EGR 4830

Autonomous And Self Driving Vehicle

Cal Poly Pomona

Tyler McNeil, Edwin Huape, Kevin Cardenas

Professor Omar

November 20th, 2024

Abstract

Autonomous vehicles are beginning to expand in the car industry and will soon become the future of cars. What exactly is an autonomous vehicle you may ask? An autonomous vehicle is a self-driving car that can observe the environment, operate, navigate, and make decisions safely without the need for human intervention. These types of cars involve advanced sensors, actuators, and other artificial intelligence systems. In the project we will be presenting, we are tasked with taking a previous RPV (Remote Piloted Vehicle) and making it universally autonomous while establishing the foundation for future groups to build off of. The vehicle being worked on is a go-kart that stems from the company Razer after their ground force drifter line of vehicles became discontinued. It has received attention from a previous group that left a platform for the modifications our group intends to add. The vehicle is powered through a 24-volt system, and operated by a 4G hotspot and Raspberry Pi. This report will provide an overview of our implementation of the methodologies described, such as adaptive cruise control through the use of lidar, GPS following via GPS module, and general hardware fixes and upgrades. We will also look into the effects of self-driving vehicles on our society and what we may benefit from.

Table of Contents

Introduction

In today's ever-growing society, technologies are pushed to their limits in their exponential growth and improvements. We have watched as phones transformed from big, blocky, and cumbersome forms into something near alien compared to what they used to be. Computers evolved from massive machines into tiny silicon chips that fit on the tip of your finger. The list of technological improvements is endless, so it only makes sense that we would want the same for one of the most used items in our lives, a car. When they first were created, they barely reached sustained speeds of 10 MPH and used more resources than what it was worth. Now we have cars that can reach up to 300 MPH with many intricate quality of life upgrades. With the addition of computers to their part lists, cars gained newfound abilities never thought possible even in the early 2000's. We exist at a time when cars can control their speed and even accurately judge the distance between themselves and their environment. This leads to the next obvious step in the technology of cars, completely automating the driving process and allowing the car to have full control.

Self-driving vehicles are the future of mobility. They are continuing to be developed with features that were not expected and the tech behind these vessels is intricate. As they have grown, more knowledge has been gained in being able to figure out and understand what these vehicles are capable of, and what basic features are needed to be able to run properly. An autonomous vehicle operates without the need for someone to intervene and can perform everything a skilled driver can do. The hardware part of the vehicle is in charge of detecting environmental characteristics and passing the information to the device while the software can process the information from its surroundings to decide what actions it needs to take. For such a car to operate, the vehicle needs to consider things such as braking, accelerating, and steering. For instance, accelerating can be done by controlling the input such as current, and be done by a computer. Braking is more complex since it needs numerous cylinders and pistons to function which in turn makes electronic controllers simple and effective in friction braking [1].

There are five levels of autonomous vehicles according to the Society of Automotive Engineers. Starting from level 0 which has no automated features and requires the driver to be in full control. Level 1 vehicles are equipped with one or more automated features such as cruise control but do require the driver to do all the tasks. Level 2 has two or more primary features such as lane-keeping that relieves the driver from controlling these functions. Level 3 vehicles have features that make it possible for the driver to stop having control but are expected to have the driver take control when needed. Level 4 vehicles can perform driving even if the driver does not respond to a request. Lastly, level 5 is a fully autonomous vehicle without a driver present. Autonomous vehicles are equipped with a plethora of GPS, cameras, ultrasonic sensors, prebuilt maps, radar, lidar, inertial navigation systems, and dedicated short-range communication [2].

Of course, like any new change to daily life, there are both serious consequences and potential scenarios that can come from such an act. One might assume that if you were given the ability to multitask in your vehicle, companies may try to capitalize on such an increase in personal time and use it as justification for assigning more work [3]. There's also a level of

liability that goes into having technology control your vehicle. If the vehicle lost control and there's no way to control it, like in a level 5 SDV case for instance, there could be lives lost with no one able to be held liable since the car itself was in control. While women and men express different concerns on the matter, they all can also agree that the cost of any SDV is something that could be a concern as well [4]. Speaking in terms of purely negative attributes of these cars, the list goes on to include many sensory issues for the sensors alone. For instance, road conditions, weather conditions, traffic conditions, accident liability, and radar interference are all something that needs to be taken into account for SDV vehicles. Such interferences prove to be an issue as they could cost more than just damage to the environment, but could even cost lives in populated areas.

It is good to know how to overcome these challenges because these are issues that are out of our current control and are all an obstacle that still needs to be improved upon for these types of vehicles to be viable in society. With such a grim prospect, why would it be something we even consider doing for ourselves? There are some before-mentioned advantages, such as multitasking while on your drives. By being able to turn travel time into anything time, the possibilities vastly open up. Long grueling road trips become train rides where even the “driver” can watch the environment come and go from place to place. To play devil's advocate, having the time it takes to get to your destination as a moment to double-check or finish last-minute work helps with work efficiency. Along with other safety features, we have a forward collision warning [5] that allows the vehicle to alert drivers of a potential collision by analyzing the speed and distance from an object distance and preventing a collision, it enhances awareness and reduces the chances of front-end collisions. If done entirely properly, there wouldn’t be any traffic or congestion as the cars could navigate without hesitation and thus avoid traffic problems in general. An LA commute could take only 25 minutes even at peak rush hour as every car would participate in high speeds without the worry of crashing. Such a future is possible with the advancement of SDVs.

The idea of a self-driving car is the next step in a car's evolution, however creating such an invention is no easy feat. There are plenty of issues that arise even with how powerful and accurate the tech has become. The issue begins with how complex the act of driving is, and due to the process being very reactionary, it is hard for an autonomous vessel to understand and handle [6]. Quick reaction times rely on a good amount of “perception” sensors to be able to witness the world. As sensors get better, there will be more options to combat this issue of lack of perception, but as it stands, even though the technology might work, nothing compares to and will be able to emulate human logic or instincts to make a split-second decision [7]. This doesn’t leave us inept in any way, however. Alongside our physical technologies getting better, we have also had breakthroughs on the software side as well. AI continues to expand and prove its capabilities within our job fields, so it is of little surprise that it has been tested for an occasion such as this. By using deep learning to our advantage, we could potentially unlock the piece of the puzzle that gives a car what it needs to operate on its own. While it has only been the center of tests for self-driving vehicles so far, there have been focuses on road, lane detection, vehicle

detection, pedestrian detection, drowsiness detection, collision measurement and avoidance, and traffic sign detection [8]. As the name implies, deep learning would help allow a self-driving vehicle to learn the environment around it as it is driving, and even be able to save its experience to be referred upon later. This would mean that in well-driven areas, the cars would perform at their best since there would be a confident library of experience to run off of. The concept can even be expanded upon by connecting each car on the road to a similar server where drive experiences can be shared across a large network and thus create an even bigger reference library. This is similar to deep learning in general with technologies, however, the car's perception would be unique due to the types of sensors that can be paired with deep learning.

Hundreds of sensors and actuators make up the vehicle's sensory system. These systems include, but are not limited to, navigation and guidance, driving and safety, and performance. Navigation and guidance use GPS and LORAN radiolocation, driving and safety needs a 360-degree view as well as front and back view to sense objects on the road, and performance needs power management from several control units [9]. Sensors are used for the safety modules to be able to detect the environment to avoid any type of collision utilizing sensors such as 16-line laser radar, two single-line laser radars, millimeter-wave radars, and five cameras just to name a few [10]. Sensors can detect surroundings by cameras or by bouncing light impulses or radar signals off of objects. Some advantages of these cars are traffic congestion reduction as well as fewer accidents. Self-driving vehicles normally have sensors such as LIDARs which are a detection system that works on the principle of radar but uses light from a laser and camera, in addition to radar. Adding sensors allows the vehicle to detect not only obstacles and vehicles on the road but also maintain an idea of its environment and situation which is very important to keep the vehicle aware of its surroundings when changing lanes or avoiding collisions [11]. Lane assist is used as a safety precaution to prevent drivers from merging into other lanes that utilize surround-view cameras that offer a 360-degree view of the vehicle, reducing the number of blind spots given to the driver [12][13]. In general, however, the bulk of the sensors these types of vehicles utilize are ultrasonic sensors to determine the distance of items in the environment.

By sending out small frequencies, and waiting for their response, the sensor gains an idea of how far something may be. These types of sensors are split into 3 separate categories based on their range; short, medium, and long range [14]. Ultrasonic Sensors, which are one of the most important, play a big role in these advanced systems that are currently in vehicles [15]. Ultrasonic Sensors accurately detect objects, and persons based on distance, and are one of the best in terms of detecting closeby objects, offering reliable performance compared to other technologies that are currently being implemented in newer vehicles. The Tesla Vision, to name an example, works as a camera-based system instead of an ultrasonic providing a different perspective and take on an SDV. Even with ultrasonic sensors surrounding the car, there is still the issue of unpredictable circumstances, the biggest one being weather. As you can imagine, when it rains, sensors like these struggle as rain can interrupt them or give false readings. As technology has improved, there have been some ideas on how to properly circumvent these issues and pay the weather no mind. The current path to this idea has been a fusion between

radar systems and cameras which has proven to reduce the missed detection rate of autonomous vehicles in severe weather [16]. During a specific build and test, engineers used a software called Teraki to be able to merge an Ouster LIDAR with an HD camera which was an impactful tool since it was great for data processing [17].

As mentioned above, one of the features that self-driving vehicles might have is automatic emergency braking [18]. This system uses sensors such as radar, lidar, cameras, ultrasonic sensors, and a control module with the braking system to apply the brakes automatically when there is no input from the driver or user. Radar sensors emit radio waves to detect the moving object from a given distance and speed. Lidar sensors use lasers to create a 3D map of the vehicle's surroundings to give a better picture of what it has around it, and through the use of screens, even display this information to the driver. All this system needs is a control module to build bridges between all these sensors allowing for all sorts of features and safety nets.

