The Transformation of Vehicle Controls to Drive-By-Wire

A Technical Report submitted to the Department of Mechanical and Aerospace Engineering

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Alexander Matthew Pascocello

Spring, 2022

Technical Project Team Members

Jacob Deane

Matthew Deaton

Henry Goodman

Logan Montgomery

Vishal Singh

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Advisor

Tomonari Furukawa, Department of Mechanical and Aerospace Engineering

TABLE OF CONTENTS

1. Introduction
2. Background and Essential Knowledge
3. Design Process
4. Final Design
    - 4.1. Full Design Diagrams
    - 4.2. Throttle-by-Wire System
    - 4.3. Brake-by-Wire System
    - 4.4. Steer-by-Wire System
    - 4.5. Emergency Stop and Autonomation Switch
    - 4.6. Sensor Suite
    - 4.7. Software
5. Mathematical and Numerical Analyses
    - 5.1. Motion Model Comparison
    - 5.2. PID Tuning
6. Experimental Validation
7. Operations Manual
8. Conclusions and Future Work

1 INTRODUCTION

Autonomous vehicles are revolutionizing the transportation industry. If implemented correctly they can provide greater freedoms to the eldery and disabled, create safer roads, and give back time to commuting workers. While this technology is not ready to be assimilated into society, there are steps being taken to introduce semi-autonomous vehicles to the road. There is a scale of autonomy for such vehicles that ranges from zero, no autonomy, to five, full autonomy, as seen in Figure 1. Currently, some electric vehicles already use low levels of automation such as self-parking and lane assist (Shipley, 2021). This would not be possible without the ability to control the vehicle's integral driving functions such as steering, throttle, and braking. The goal of this project is to lay the groundwork for the future autonomous control of a Ford Escape by creating an electromechanical drive-by-wire system to externally steer the vehicle. This will allow the implementation of semi-autonomous processes and eventually full autonomy.

Fig. 1. Levels of Autonomy as Defined by The Society of Automotive Engineers (Wevolver, 2020)

Early on in the design process, the team established a set of target specifications that reflected the needs of the prospective clients. This target specification list can be seen below in Figure 2. In the past, there have been multiple attempts to create a higher level autonomous vehicle with some of these design elements, with varied levels of success. The Stanford Stanley project is a modified version of a Volkswagen Touareg R5, designed for competing in DARPA Grand Challenge in 2005. Stanford's Stanley placed first in this competition, and boasted a completion time of 6 hours and 54 minutes (see Appendix F). While this vehicle included important components of the target specifications mentioned above such as functional steering, throttle, and braking systems, it lacks important systems, such as the integration of a remote control, allowing the car to be manually controlled from an outside user.

Outside of vehicles designed for the DARPA Grand Challenge, there are multiple companies currently developing level 4 autonomous vehicles. Waymo, formerly the Google Self-Driving Car Project, has developed a fleet of functioning autonomous vehicles, operating in Phoenix, Arizona (Schwall et al., 2020). While these vehicles are impressive and present groundbreaking breakthroughs in autonomous driving technology, they operate on a slightly different principle than what has been defined for our project by the initial target specifications. Waymo vehicles operate off of a highly detailed map that is premade for a specific city in conjunction with the sensor suite integrated into the vehicle (Laukkonen, 2020). For the purposes of this project, the ultimate goal is to create a vehicle that does not need to operate in a specific city, or be limited to use only on well maintained and marked roads. There is definitely much to take away from the way Waymo's autonomous vehicles operate, but the end goal of this project differs categorically from Waymo's Self-Driving Car.

The main objectives of this project are to create a vehicle that is able to map its surroundings in real time and make decisions regarding what actions to take in order to safely drive to its objective. This process should not be restricted to operating solely on paved city streets, and as a result cannot be solely reliant on external mapping and road markings. A full list of these goals can be seen below in Figure 2, and potential design solutions based around the completion of these goals can be seen in Figure 3. These goals are not expected to be completed in their entirety by the end of the year, and are more reflective of the long term, multi-year goals of the project. For this year, the main objectives of the project are to design and integrate functioning steer-by-wire, throttle-by-wire, and brake-by-wire systems into the car, as well as create a design for the essential sensors for progress in future years.

2 BACKGROUND AND ESSENTIAL KNOWLEDGE

The Ford Escape being used for the team's project came from a previous Virginia Tech project with similar goals. When the car was received it was stripped of most subsystem connections including all LiDar and radar systems. There was a mounted pulley with a rusted stepper motor already installed for the brake-by-wire system, but it had to be replaced due to rust. The computer's hard drive became corrupted and had to be replaced, but some past Arduino and Python code was salvageable.

