.

A computer screen shot of text

Description automatically generated

Figure 34: Steering Correction Algorithm

A screen shot of a computer code

Description automatically generated

Figure 35: Steering Function

Finally, when it all runs together, the cart stays with the two lines detected and runs autonomously unless command is taken over by the user.

# Standards and Constraints

The standard that this vehicle must conform to is the Internet Protocol (IP) and the Transmission Control Protocol (TCP) as it will be utilizing the cellular network provided by an Internet Service Provider. To ensure that the vehicle can communicate on standard computer networks, the main C++ program uses the socket functions from the standard UNIX libraries like socket.h. For the Windows-based client programs, the programs use the ws2tcpip.h header for the same functions.

Another hardware standard that the vehicle had to conform to are the various serial protocols such as SPI, I2C, and UART that are used. This was achieved through the use of WiringPi at the start of the project, then pigpio.

One constraint that was upon the project was that the vehicle must be remotely controllable from anywhere in the world. This was achieved using a mobile phone and the cellular network, as described in the network model. In theory, the vehicle can be controlled anywhere a domestic carrier has reception.

Another constraint was that the vehicle must be controllable from the VRX simulator. Fortunately, the Thrustmaster wheel that the simulator comes with is compatible with Xbox controller libraries which is what was used to acquire data from the peripherals. This also means that the vehicle can also be controlled using an Xbox 360 controller with little code modification.

# Results

A series of tests were run with multiple trials to determine the performance of our communications system and whether it satisfies the initial project requirements. All tests were performed while the vehicle was connected to our AWS OpenVPN server via a 4G LTE connection. The cellular connection was provided by a Samsung Galaxy Note 9 for AT&T and an iPhone XS for T-Mobile. The pilot was connected to the Internet using the university’s lab Ethernet connections.

## Video Streaming Latency

The first test is a latency test of the WebRTC live video stream to determine how much delay there is between the real world and what is perceived on the pilot’s screen. This was performed by displaying a synchronized stopwatch on the pilot’s screen (Pilot) and showing a phone screen with a synchronized stopwatch to the camera (Remote). Screenshots were taken at regular intervals and recorded as samples. An example of a screenshot is shown below in Figure 36: Sample Screenshot, where the left is the remote and the right is the pilot.

A screenshot of a computer

Description automatically generated

Figure 36: Sample Screenshot

Table 3: WebRTC Video Latency over AT&T

|  |  |  |  |
| --- | --- | --- | --- |
| **Time (sec)** | **Remote (sec)** | **Pilot (sec)** | **Latency (sec)** |
| 0.0 | 55.1 | 55.5 | 0.4 |
| 1.0 | 56.1 | 56.4 | 0.3 |
| 2.0 | 57.1 | 57.4 | 0.3 |
| 3.0 | 58.1 | 58.5 | 0.4 |
| 4.0 | 59.1 | 59.5 | 0.4 |

Table 4: WebRTC Video Latency over T-Mobile

|  |  |  |  |
| --- | --- | --- | --- |
| **Time (sec)** | **Remote (sec)** | **Pilot (sec)** | **Latency (sec)** |
| 0.0 | 30.0 | 30.3 | 0.3 |
| 1.0 | 31.0 | 31.3 | 0.3 |
| 2.0 | 32.0 | 32.3 | 0.3 |
| 3.0 | 33.0 | 33.2 | 0.2 |
| 4.0 | 34.0 | 34.3 | 0.3 |

## Network Latency (Video Only)

The second test is a ping test between the pilot and the vehicle. This test shows the round trip time for a packet to be transmitted between the pilot, the vehicle, then back to the pilot. It was performed using the built-in “ping” command in Linux.

Table 5: Network Latency (Video Only) AT&T

|  |  |
| --- | --- |
| **Time (sec)** | **Latency (ms)** |
| 0.0 | 124 |
| 1.0 | 151 |
| 2.0 | 105 |
| 3.0 | 106 |
| 4.0 | 134 |
| 5.0 | 96.8 |
| 6.0 | 122 |
| 7.0 | 75.9 |
| 8.0 | 103 |
| 9.0 | 96.2 |

Table 6: Network Latency (Video Only) T-Mobile

|  |  |
| --- | --- |
| **Time (sec)** | **Latency (ms)** |
| 0.0 | 117 |
| 1.0 | 141 |
| 2.0 | 121 |
| 3.0 | 90.9 |
| 4.0 | 134 |
| 5.0 | 108 |
| 6.0 | 129 |
| 7.0 | 88.7 |
| 8.0 | 109 |
| 9.0 | 112 |

The meaning of “video only” in Table 5 and 6 specifies that the ping test was performed in conjunction with an active WebRTC video stream only.