With each type of sensor, we experience different pros and cons in terms of system resources and technicalities. Some of the pros of radars are that they have less data to transfer than most sensors and work well in rain and snow; however, the cons are lower resolution for screen displays, and they require multiple units for 360-degree view. Lidar works well in low light and creates accurate 3D maps of surroundings, but the cons appear with fog and rain and require a direct line of sight for them to be worthwhile [19]. Cameras provide color and character recognition while also being affordable, but lack a grand field of view and tend to experience issues with changing lights and shadows. Ultrasonic range sensors can provide accurate info in short range but boast a limited range making them ineffective at gauging speed. Lastly, for GPS, the pros are worldwide coverage provided by satellites as well as any weather operation; however, its cons fall within its accuracy depending on the number of satellites in the field of view. These are a few of the pros and cons that are involved in autonomous vehicle sensors for them to properly function, which can affect their efficiency and accuracy [20].

Most of the features that were described are important to consider when trying to create a self-driving vehicle. One that can be implemented and has slowly been making its way onto the market is traffic light and stop sign control, meaning that it can identify traffic signals by using cameras and GPS data to also determine at what speed the vehicle should be going. This would mark a final leap for SDV technology and usage as you wouldn't have to input any user control once a successful model of these sensors was to come out. The GPS data is responsible for locating the intersections where stop lights and signs may be, and monitors traffic light status [21]. As we delve into our project, there is a major idea that we have to keep in mind while creating our designs. In official builds, there exist several different architectures that determine the flow of thought for these vehicles. The biggest and most important ones are the perception system, the decision-making system, and the autonomy system which all control the reactionary parts of the autonomous car [22]. These systems are split between hardware and software, feeding each other information in mere nanoseconds to result in actions. The software that an SDV runs is responsible for a lot of different factors and detections. The system is responsible

for traffic light detection, lane detection, vehicle detection, obstacle detection, and pedestrian detection. Systems are being implemented such as anti-locking braking or electronic stability control that have been used for a while and are being added in parallel to all these new detections and code. In addition, these vehicles are equipped with new systems such as advanced driver assistance and crash avoidance systems so they can have more redundancy of safety when driven. Software can also involve perception layers, localization layers, and mapping layers which the vehicle can use to operate with several control units sending commands to the actuators and components. This enables the vehicle to accelerate, brake, steer, and perceive everything around it. In addition to these kinds of controls, the software can display the statuses of each of these variables or even pictures and live video to help the driver further understand the perception of the car.

In such intricate systems, we still find too many variables that can cause issues in reaction time, especially surrounding the traffic light system we all react to. This is where our addition to the project comes into play. Most self-driving cars in these modern times are connected to something called the ITS or intelligent traffic systems [23]. The issue with this idea is that it requires an upgrade to both the vehicle and the traffic system that's already in place around the United States. This makes for a very costly upgrade however serves as the easiest method for ensuring the vehicle knows what color the light is. While GPS offers very good location and traffic updates, it would need to be paired with the ITS to fully know what the light's status is. In our project, we attempt to create our own ITS by setting up the traffic light in the lab to send the vehicle data, as well as establish a connection to the vehicle to make it responsive to the traffic light seen in the lab.

All these features have their benefits and challenges. They offer driver assistance, road safety enhancements, the potential to solve traffic jams and congestion, and even make family road trips more possible. In terms of challenges, we might see software issues relating to reaction times or even corrupted code, hardware sensors or parts eroding or simply breaking, and even environmental problems such as weather and the aforementioned traffic conditions. Law and regulations for SDVs continue to be updated year by year, illustrating that there are limitations and room for improvement towards the performance of these advanced features. This is a significant change that the automotive industry is having, and as we push further into the future, there will continue to be more effort on development in advancing the technologies that these vehicles rely on to provide the best driver assistance and safety [24].

Methodology

Network Model and Stability Problems

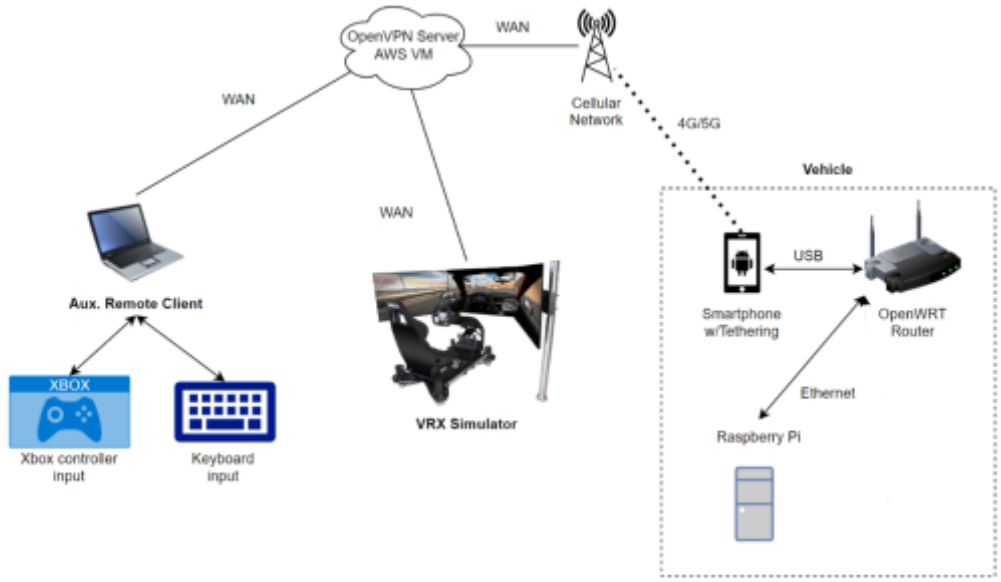


Figure 1: Network Model

This diagram shows a remote control system for a VRX simulator and a vehicle using an OpenVPN server on AWS, connected through WAN and cellular networks (4G/5G). The remote client uses an Xbox controller, VRX Simulator, or keyboard to control the vehicle. A smartphone with USB tethering acts as a modem to an OpenWRT router, which links to a Raspberry Pi. Due to a setup like this, there are potential network instability issues due to reliance on cellular connections and VPN for remote operation, however, it is sufficient for our purposes. Initially, there was a Mini PC attached to the vehicle that was able to do lane detection, however, it has since been depreciated in this project.

In terms of the router, one major problem that we faced was that the hardware was set up in such a way that DC power to the router continuously fluctuates. With the power source being a 12-volt battery, we should see no drop in power, yet we discovered that too much power was being drawn simultaneously in some cases due to the motor being in the same circuit as the hardware. In addition, a converter that steps the voltage down is used to power the system and many major components, however, with them being on the same battery, there are cases where we can drop voltage. For instance, as seen in Figure 2, the component that is in charge of turning the 12-volt battery voltage into a usable 5 volts produces an unstable 5 volts when there is too much power draw, or the battery isn't fully charged. On the 12-volt input side, if there's a drop in voltage or the battery runs slightly out of power so it no longer produces exactly 12 volts, the chip can't properly transfer the voltage over, resulting in fluctuations. On the 5-volt output side, we see an issue arise with both the router and motor taking from the same source. As the Raspberry Pi runs its programming and distributes commands, it causes fluctuations on the 5-volt output side that cause the router to turn off momentarily, meaning that the connection to the actual router is unstable. This causes random disconnects that reset the entire system, meaning the user must go through the long connection process again to reconnect to the vehicle.

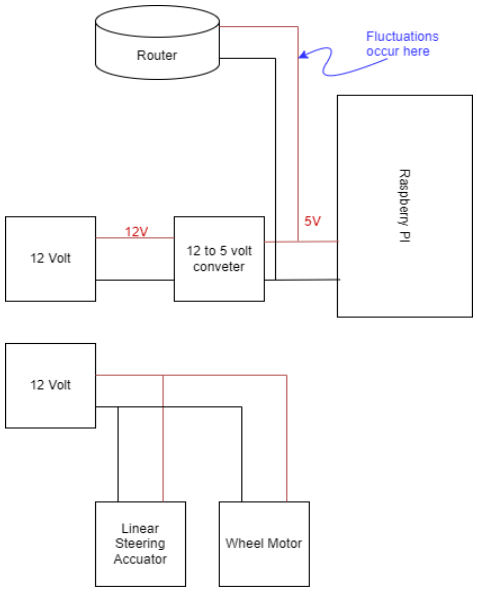


Figure 2: Fluctuations Diagram

One implementation we will be using is a capacitor in charge of keeping the stability of the system intact so even when the batteries aren't charged to the fullest, we never see a drop in the necessary voltage for the system to operate. This concept can be seen visually in Figure 2 of this report.

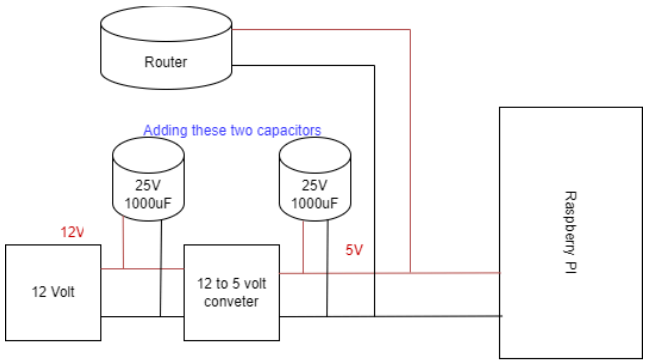


Figure 3: Adding Two Capacitors Diagram

With the addition of these 2 capacitors into our circuit, our hardware should be resistant to voltage drops due to a fluctuation problem and will be able to operate more confidently even when the 12-volt batteries drop in voltage slightly.