Beside the state of the car, no report or documentation is known regarding the success of the previous team. While the current TA involved with the project spearheaded the previous project at Virginia Tech, his involvement is purely educational, helping guide the team with his previous knowledge of the systems.

There were several required subjects the team researched in preparation for the drive-by-wire project. These topics included electrical systems of electric power steering and throttle system components, closed-loop control systems specializing in PID controllers, and Controller Area Network (CAN) bus readings utilizing python codes. The technological background aided the team in accomplishing the set forth goals and creating functional steer-by-wire, brake-by-wire, and throttle-by-wire subsystems.

Advancements were made on previous drive-by-wire projects in several different ways in accordance with customer requirements. Firstly, the car will have an emergency stop feature that will immediately halt all autonomous capabilities and give control over to the driver. Furthermore, there is an accessible switch to easily switch from driver to autonomous mode in a non-emergency setting. A second advancement to the drive-by-wire vehicle is a PID feedback closed-loop control system for each subsystem to ensure a smooth and safe controlled driving experience. This control system will be used, for example, to confirm steering angles while driving at high speeds where the steering wheel could naturally return to a neutral position due to tire friction. In these ways the team's project improves on other drive-by-wire systems.

3 DESIGN PROCESS

To determine the goals of the project and design, we first surveyed a customer on what they would like to see in their drive-by-wire vehicle that would eventually become autonomous in the future. For the scope of this year's project goal, a strictly drive-by-wire car with no autonomy, the most important needs from the customer were a "drive-by-wire system with practical controller," and it "must have steering and override electrical or mechanical

capabilities," referring to the E-stop needed for the car. The biggest customer need was that the controller was "safer than a human operator," which was emphasized in our target specifications.

Taking into account the relation to customer needs, the technical importance, and the difficulty to implement, target specifications were generated based on important safety and performance criteria for a drive-by-wire system and the car as a whole. The full Quality Function Deployment (QFD) chart is shown in the appendix, but a short summary is shown in Figure 2 below.

Turning resolution	0 - 60mph acceleration time	Response time for Braking	Response time for throttle	Obstacle avoidance distance	Response time for steering	Total braking distance from 60- 0mph	Throttle (speed changing) resolution
↓	↓	↓	↓	○	↓	○	↓
Marginal Target Value							
3 deg	10 s	0.5 s	0.5 s	5 ft	0.5 s	200 ft	3 mph
Technical Importance (without taking into account difficulty)							
10.4%	11.2%	16.4%	9.0%	9.6%	13.7%	16.4%	13.4%

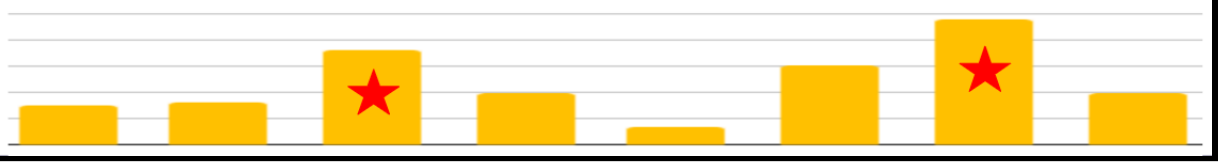


Fig. 2. Summary of the QFD Chart

The two most prioritized specifications are the total braking distance from 60-0 mph and the response time for braking. The target specifications for response time for braking and throttle were the focus of our testing, which will be explained in the mathematical and numerical analyses section.

To brainstorm and decide potential solutions to implement drive-by-wire in the braking, throttle, and steering, concept generation was used. The full concept generation, including the

functional decomposition, will be found in the appendix, but Figure 3 shows below a summary of the different solutions brainstormed and which solutions will be implemented.

Design Solutions															
Solutions	Braking					Throttle					Steering				
	Actuator	Power	Input Control	Microcontroller	Feedback Control Method	Method	Power	Input Control	Microcontroller	Feedback Control Method	Method	Power	Input Control	Microcontroller	Feedback Control Method
1	Stepper Motor	12V Car Battery	GUI on Onboard PC	Arduino Mega/Uno	CAN Bus Data	DigiPot Signal	12V Car Battery	GUI on Onboard PC	Arduino Mega/Uno	Pre-Existing Imbedded Potentiometer Suite	Torque Sensor Message to ECU	12V Car Battery	GUI on Onboard PC	Arduino Mega/Uno	CAN Bus Steering Position Data
2	Hack Power Braking	12V Car Battery	GUI on Onboard PC	TI Microcontroller	Absolute Encoder	Servo Motor to Push Pedal	12V Car Battery	GUI on Onboard PC	TI Microcontroller	Quadrature Encoder	Torque Sensor Message to ECU	12V Car Battery	GUI on Onboard PC	TI Microcontroller	CAN Bus Steering Position Data
3	Stepper Motor	12V Car Battery	GUI on Onboard PC	Parallax P2 Chip	Sensor Suite to Measure Speed	Pneumatic Actuator to Push Pedal	12V Car Battery	GUI on Onboard PC	Parallax P2 Chip	Pre-Existing Imbedded Potentiometer Suite	Motor in Line With Steering Column	12V Car Battery	GUI on Onboard PC	Parralax P2 Chip	CAN Bus Steering Position Data