## Network Latency (Video + Controls)

The third test is identical to the second test apart from the TCP socket providing vehicle controls. This test shows the same round trip time as described earlier, but now accounts for any latency imposed by the TCP socket.

Table 7: Network Latency (Video + Controls) AT&T

|  |  |
| --- | --- |
| **Time (sec)** | **Latency (ms)** |
| 0.0 | 112 |
| 1.0 | 109 |
| 2.0 | 93.1 |
| 3.0 | 88.1 |
| 4.0 | 115 |
| 5.0 | 101 |
| 6.0 | 102 |
| 7.0 | 92.7 |
| 8.0 | 139 |
| 9.0 | 158 |

Table 8: Network Latency (Video + Controls) T-Mobile

|  |  |
| --- | --- |
| **Time (sec)** | **Latency (ms)** |
| 0.0 | 109 |
| 1.0 | 95.7 |
| 2.0 | 125 |
| 3.0 | 127 |
| 4.0 | 132 |
| 5.0 | 126 |
| 6.0 | 89 |
| 7.0 | 118 |
| 8.0 | 177 |
| 9.0 | 145 |

## Network Throughput

The last test performed was a network throughput test to measure the amount of data being moved per second. During this test, a WebRTC video stream was active and the vehicle was being controlled using a TCP socket. The Linux application “nload” was used to measure the instantaneous throughput of the network connection.

Table 9: Network Throughput (Video + Controls) AT&T

|  |  |
| --- | --- |
| **Time (sec)** | **Throughput (Mbps)** |
| 0.0 | 1.25 |
| 0.5 | 1.4 |
| 1.0 | 1.69 |
| 1.5 | 1.27 |
| 2.0 | 0.80 |
| 2.5 | 0.95 |
| 3.0 | 1.15 |
| 3.5 | 1.0 |
| 4.0 | 0.63 |
| 4.5 | 0.67 |

Table 10: Network Throughput (Video + Controls) T-Mobile

|  |  |
| --- | --- |
| **Time (sec)** | **Throughput (Mbps)** |
| 0.0 | 1.55 |
| 0.5 | 1.54 |
| 1.0 | 1.57 |
| 1.5 | 1.53 |
| 2.0 | 1.56 |
| 2.5 | 1.66 |
| 3.0 | 1.61 |
| 3.5 | 1.56 |
| 4.0 | 1.60 |
| 4.5 | 1.59 |

## Graphs and Discussion of Findings

Figure 37: Graph of Video Streaming Latency shows the latency of the WebRTC video stream over the LTE connection. The latency is consistent across both carriers with AT&T having an average of 0.36ms and T-Mobile with 0.28ms. This latency is felt when driving the vehicle but is low enough to not be a significant driving impairment. Lower latency connections such as 5G would minimize this delay and provide near instantaneous feedback.

Figure 37: Graph of Video Streaming Latency

Figure 38: Graph of Network Latency with Video Streaming shows the round-trip time for a packet to travel within the network. The latency appears to swing frequently, likely due to the nature of cellular networks. The utilization of a cell tower at any given moment is always changing and is likely contributing to the fluctuating latency. However, even in the worst case of 151ms, it has little effect on the pilot’s ability to maneuver the vehicle. The perceived response of the vehicle from the pilot’s controls is still very usable. The average of this sample set is 111.39m**s** for AT&T and 115.06ms for T-Mobile.

Figure 38: Graph of Network Latency with Video Streaming

Figure 39: Graph of Network Latency with Video and Controls shows the same unit of latency as in Figure 38: Graph of Network Latency with Video Streaming, however it also considers the possible latency imposed by the TCP socket for vehicle controls. The same kind of fluctuation is seen here and is likely due to the same external variables discussed in the paragraph above. However, taking the average of this set yields 110.99ms for AT&T and 124.37ms for T-Mobile, which indicates that the TCP socket has negligible effect on the overall network latency, even when actively sending new data through the socket.