RPV System Model

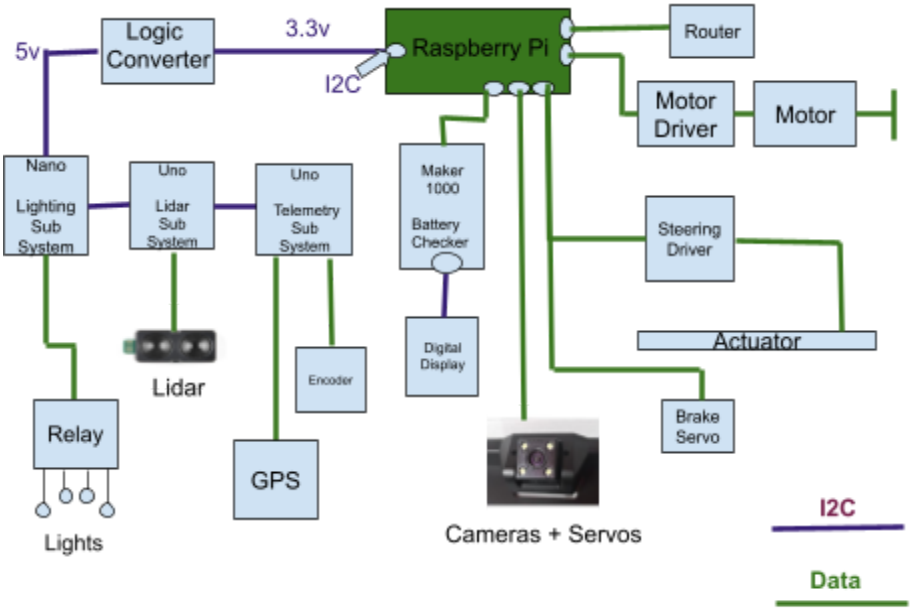


Figure 4: RPV System Model

Figure 4 shows the system where the Raspberry Pi 4 acts as the central controller, managing various subsystems in a vehicle-like setup. It controls the steering motor, drive motor, and brake servo through motor drivers. A steering angle potentiometer provides feedback through an ADC. In addition to all this, the system integrates a Pi Camera for live streaming network connectivity is established via a router/modem tethered to a smartphone for internet

access. Various communication protocols (I2C, SPI, UART) link the components for real-time control and feedback.

I2C Bus Model

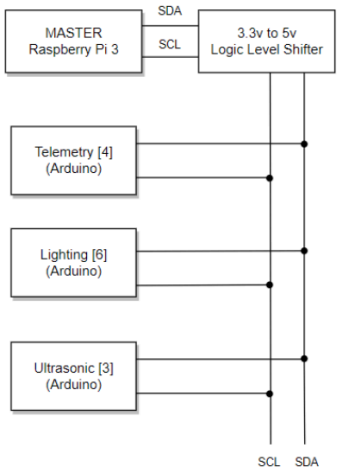


Figure 5: I2C Bus Model

This diagram shows an I2C communication setup where a Raspberry Pi 4 (operating at 3.3V) communicates with multiple Arduinos (operating at 5V) via a logic level shifter. The Raspberry Pi serves as the master, and the Arduinos (Telemetry, Lighting, and Ultrasonic) are the slave devices. The logic level shifter adjusts the voltage levels between the Raspberry Pi and the Arduino, allowing them to share data over the I2C bus using the SDA (data) and SCL (clock) lines. This particular setup makes the car's systems modular, and adding to this system in the future should be easier as there are plenty of addresses available for any system a team would want to add.

Hardware Overhaul

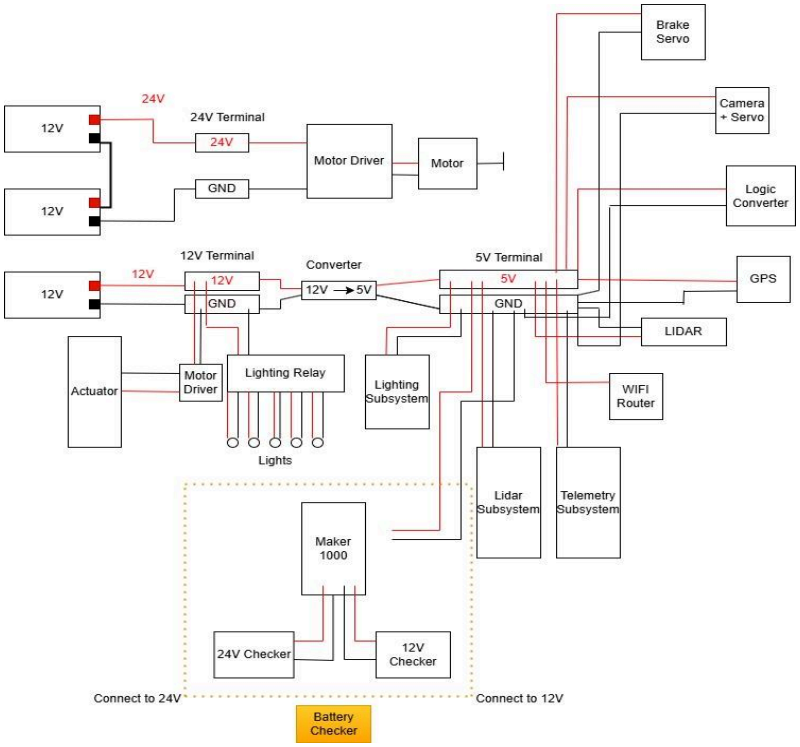
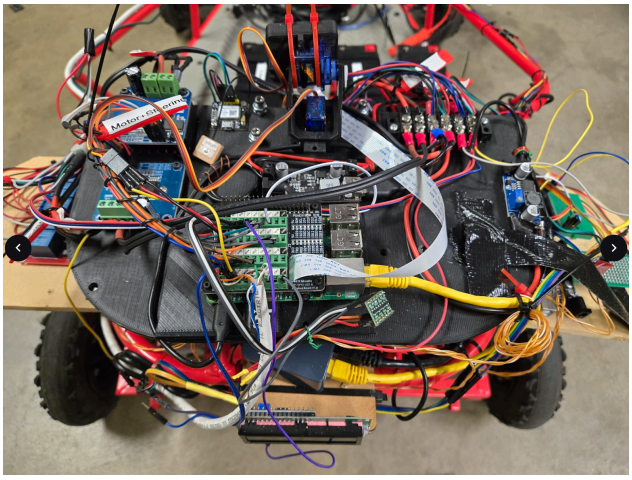


Figure 6: Power Structure

In this project, we were faced with a major problem left over from previous teams. Several of the controllers were wired in series both in grounds and +5 V, not to mention completely unorganized with controllers and components randomly taped to the frame of the vehicle. In Figure 6 we illustrate how we changed the wiring to better not only the circuit but also the organization of the overall vehicle. In the figures below (Figure 7, 8) we can see a before and after picture of our edits displaying the organizational changes.



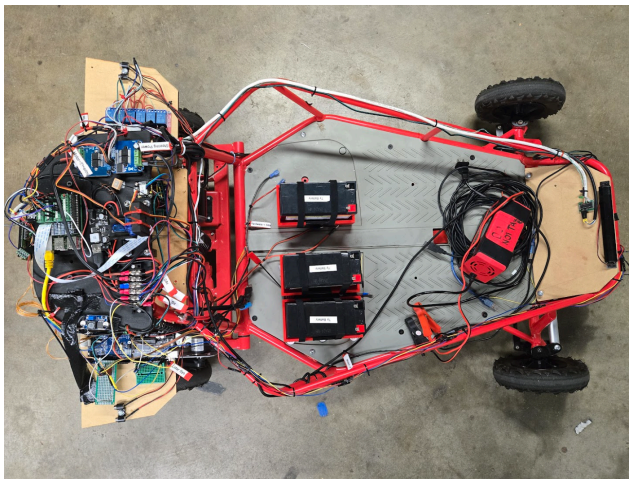
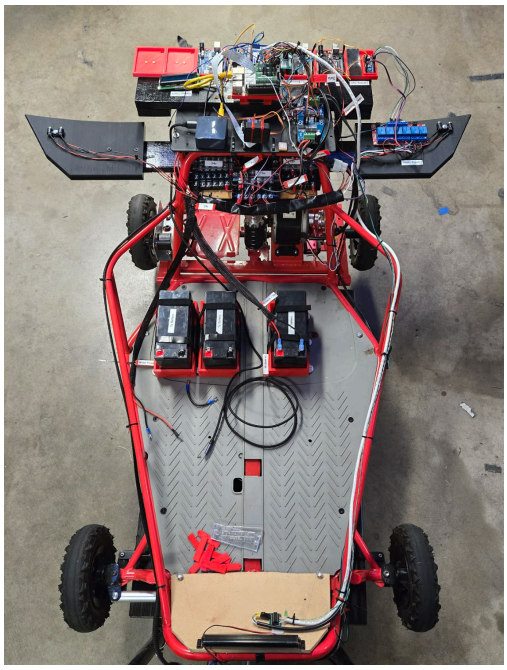


Figure 7: Before Rewire



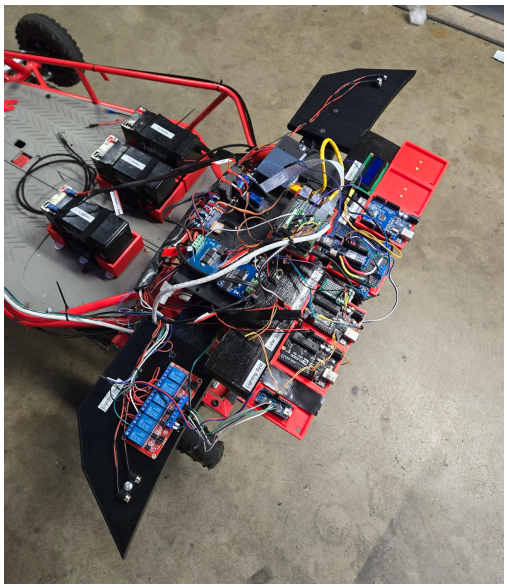


Figure 8: After Rewire

In addition to the new circuitry, we also labeled all of the wire buses making it easier to tell where each group goes. The microcontrollers are now lined up in an array that makes it easier to wire subsystems to the I2C bus making future additions easier and more streamlined. The hardware has received an entire upgrade due to this rewiring which in turn causes the software to work better since there are no fluctuations or disconnects.