Fig. 3. Concept Generation Design Solutions

Based on the morphological analysis, three potential solutions were created, and then these designs were evaluated based on their strengths and weaknesses. Out of three designs, solution number 1 was selected, as denoted by the yellow highlighted cells.

Our potential design was then compared to other similar designs by other institutions or existing solutions in the industry in a concept selection process. Figures 4 and 5 show the concept selection screening and scoring processes.

Potential Solution → / Selection Criteria ↓	CMU's Boss	Stanford's Stanley	VT's VictorTango	MIT's Talos	Volvo Drive Me	NuTonomy Driverless Taxi	Our sol. 1	Our sol. 2	Our sol. 3
Turning Resolution	+	+	-	+	0	+	+	+	+
0-60mph Acceleration	+	-	-	-	-	-	+	+	0
Response Time for Braking	0	-	-	-	-	-	0	-	-
Response Time for Throttle	+	+	+	+	+	+	+	+	+
Obstacle Avoidance Distance	0	-	-	-	-	-	0	-	-
Response Time for Steering	0	-	-	-	-	0	+	+	+
Total Braking Distance From 0-60mph	+	+	+	+	+	0	+	-	0
Throttle Resolution (Speed Change)	+	+	+	+	0	-	+	+	+
Sum +'s	5	4	3	4	2	2	6	5	4
Sum 0's	3	0	0	0	2	2	2	0	2
Sum -'s	0	4	5	4	4	4	0	3	2
Net Score	5	0	-2	0	-2	-2	6	2	2
Rank	2	5	7	5	7	7	1	3	3
Continue?	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes

Fig. 4. Concept Selection (Screening)

Inferences on how the different subsystems and target specifications of other solutions compared to our solutions were made based on performance, type of car, and the types of DARPA challenges they competed in. Our primary solution ranked the best in this qualitative analysis.

	Solution →		CMU's Boss		Stanford's Stanley		VT's VictorTango		MIT's Talos		Our sol. 1		Our sol. 2		Our sol. 3	
#	Selection Criteria ↓	Weight	Rating	Weight	Rating	Weight	Rating	Weight	Rating	Weight	Rating	Weight	Rating	Weight	Rating	Weight
1	Turning Resolution	10%	5	0.5	5	0.5	0	0	5	0.5	5	0.5	4	0.4	5	0.5
2	0-60mph Acceleration	10%	5	0.5	1	0.1	1	0.1	1	0.1	5	0.5	5	0.5	3	0.3
3	Response Time for Braking	20%	1	0.2	1	0.2	1	0.2	3	0.6	0	0	1	0.2	1	0.2
4	Response Time for Throttle	10%	5	0.5	4	0.4	5	0.5	4	0.4	5	0.5	5	0.5	5	0.5
5	Obstacle Avoidance Distance	10%	1	0.1	1	0.1	1	0.1	1	0.1	0	0	1	0.1	1	0.1
6	Response Time for Steering	10%	1	0.1	1	0.1	1	0.1	1	0.1	5	0.5	5	0.5	5	0.5
7	Total Braking Distance From 0-60mph	20%	4	0.8	4	0.8	4	0.8	4	0.8	5	1	2	0.4	3	0.6
8	Throttle Resolution (Speed Change)	10%	4	0.4	5	0.5	4	0.4	4	0.4	4	0.4	4	0.4	4	0.4
		100%		3.1		2.7		2.2		3		3.4		3		3.1
			RANK	4		2		1		3		5		1		4

Fig. 5. Concept Selection (Scoring)

In the concept scoring process, relative weights were applied to each selection criteria based on importance determined during the target specifications process. The rankings aligned with the concept screening rankings, with our solution 1 ranked the highest by receiving top scores in all categories besides response time for braking and obstacle avoidance distance.