Figure 39: Graph of Network Latency with Video and Controls

The last test is shown below in Figure 40: Network Throughput in Normal Operation, which depicts the instantaneous network throughput during normal operation with video streaming and vehicle controls. The fluctuations in throughput here are likely due to WebRTC’s compression algorithm continuously optimizing its data transfer. The average of this set is 1.08 Mbps for AT&T and 1.58 Mbps for T-Mobile, which is very friendly to today’s cellular connections that can handle much higher bandwidth. It allows for remote viewing even in low signal environments. This was one of the primary motivations behind selecting WebRTC over other streaming protocols like RTSP.

Figure 40: Network Throughput in Normal Operation

# Conclusions

Taking on an autonomous vehicle project has proved to be much more of an endeavor filled with loads of learning and mistakes along the way. Essentially starting from scratch by scrambling to get the basic motor functions of a vehicle running, one ends up learning a lot about how the vehicle operates. Many problems present themselves as more components are added, but with it, the opportunity to create and circumvent such roadblocks with creative solutions.

From getting the kart running to implementing a battery monitoring system to getting a router and VPN set up to talk with the kart and fixing up loose ends. These were the beginning of what was going to be improved upon and evolve into our solidified version. With the basics achieved, our new goals for the kart to be autonomous and the ability to have different modes of control. To help get the autonomous mode going, the implementation of the ultrasonic sensors helped by having an object detection system to aid the kart at times when in the autonomous mode.

The implementation of lane detection in this project serves as a critical point in the development of more autonomous features for the kart. Through testing and tuning parameters such as the Hough Transform, or changing the region of interest, the kart can perform its duties on varying terrain and conditions. This ensures that the kart’s capabilities and this project overall is not adapted to a single scenario but emphasize on adaptability and effectiveness of the kart.

Looking at all the tests performed, it appears that the carrier being used for the vehicle’s data connection has a slight effect. For network latency in cases of video only and video + controls, AT&T takes the lead in having 3.67ms lower latency in video only and 13.38ms lower latency in video + controls. However, when it comes to streaming WebRTC video, T-Mobile exhibits 0.08ms lower latency as compared to AT&T. T-Mobile also comes out ahead when looking at network throughput with 0.496 Mbps higher throughput.

Despite these results, the conclusions drawn should not be a final say over which carrier is best for this application. There are many variables involved in the network performance of cellular devices, whether that be current cell tower usage, physical obstacles, or if the handheld device is running any network-intensive background tasks, to name a few. In such applications as the remote pilotless vehicle, a dedicated cellular modem paired with an appropriate data plan should be used to provide enhanced reception, latency, and speeds.

The extended goal for this project is to set the stage for future students to take on the challenges of autonomous capabilities. By establishing a framework that is easy to follow and understand, the next generations can continue improving and building upon the past. The hope is to have an educational tool for others to be ready for more advanced integrations.

# References

[1] T. A. Tutunji, M. Salah-Eddin and H. Abdalqader, "Unmanned Ground Vehicle Control using IoT," 2020 21st International Conference on Research and Education in Mechatronics (REM), Cracow, Poland, 2020, pp. 1-5, doi: 10.1109/REM49740.2020.9313890.

[2] S. Vishnu, M. Anil Kumar, M. G. Manjesha, Z. Pasha and S. Madhu, "A Low-Cost Prototyping Approach for Autonomous Unmanned Ground Vehicle for Real-Time Surveillance," 2022 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET), Kota Kinabalu, Malaysia, 2022, pp. 1-6, doi: 10.1109/IICAIET55139.2022.9936808.

[3] Thorat, S. Mahadik, S. Mane, S. Mohite, and A. Udugade, “Self Driving Car using Raspberry-Pi and Machine Learning,” International Research Journal of Engineering and Technology, Mar. 2019.