Increased Speed Implementation

As a part of the hardware overhaul, we also wanted a change in the speed/performance of our vehicle. In the process of implementing this speed increase, there were many factors to consider and account for such as current draw and voltage. The motor used to be implemented with only 12 volts powering it, which was ineffective as our motor is a 24-volt component. Before our rewire, the steering actuator and motor were paired together in the same circuit. In doing so, the motor could barely operate and would cause major fluctuations throughout the car. At first, we separated the elements pairing the actuator and hardware together, and wiring the car's motor to 24 volts through a relay. This proved to be inefficient as the relay bypassed the motor driver and took the ability to control the car with PWM away. This meant no going backward, the vehicle could only go forward as no waves were telling it to go back. We then wired it through the motor driver and left the motor isolated as opposed to trying to pair it with other components, and that's when our vehicle reached new speeds. Connecting the pre-existing batteries in series from Figure 9 below provided us with the maximum voltage demand of the motors. In this configuration, we doubled our speed from around 11 MPH to 23 MPH in perfect conditions.

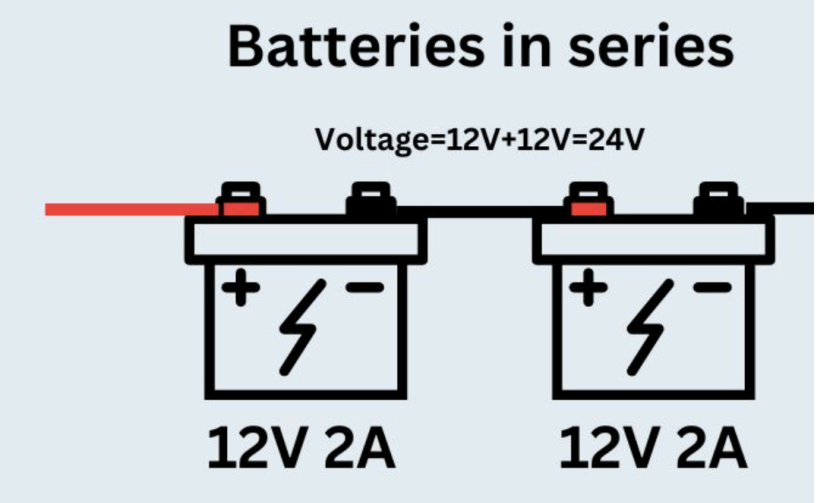


Figure 9: Batteries in Series
Renogy CA (Superior Energy Solutions)

Figure 10 below, shows a representation of a block diagram that illustrates the before and after of the controls, inputs and outputs, motor, and what is used throughout the implementation of speed control.

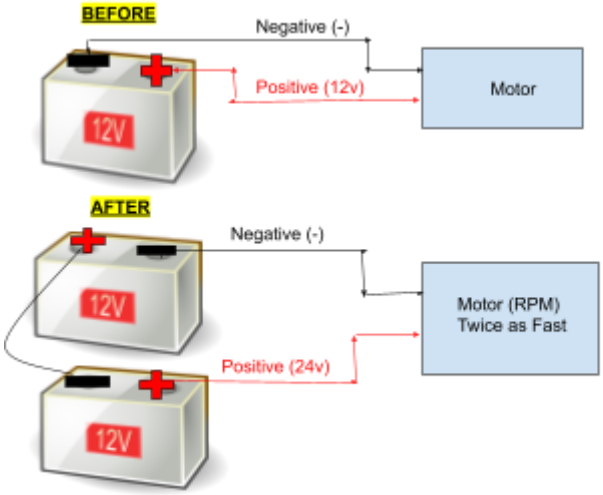


Figure 10: Before and After batteries

This diagram shows the effect of connecting two 12V batteries in series to power a motor.

- **Before**: A single 12V battery powers the motor at a regular speed (12V input).
- **After**: Two 12V batteries are connected in series, producing a combined 24V, which makes the motor run twice as fast due to the increased voltage.

The series connection increases the voltage supplied to the motor, resulting in higher RPM (revolutions per minute) and faster operation.

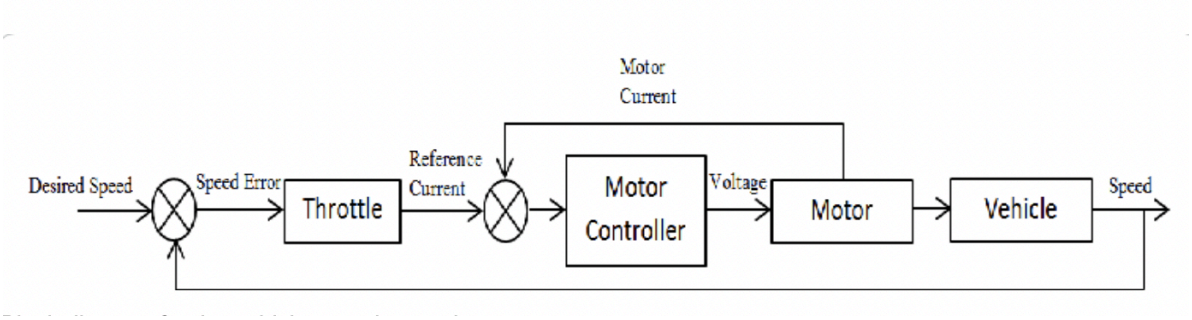


Figure 11: Speed Control Block Diagram (Research Gate)

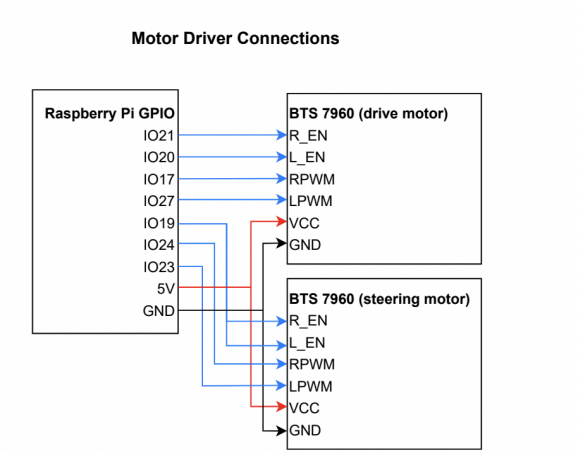


Figure 12: Motor Driver Connection Diagram

The two figures above (Figure 11, 12) give insight into how the drivers are connected, and how the logic works.

Adaptive Cruise Control Implementation

This vehicle has the implementation of adaptive cruise control. Adaptive cruise control automatically adjusts the speed of the vehicle to maintain a safe distance from an item ahead of it. To execute this function we added a lidar to be able to monitor the upcoming traffic and vehicles. This is critical for the vehicle to maintain a safe following distance and stay within a certain speed limit that is set by the user. In the figures below (Figures 13, and 14), we see the lidar sensor itself and how it is wired to the Arduino subsystem.

Figure 13: Lidar Sensor

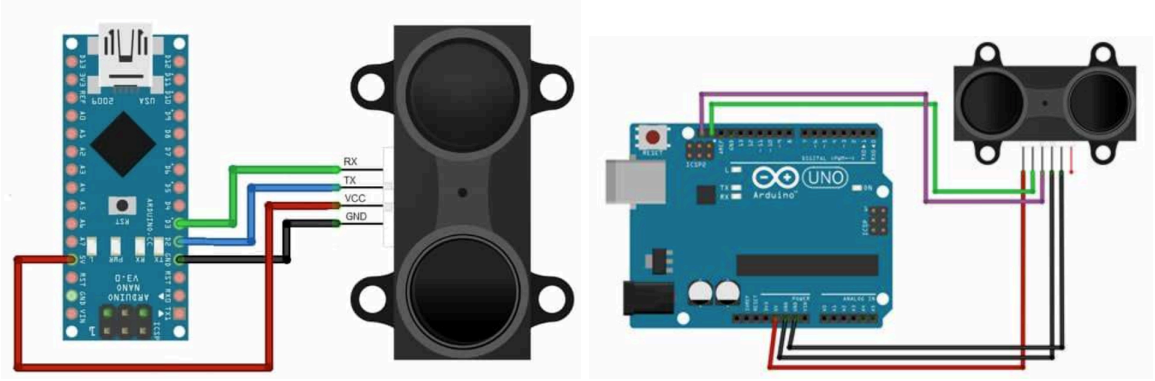


Figure 14: Lidar wiring diagram with Arduino

The lidar sensor has been added to the front of the vehicle to detect the distance and speed of an obstacle ahead. This will ensure the vehicle is careful while operating on its own and acts as a low-level safety feature. Figure 14 below demonstrates the adaptive cruise control coding block diagram. Furthermore, it is essential to ensure that the brakes do not activate at the same time as the motor because we do not want the vehicle to do both commands at the same time. If it were to run the brakes and motor at the same time it would create a stall in the vehicle or rub the brake pads down to inefficient levels. The lidar is implemented using an Arduino however the main processing unit, Raspberry Pi, is in charge of keeping the vehicle at a safe distance from any nearby objects and maintaining a certain speed throughout its driving. In addition, the processor is also in charge of the code that would make the car brake if an object gets too close to it and accelerate after when it is safe to do so.

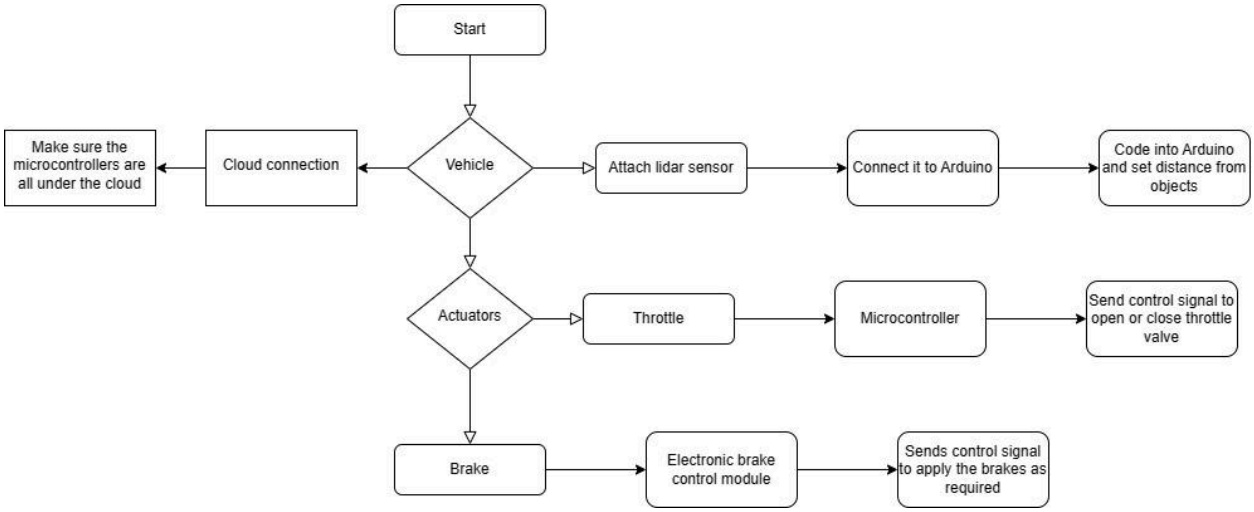


Figure 15: Adaptive Cruise Control Block Diagram

Lidar System Testing Points

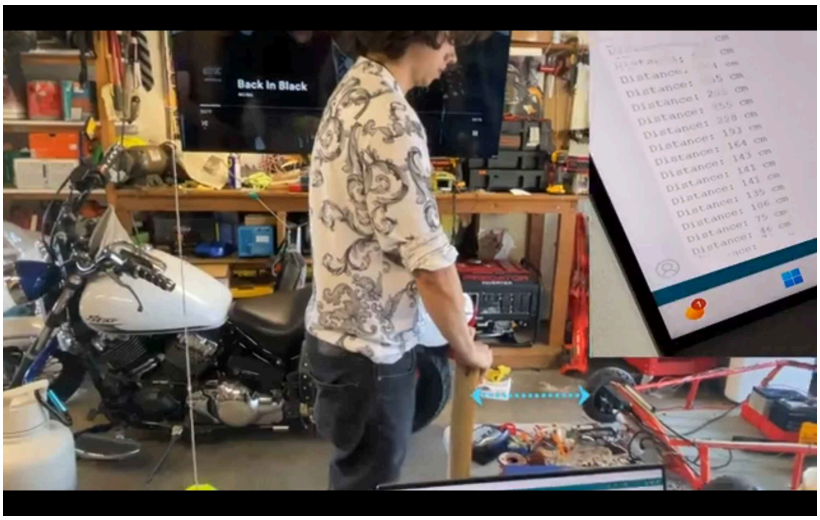


Figure 16: Lidar Measures Object Close

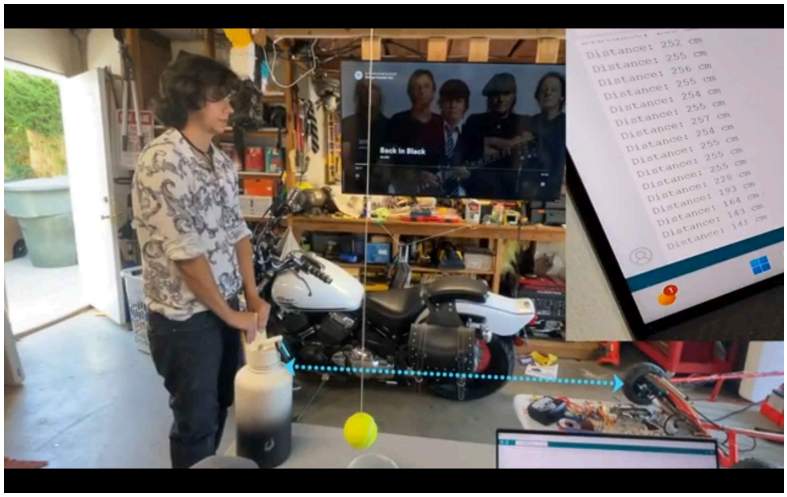


Figure 17: Lidar Measure Object Further

Distance (cm)
46
75
106
135
141
141
143
164
193
228
255
255
255
254

Table 1: Lidar System Measurements

Figures 16 and 17, show a member of the team slowly backing up from the lidar to test its accuracy and effectiveness. Table 1 above is a compilation of the recorded measurements. As the member gets further and further away from the lidar unit, the distance increases and is registered by the lidar system. It receives a very quick update from the system allowing for even gradual or finer measurements making the overall system safer.

```arduino
// Function to read distance from TF-Luna sensor via UART (non-blocking)
void readDistance() {
  while (tflSerial.available()) {  // Check if data is available
    uint8_t byte = tflSerial.read();  // Read a byte from TF-Luna

    // Look for packet header (two consecutive 0x59 bytes)
    if (recvIndex < 2 && byte == 0x59) {
      recvBuffer[recvIndex++] = byte;  // Store header bytes
    } else if (recvIndex >= 2) {
      recvBuffer[recvIndex++] = byte;  // Store other packet bytes

      // If a full packet (9 bytes) is received
      if (recvIndex == 9) {
        uint16_t checksum = 0;

        // Calculate checksum from the first 8 bytes
        for (int i = 0; i < 8; i++) {
          checksum += recvBuffer[i];
        }

        // Validate checksum and extract distance
        if ((checksum & 0xFF) == recvBuffer[8]) {
          distance = recvBuffer[2] | (recvBuffer[3] << 8);  // Combine low and high bytes
          // Clamp distance within valid range
          distance = (distance > MAX_DISTANCE) ? MAX_DISTANCE : distance;

          // Print the measured distance to the serial monitor
          Serial.println(String(distance) + " cm");
        }

        // Reset index to start reading the next packet
        recvIndex = 0;
      }
    } else {
      // Reset index if the header bytes are incorrect
      recvIndex = 0;
    }
  }
}
```

Figure 18: Arduino Lidar Code

```raspberry-pi
// Reads distance data from a Lidar sensor and manages cruise control activation
int run_lidar(int toggle, int piHandle, int lidarHandle, int lidarAddr) {
    // --- Initialize Variables for Lidar Data ---
    uint16_t distance;    // Variable to hold the calculated Lidar distance
        char data[2];          // Buffer to store the 2 bytes of raw distance data

    // --- Read Data from Lidar Sensor ---
    int bytesRead = i2c_read_device(piHandle, lidarHandle, data, 2);
    if (bytesRead == 2) {
        // Parse the distance from the 2-byte data buffer
        distance = (data[1] << 8) | data[0]; // Combine the two bytes into a 16-bit distance value
        cout << "Lidar Distance: " << distance << " cm" << std::endl;

        // If cruise control is not active, return -1 (no distance is used)
        if (!cruise_activated) {return -1;}
    }
    else {
        // --- Handle Lidar Read Errors ---
        // Log an error message and attempt to reinitialize the Lidar I²C handle
        cout << "Lidar failed. Reinitializing I2C handle...\n";
        i2c_close(piHandle, lidarHandle); // Close the existing Lidar handle
        lidarHandle = i2c_open(piHandle, 1, lidarAddr, 0); // Reopen the I²C handle for the Lidar
        return -1; // Return -1 to indicate an error occurred
    }

    return distance;  // Return the calculated Lidar distance (in cm)
}
```

Figure 19: Raspberry Pi Data Receive Code

```raspberry-pi
// --- CRUISE CONTROL ---
lidarDist = run_lidar(buttons2, pi, lidarHdl, lidarAddr);
if (cruise_active()) {
    // Adjust acceleration and braking based on Lidar distance
    controlledAccelValue = calculate_cruise(lidarDist);
    controlledBrakeValue = calculate_cruise_breaks(lidarDist);
} else {
    // Use manual control inputs for acceleration and braking
    controlledAccelValue = accelValue;
    controlledBrakeValue = brakeValue;
}
// --- AUTOPILOT CONTROL ---
gps_destination_counter += run_autopilot(buttons2, run_GPS(pi, GPSHdl, GPSAddr),
destination_data[gps_destination_counter]);

if (autopilot_active())
{
    // Adjust steering based on GPS waypoints
    cout << "gps_destination_counter: " << gps_destination_counter << "\n"; // Current waypoint
    steerValue = calculate_autopilot_steering(run_GPS(pi, GPSHdl, GPSAddr),
destination_data[gps_destination_counter]);
}
```

Figure 20: Cruise Control Implementation

In Figures 18, 19, and 20, we see the code responsible for handling the lidar and its intended functions. In Figure 18 specifically, we see the Arduino subsystem where the entire purpose of this code is simply to measure the distance, and then send it along the I2C bus. It measures through the UART protocol as opposed to the I2C, then sends it along the bus. Originally, we set the lidar itself up on the I2C bus using the I2C protocol, but in doing so, we created major timing issues that ended up being the reason the I2C bus would lock up occasionally. By changing it to the UART protocol, we experienced a smooth measuring and transmission of data every time. In Figures 19 and 20, we focus our sights on the Raspberry Pi processor where the actual math of the cruise control comes into play. The distance is first received by the Pi and then transformed into a 2-bit integer that can be accessed by all of the functions. From there, a couple of equations handle the distance ensuring that the brakes and motor activate and deactivate based on the threshold set by the user.

GPS Implementation

The new approach moving forward will no longer have the traffic light managing the GPS. Instead of sending data to the traffic light, it will remain within the Raspberry Pi processor. By obtaining the latitude and longitude utilizing a GPS chip, we could have the variable remain on board, or even ship it to another Raspberry Pi unit. Either way, we would be able to contain it and use the coordinate for any functions we want including an autonomous mode. Our objective was to establish the foundation the next team could use for implementing such a level of autonomy. For this feature, we need to consider the different algorithms that can be effectively used in GPS implementation to compute various outcomes such as distance and direction, pathfinding, and more. An algorithm such as the Rapidly-exploring Random Tree (RRT) is an item we were considering to be able to randomly sample points in the environment and build a path by connecting these points, however, we decided against it due to the allocated time. The next team would be able to build off of our code and hopefully implement such an algorithm to allow for a level 5 autonomous vehicle. In our implementation, the vehicle would have a starting point and from that initial position, it would move towards the waypoints we set. We created specific coordinate points that assign the vehicle to use as a map for testing purposes.