[4] S. Viwatpinyo and S. Phatchuay, "The Automatic Car to Implementation of Lane Detective using Raspberry Pi 3 Model B on OpenCV," 2022 International Conference on Cybernetics and Innovations (ICCI), Ratchaburi, Thailand, 2022, pp. 1-5, doi: 10.1109/ICCI54995.2022.9744158.

[5] K. Paponpen, K. Sucharitpongpan, N. Termsaithong and P. Chaipunya, "The Implementation of Steering Angle Estimation on Miniature Raspberry Pi-based Autonomous Car," 2022 IEEE 17th Conference on Industrial Electronics and Applications (ICIEA), Chengdu, China, 2022, pp. 1037-1042, doi: 10.1109/ICIEA54703.2022.10005980.

[6] J. Qiu and G. Cao, "Steering control of smart car based on image sensor," 2009 3rd International Conference on Power Electronics Systems and Applications (PESA), Hong Kong, China, 2009, pp. 1-3.

[7] N. H. Sarhan and A. Y. Al-Omary, "Traffic light Detection using OpenCV and YOLO," 2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Sakheer, Bahrain, 2022, pp. 604-608, doi: 10.1109/3ICT56508.2022.9990723.

[8] A. Chodorek, R. R. Chodorek and K. Wajda, "Media and non-media WebRTC communication between a terrestrial station and a drone: the case of a flying IoT system to monitor parking," 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT), Cosenza, Italy, 2019, pp. 1-4, doi: 10.1109/DS-RT47707.2019.8958706.

[9] P. J. Burke, "A Safe, Open Source, 4G Connected Self-Flying Plane With 1 Hour Flight Time and All Up Weight (AUW) <300 g: Towards a New Class of Internet Enabled UAVs," in *IEEE Access*, vol. 7, pp. 67833-67855, 2019, doi: 10.1109/ACCESS.2019.2917851.

[10] X. Xue, S. X. Yang and M. Q. . -H. Meng, "Remote sensing and teleoperation of a mobile robot via the Internet," *2005 IEEE International Conference on Information Acquisition*, Hong Kong, China, 2005, pp. 6 pp.-, doi: 10.1109/ICIA.2005.1635147.

[11] B. Behroozpour, P. A. M. Sandborn, M. C. Wu and B. E. Boser, "Lidar System Architectures and Circuits," in IEEE Communications Magazine, vol. 55, no. 10, pp. 135-142, Oct. 2017, doi: 10.1109/MCOM.2017.1700030.

[12] Dai, Z.; Wolf, A.; Ley, P.-P.; Glück, T.; Sundermeier, M.C.; Lachmayer, R. Requirements for Automotive LiDAR Systems. Sensors 2022, 22, 7532. https://doi.org/ 10.3390/s22197532

[13] “Multicell Voltage Monitoring for Lithium Battery Pack in Electric Vehicles,” circuitdigest.com. https://circuitdigest.com/microcontroller-projects/multi-cell-voltage-monitoring-for-lithium-battery-pack-in-electric-vehicles (accessed Mar. 24, 2023).

[14] Admin, “Designing of MPPT Solar Charge Controller using Arduino,” *How To Electronics*, Jan. 21, 2023. https://how2electronics.com/designing-of-mppt-solar-charge-controller-using-arduino/

[15] R. Santos, “Complete Guide for Ultrasonic Sensor HC - SR04,” Random Nerd Tutorials, 2019. <https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/>

[16] M. Hardwick, “Simple Lane Detection with OpenCV,” Medium, Aug. 22, 2018. <https://medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0>

‌

‌Appendix

Consulting the datasheet revealed that the program needs to send three consecutive bytes {1, 128, 0} to initiate a reading from channel 0. The 1 represents the start bit, 128 represents 10000000 where 1 indicates a single-ended measurement and the following three zeroes indicate the channel to read from. The remaining zeroes are don’t-cares. The ADC replies with the measurement represented in 10 bits which are split across two bytes as shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte # (spi\_rx[]) | 0 | 1 | 2 |
| Bit # of measurement | xxxxxxxx | xxxxxx98 | 76543210 |

Using the code shown below, the complete 10-bit number can be assembled:

spi\_xfer(pi, adcHdl, spi\_tx, spi\_rx, 3);

adc\_data = (spi\_rx[1] << 8) | spi\_rx[2] & 0x3FF;

# 