Pinpoint Coordinate Measurements

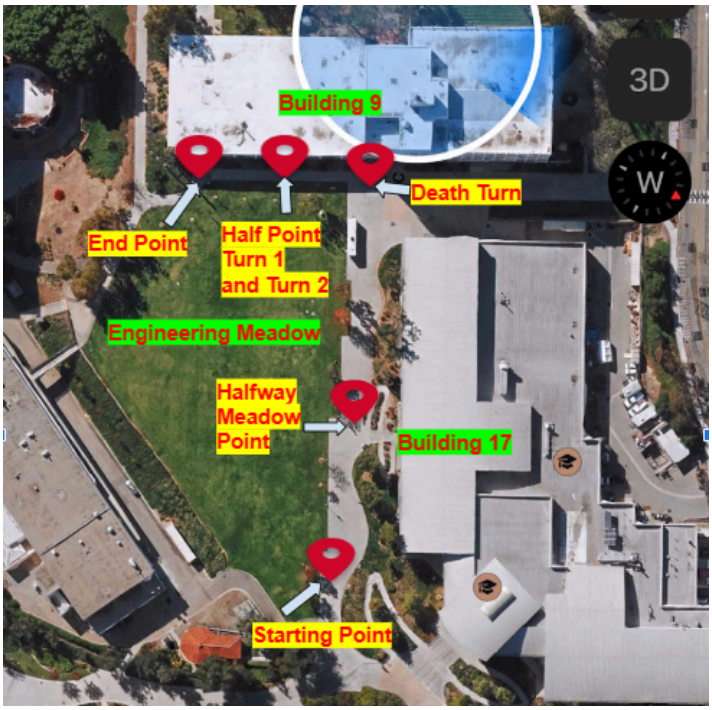


Figure 21: GPS Measurements Map

The map above in Figure 21 shows the different pinpoints we used to get coordinates for the vehicle to follow. This will eventually be the testing grounds for the fully autonomous mode.

Phone vs Vehicle GPS Data

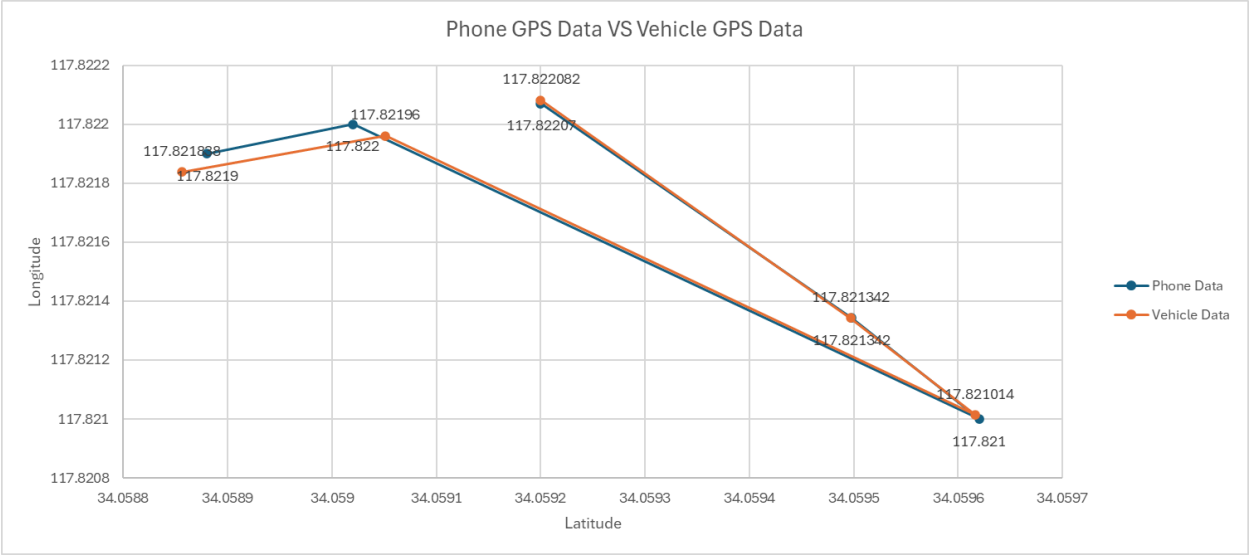


Figure 22: Comparison Between Phone and Vehicle GPS

The above plot shows a comparison between the phone and vehicle GPS accuracy. If the vehicle is accurate, it should closely match what the phone is providing as a coordinate which we can see is the case due to how close the points are to each other. In the tables below, we see another form of these tests where you can directly compare and contrast the two methods of GPS data collection.

Table 2: Vehicle GPS Data

Location	Points	Latitude	Longitude
Death Turn	1	34.0592	117.822082
Halfway Meadow Point	2	34.059497	117.821342
Starting Point	3	34.059616	117.821014
Half Point Turn 1 and Turn 2	4	34.059051	117.82196
End Point	5	34.058856	117.821838

Table 3: Phone GPS Data

Location	Points	Latitude	Longitude
Death Turn	1	34.0592	117.82207
Halfway Meadow Point	2	34.059498	117.821342
Starting Point	3	34.05962	117.821
Halfway Point Turn 1 and Turn 2	4	34.05902	117.822
End Point	5	34.05888	117.8219

Table 4: Percent Error (Phone as Master)

Latitude %	Longitude %
0	1.01848E-07
2.93604E-08	0
1.17441E-07	1.18824E-07
9.10185E-07	3.39495E-07
7.04662E-07	5.26218E-07

In table 4, we can see the percent error between the phone and module giving us insight into how much of an error in judgment our module has.

```
// Send GPS data via I2C when requested
void sendGPSData() {
  if (currentFix.valid.location) {
    float latitude = currentFix.latitude();
    float longitude = currentFix.longitude();
    int16_t heading_scaled = (int16_t)(currentFix.heading() * 100); // Scale heading for precision

    // Send data via I2C
    Wire.write((byte*)&latitude, sizeof(latitude));   // Latitude
    Wire.write((byte*)&longitude, sizeof(longitude)); // Longitude
    Wire.write((byte*)&heading_scaled, sizeof(heading_scaled)); // Heading
  }
}
```

Figure 23: Arduino GPS Code

```
// --- CRUISE CONTROL ---
lidarDist = run_lidar(buttons2, pi, lidarHdl, lidarAddr);
if (cruise_active()) {
    // Adjust acceleration and braking based on Lidar distance
    controlledAccelValue = calculate_cruise(lidarDist);
    controlledBrakeValue = calculate_cruise_breaks(lidarDist);
} else {
    // Use manual control inputs for acceleration and braking
    controlledAccelValue = accelValue;
    controlledBrakeValue = brakeValue;
}
// --- AUTOPILOT CONTROL ---
gps_destination_counter += run_autopilot(buttons2, run_GPS(pi, GPSHdl, GPSAddr),
destination_data[gps_destination_counter]);

if (autopilot_active())
{
    // Adjust steering based on GPS waypoints
    cout << "gps_destination_counter: " << gps_destination_counter << "\n"; // Current waypoint
    steerValue = calculate_autopilot_steering(run_GPS(pi, GPSHdl, GPSAddr),
destination_data[gps_destination_counter]);
}
```

Figure 24: General GPS Function

```
// Calculates a steering value (0-100) based on GPS data and destination coordinates
int calculate_autopilot_steering(double* gps_data, double* destination_data) {
    // Extract GPS data
    double current_lat = gps_data[0];            // Current latitude
    double current_lon = gps_data[1];            // Current longitude
    double current_course_angle = gps_data[2];       // Current course angle from GPS in degrees

    // Extract destination data
    double destination_lat = destination_data[0];    // Destination latitude
    double destination_lon = destination_data[1];    // Destination longitude

    // Convert latitude and longitude from degrees to radians for trigonometric calculations
    double current_lat_rad = current_lat * PI / 180.0;
    double current_lon_rad = current_lon * PI / 180.0;
    double destination_lat_rad = destination_lat * PI / 180.0;
    double destination_lon_rad = destination_lon * PI / 180.0;

    // Calculate the difference in longitude between the current position and destination
    double delta_lon = destination_lon_rad - current_lon_rad;

    // Calculate the desired bearing angle (in radians) to the destination
    // This uses the haversine formula but may not be accurate
    double y = sin(delta_lon) * cos(destination_lat_rad);
    double x = cos(current_lat_rad) * sin(destination_lat_rad) - sin(current_lat_rad) * cos(destination_lat_rad) * cos(delta_lon);
    double desired_bearing_rad = atan2(y, x);

    // Convert bearing from radians to degrees
    double desired_bearing = fmod((desired_bearing_rad * 180.0 / PI) + 360.0, 360.0);

    // Convert the desired bearing from radians to degrees and normalize it to [0, 360]
    double bearing_diff = desired_bearing - current_course_angle;

    // Normalize the bearing difference to the range [-180, 180]
    if (bearing_diff > 180) {
        bearing_diff -= 360;
    } else if (bearing_diff < -180) {
        bearing_diff += 360;
    }

    // Use map function to get a smoother steering value
    // Map bearing_diff from [-180, 180] to [0, 100] for smoother control
    int steerValue = map(static_cast<long>(bearing_diff), -180, 180, 0, 100);

    // Debugging Information: Outputs the current GPS data, destination, and calculated values
    printf("GPS INFO:    Latitude: %.8f, Longitude: %.8f, Course: %.2f\n", current_lat, current_lon, current_course_angle);
    printf("DESTINATION: Latitude: %.8f, Longitude: %.8f\n", destination_lat, destination_lon);

    cout << "Desired Bearing: " << desired_bearing << "\n";
    cout << "Bearing Difference: " << bearing_diff << "\n";
    cout << "Mapped Steering Value: " << steerValue << "\n";

    // Return the calculated steering value
    return steerValue;
}
```

Figure 25: Autopilot Function

In Figures 23, 24, and 25, we see the code responsible for the GPS implementation. In Figure 23, similar to the lidar system, the Arduino is responsible for collecting the GPS data and then sending it to the processor upon request. In Figure 24, we see the Pi receive said data, and in Figure 25, we see that data is used to calculate where the car is, where the car is going, and where the car needs to go. The biggest issue with this implementation currently is the calculations rely heavily on the car's current course angle. Without it, the vehicle has no bearing on where it is at, nor where it's going, thus making it so it doesn't make it to the waypoints. This is currently where this implementation stops and simply needs some more time debugging the course angle. Otherwise, the math works as verified by MATLAB in a simulation where this exact formula was put to the test.

Network Latency between AT&T and T-Mobile

In this section, we dive into the latency between different phone providers that have been used in this project. The previous team used AT&T whereas this team used T-Mobile. Both services work great, this section is more so just as a reference for what's happening in the server while you are operating the vehicle.

Table 5: Network Latency between RSU and RPV

Time (s)	AT&T (ms)	T-Mobile (ms)
0	153	155
1	152	118
2	150	146
3	171	147
4	193	148
5	126	146
6	175	122
7	153	164
8	176	131
9	165	146
10	174	193
11	146	173
12	135	340
13	147	296
14	157	167

The average of T-Mobile's latency is 172.8 ms and the average for AT&T is 158.2 ms.

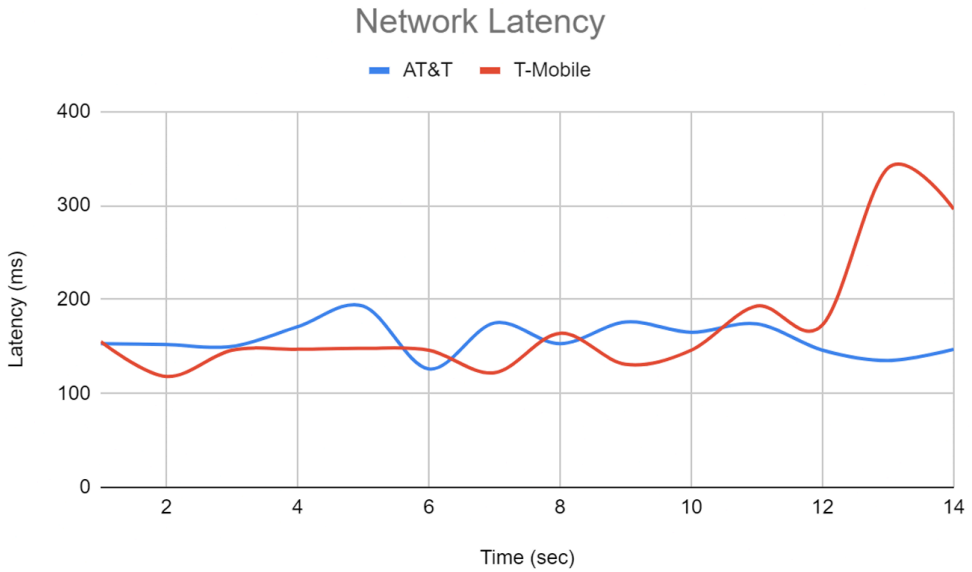


Figure 26: Network Latency between RSU and RPV

Table 6: Network Latency between RSU and Amazon Web Services (AWS)

Time (s)	AT&T (ms)	T-Mobile (ms)
0	244	77.3
1	279	66.8
2	64	73.2
3	71.7	63.7
4	71.5	80.7
5	60.6	83.3
6	61.2	54.8
7	65	55.1
8	63.7	63.2
9	64.5	95.9
10	70.3	71.4
11	60	79.6
12	66.9	97.4
13	97.4	124
14	64.1	68.1

T-Mobile averaged a latency of 76.96 ms and AT&T averaged 93.59 ms.

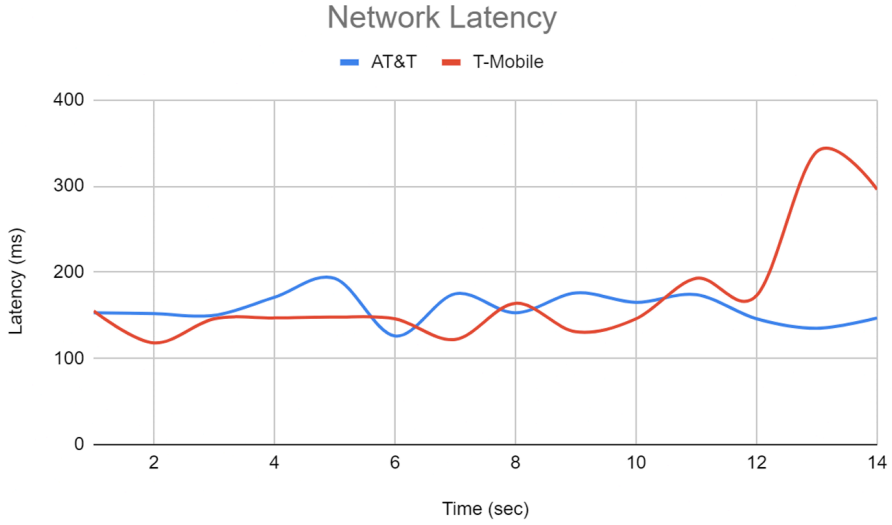


Figure 27: Network Latency between RSU and AWS

The above tables and graphs are documented from the previous group by testing the latency of the vehicle concerning the AT&T and T-Mobile network. The latency is important because it shows how responsive the vehicle is. Latency can affect the system's ability to respond effectively to changes in the environment or input signals. If the delay is too long, it can destabilize the system, leading to poor performance.

Depreciated Code And Hardware

Items in this section still exist either on or in the vehicle, but have since been depreciated for various reasons. Most of these parts are either undocumented, missing, or broken to an immeasurable degree.

```
// Controls the status LED for computer vision (CV) mode based on its activation state
void run_cv_status_led(int status, int piHandle, int lightingHandle) {
    // Static variables to maintain state across function calls
    static bool flag1 = 1; // Debounce flag to prevent redundant ON commands
    static bool flag2 = 1; // Debounce flag to prevent redundant OFF commands

    // --- Handle CV Status LED ---
    if (status == 1) { // CV mode is enabled
        flag2 = 1; // Reset the OFF debounce flag
        if (flag1) { // Only send the ON command if the debounce flag is set
            i2c_write_byte(piHandle, lightingHandle, 8); // Send command to turn the status LED ON
            flag1 = 0; // Clear the ON debounce flag to prevent repeated commands
        }
    } else if (status == 0) { // CV mode is disabled
        flag1 = 1; // Reset the ON debounce flag
        if (flag2) { // Only send the OFF command if the debounce flag is set
            i2c_write_byte(piHandle, lightingHandle, 9); // Send command to turn the status LED OFF
            flag2 = 0; // Clear the OFF debounce flag to prevent repeated commands
        }
    } else { // Default state for safety or invalid status values
        // Reset both debounce flags to their initial state
        flag1 = 1;
        flag2 = 1;
    }
}
```

Figure 28: CV LED Flag

In Figure 28, this snippet of code is for an LED located at the front of the car that used to turn on to signify the mini PC was connected and running its program. This entire system has been taken off the vehicle and since depreciated.

```
// DEPRECIATED
int run_active_safety(int piHandle, int ultrasonicHandle) {
    static char usSensors[2];
    int threshold = 35;
    i2c_read_device(piHandle, ultrasonicHandle, usSensors, 2);
    /*for (int i = 0; i < 4; i++)
        printf("Sensor %d: %d\n", i, usSensors[i]);*/
    printf("Front: %dcm\n", usSensors[0]);
    printf("Back: %dcm\n", usSensors[1]);
    if (usSensors[0] < threshold || usSensors[1] < threshold)
        return 1;
    return 0;
    /*
    0 = normal
    1 = halt motors and brake.
    */
}
```

Figure 29: Active Safety System

Since the beginning of my team's time with this project, this section of code displayed in Figure 29 has been inoperable. This code was so problematic, that we had to completely remove it from the main loop as it would cause the entire car to react and lose control while simultaneously locking up the program, requiring you to rebuild the program in the IDE fully. The subsystem in charge of the data collection for this set of code was the ultrasonic subsystem, which consisted of two ultrasonic sensors placed on the front and back of the vehicle meant for close-range detection. The actual function is simple, when something is within the range of a sensor, the whole vehicle should stop. While there was nothing particularly broken with the code, we suspect that the addressing might be incorrect since it appears to be out of the range of the I2C bus. Usually, an I2C bus range is between 0x10 and 0x70 for a Raspberry Pi, which is the amount of addresses there are on a Pi. The first 10 addresses are occupied for default functions like clock functions and reset functions. The ultrasonics exist on address 6 which does not land within the range. It could be possible to introduce the ultrasonic system back into the data structure properly; all you need to do is change that address to fall within the range of the I2C bus.

```
    // --- Prompt for Active Safety Option ---
userInput2:
//-------------------------------------------------------------------------------------------
    // Depreciated section: Active Safety is no longer functional
    // Defaulting userOption to 2 (Active Safety OFF)
    userOption = 2;

    // cout << "Active Safety: 1 for ON, 0 for OFF: ";
    // cin >> userOption;
    switch (userOption) {
        case 2: { // Active Safety is disabled (default behavior)
            ASoption = 0;
            cout << "> Active Safety DEPRECIATED" << endl;
            break;
        }
        case 1: { // Enable Active Safety (if functionality is restored in the future)
            ASoption = 1;
            cout << "> Active Safety ENABLED" << endl;
            break;
        }
        case 0: { // Explicitly disable Active Safety
            ASoption = 0;
            cout << "> Active Safety DISABLED" << endl;
            break;
        }
        default: {
            cout << "Invalid input!" << endl;
            goto userInput2; // Retry prompt if input is invalid
        }
    }
```

Figure 30: Active Safety User Input

Figure 30 displays the user input that used to be requested when launching the program. This no longer comes up as a prompt but is here for documentation purposes. In the future, we might be able to bring this back if we want to enable safety precautions to a higher degree.

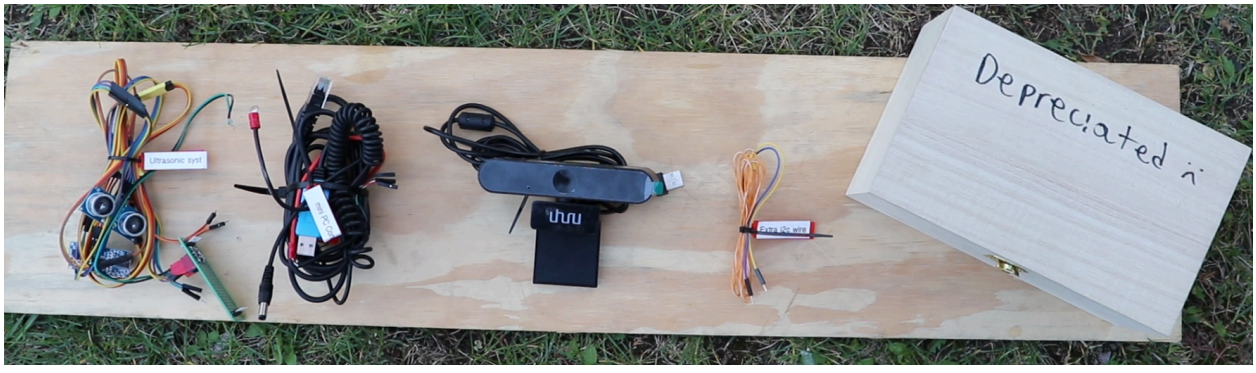


Figure 31: Depreciated Hardware

In Figure 31, we see a small spread of components taken out of the car. These are parts that have no use anymore or the code for them has been lost/depreciated. Each item exists in a box on the car for future use whenever a team decides they want to reimplement them, but for now, they cause too many issues or simply don't exist anymore. The system that caused the biggest

problem was the aforementioned ultrasonic subsystem, which the remnants can be seen in the left side of the photo. The actual Arduino unit still exists on the vehicle itself in the microcontroller array, however, it isn't wired up to ensure it doesn’t interfere with the I2C processing.

Figure 32: Mini PC

In Figure 32 we see an image of a mini PC. The previous team used to run their OpenCV code on this which was a protocol that enabled computer vision. This computer vision was geared to help detect lines on the ground for lane detection and used to do so at the Cal Poly Pomona track. Since my team has been on the project, this mini PC was taken by the team before mine, and with it went the code to run the lane detection software. Upon looking for the code in an attempt to get this system back online, it appears as if there is no documentation on this matter as the code for this open is currently lost. Due to these setbacks, we decided to remove this entire implementation to focus more on the GPS. This system has the potential to come back, as it would require a PC of similar specs, and may even be possible to utilize a Raspberry Pi to replace the PC in its entireity. This would give us the processing power required to perform computer vision for a reworked lane detection program.

References

[1] "Autonomous Cars," getelectricvehicle.com, March 2021, Available: https://getelectricvehicle.com/autonomous-cars/. Accessed: February 28, 2024.

[2] University of Michigan Center for Sustainable Systems, "Autonomous Vehicles Factsheet," University of Michigan Center for Sustainable Systems, Available: https://css.umich.edu/publications/factsheets/mobility/autonomous-vehicles-factsheet. Accessed: February 28, 2024.

[3] Ján Ondruš, Eduard Kolla, Peter Vertaľ, Željko Šarić, How Do Autonomous Cars Work?, Transportation Research Procedia, Volume 44, 2020, Pages 226-233, ISSN 2352-1465, Keywords: Autonomous cars; SAE level; technology; components; working; UFO (Accessed February 26, 2024)

[4] Daniel Howard, Danielle Dai, "Public Perceptions of Self-driving Cars: The Case of Berkeley, California" Master of City Planning, MS Transportation Engineering 2014 pp. 1 - 21, 2014 (Accessed February 28, 2024)

[5] J. Hanley, T. Bella Dinh-Zarr, A. Lund, D. McGehee"Forward Collision Warning," *My Car Does What*. https://mycardoeswhat.org/deeper-learning/forward-collision-warning/ (accessed February 26, 2024).

[6] L. Liu *et al*., "Computing Systems for Autonomous Driving: State of the Art and Challenges," in *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469-6486, 15 April 15, 2021, doi: 10.1109/JIOT.2020.3043716. keywords: {Autonomous vehicles;Sensors;Sensor systems;Security;Cameras;Real-time systems;Radar;Autonomous driving;challenges;computing systems}, (Accessed February 28, 2024)

[7] Forbes Technology Council, "Self-Driving Cars," Forbes, January 2024, Available: https://www.forbes.com/sites/technology/article/self-driving-cars/?sh=603a60d95e07. Accessed: February 28, 2024.

[8] K. Muhammad, A. Ullah, J. Lloret, J. D. Ser, and V. H. C. de Albuquerque, "Deep Learning for Safe Autonomous Driving: Current Challenges and Future Directions," in *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4316-4336, July 2021, doi: 10.1109/TITS.2020.3032227. keywords: {Roads;Task analysis;Safety;Automobiles;Accidents;Vehicles;Lane detection;Autonomous driving (AD);artificial intelligence;deep learning (DL);decision making;vehicular safety;vehicular technology;intelligent sensors}, (Accessed February 28, 2024)

[9] IIoT World. (n.d.). "Five Challenges in Designing a Fully Autonomous System for Driverless Cars." IIoT World. Available: https://www.iiot-world.com/artificial-intelligence-ml/artificial-intelligence/five-challenges-in-designing-a-fully-autonomous-system-for-driverless-cars/#:~:text=Five%20challenges%20of%20self%20driving%20cars%3A%201%201.,accidents%20liability.%20...%205%205.%20Radar%20Interference%20. Accessed: February 28, 2024.

[10] X. Hu, W. Lv, C. Xu, L. Wang, and H. Yan, "A Hybrid Deep Learning Approach to Urban Traffic Flow Prediction," in IET Intelligent Transport Systems, vol. X, no. X, pp. XXX-XXX, 2018. Available: https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2018.5351. Accessed: February 28, 2024.

[11] AutodriveAI. (n.d.). "Autonomous Mobility: A Comprehensive Overview of Hardware and Software Components." Medium. Available: https://medium.com/@autodriveai/autonomous-mobility-a-comprehensive-overview-of-hardware-and-software-components-d72ac6e22eee. Accessed: February 28, 2024.

[12] Bosch Company“Lane keeping assist,” *www.bosch-mobility.com*. https://www.bosch-mobility.com/en/solutions/assistance-systems/lane-keeping-assist/#:~:text=Lane%20keeping%20assist%20uses%20a (accessed February 26, 2024).

[13] Vaibhav, “How Does a 360 Degree View Car Camera Work?,” *Embitel*, Jun. 29, 2021. https://www.embitel.com/blog/embedded-blog/how-does-360-degree-view-car-camera-works#:~:text=Also%20referred%20to%20as%20a (accessed February 26, 2024).

[14]Synopsys “What is an Autonomous Car? – How Self-Driving Cars Work | Synopsys,” *www.synopsys.com*. https://www.synopsys.com/automotive/what-is-autonomous-car.html#:~:text=Autonomous%20cars%20rely%20on%20sensors (accessed February 26, 2024)

[15] J. Smoot “The Basics of Ultrasonic Sensors | CUI Devices,” *CUI Devices*, Apr. 2021. https://www.cuidevices.com/blog/the-basics-of-ultrasonic-sensors (accessed February 26, 2024).

[16] Z. Liu *et al*., "Robust Target Recognition and Tracking of Self-Driving Cars With Radar and Camera Information Fusion Under Severe Weather Conditions," in *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 6640-6653, July 2022, doi: 10.1109/TITS.2021.3059674. keywords: {Cameras;Radar;Autonomous vehicles;Data integration;Target recognition;Signal processing algorithms;Sensors;Multi-sensor fusion;radar camera fusion;severe weather conditions;self-driving cars}, (Accessed February 26, 2024)

[17] Generation Robots. (n.d.). "The Anatomy of an Autonomous Vehicle: A Simplified Overview." Generation Robots. Available:

https://www.generationrobots.com/blog/en/the-anatomy-of-an-autonomous-vehicle-a-simplified-overview/. Accessed: February 28, 2024.

[18] An Avnet Company, “Automatic emergency braking system: A solution for decreased human effort and improved safety,” *element14*, 2023. https://cn.element14.com/en-CN/automatic-emergency-braking-system-trc-ar#:~:text=Radar%20sensors%20emit%20radio%20waves (accessed February 26, 2024).

[19] Henry Alexander Ignatious, Hesham-El- Sayed, Manzoor Khan, An overview of sensors in Autonomous Vehicles, Procedia Computer Science, Volume 198, 2022, Pages 736-741, ISSN 1877-0509, Keywords: Autonomous Vehicles (AVs); Autonomous Driving (AD); sensors; LiDAR; RADAR; Intelligent Transportation System (ITS) (Accessed February 28, 2024)

[20] Peter Leiss, "Autonomous Vehicles Sensors Expert," The Experts Robson Forensic, September 2018, Available: https://www.robsonforensic.com/articles/autonomous-vehicles-sensors-expert. Accessed: February 28, 2024.

[21] Tesla “Model Y Owner’s Manual,” *Tesla*. https://www.tesla.com/ownersmanual/modely/en_eu/GUID-A701F7DC-875C-4491-BC84-605A77EA152C.html (accessed February 26, 2024).

[22] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B. Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M. Paixão, Filipe Mutz, Lucas de Paula Veronese, Thiago Oliveira-Santos, Alberto F. De Souza, Self-driving cars: A survey, Expert Systems with Applications, Volume 165, 2021, 113816, ISSN 0957-4174, Keywords: Self-driving cars; Robot localization; Occupancy grid mapping; Road mapping; Moving objects detection; Moving objects tracking; Traffic signalization detection; Traffic signalization recognition; Route planning; Behavior selection; Motion planning; Obstacle avoidance; Robot control (Accessed February 26, 2024)

[23] A. Chowdhury, G. Karmakar, J. Kamruzzaman, A. Jolfaei and R. Das, "Attacks on Self-Driving Cars and Their Countermeasures: A Survey," in *IEEE Access*, vol. 8, pp. 207308-207342, 2020, doi: 10.1109/ACCESS.2020.3037705. keywords: {Autonomous automobiles;Automobiles;Security;Roads;Vehicular ad hoc networks;Autonomous vehicles;Safety;Self-driving cars;intelligent transportation system;security attacks;mitigation strategies;cybersecurity;VANET}, (Accessed February 26, 2024)

[24] Alliance for Automotive Innovation “HAVs | Highly Automated Vehicles | Alliance For Automotive Innovation,” *www.autosinnovate.org*, 2023. https://www.autosinnovate.org/initiatives/innovation/autonomous-vehicles/benefits-of-havs (accessed February 26, 2024)