4 FINAL DESIGN

4.1 FULL DESIGN DIAGRAMS

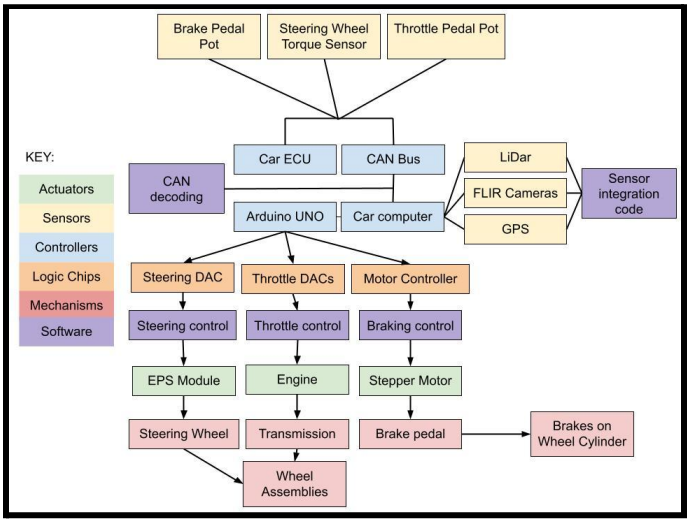


Fig. 6. Block System Diagram

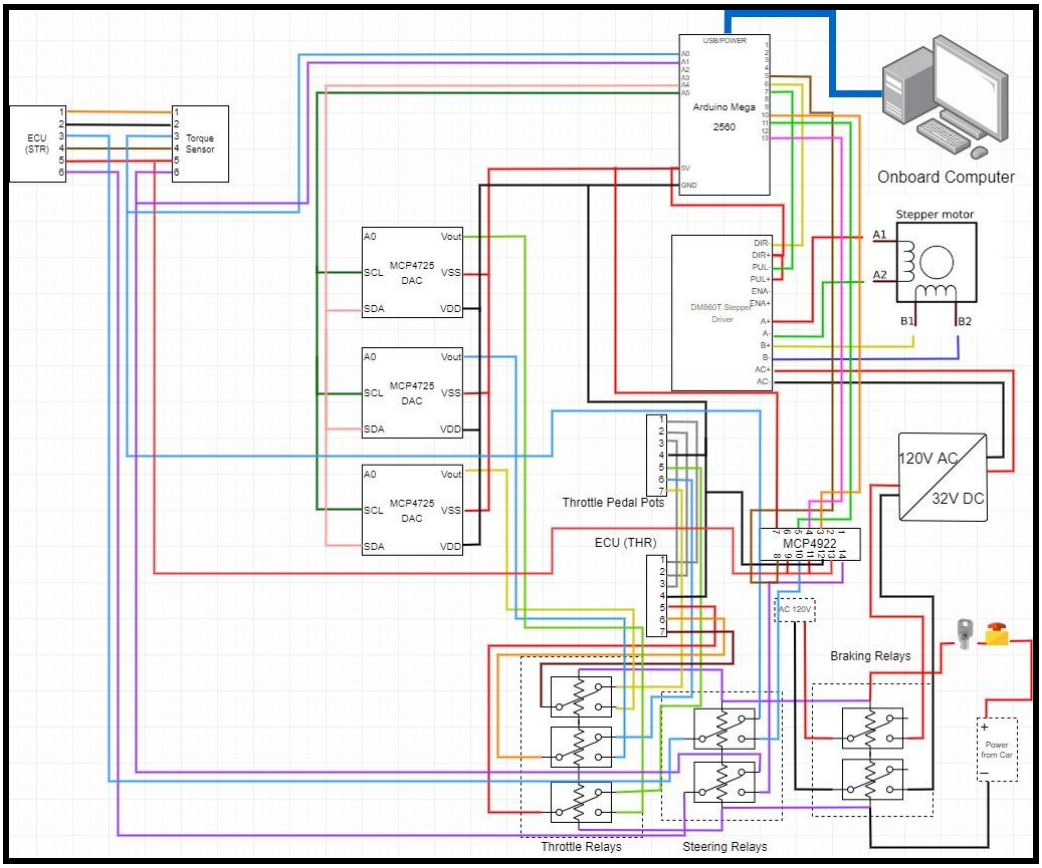


Fig. 7. Complete Wiring Diagram

4.2 THROTTLE-BY-WIRE SYSTEM

The design for the throttle control system is a direct electrical approach. Instead of using an actuator to pull the pedal (as is the case for braking) the team thought a less obtrusive and more direct option would be to mimic the signals that the throttle pedal sends the electrical control unit (ECU) to tell the car to tell the engine to accelerate. Therefore, the problem could be solved solely by using digital-to-analog converters (DACs). The throttle pedal has three potentiometers that send different ranges of voltages to the ECU to be interpreted as a throttle amount. These three signals need to be hijacked and mimicked. For this we use three DACs controlled by an Arduino UNO. The relevant electrical components are highlighted in Figure 8.

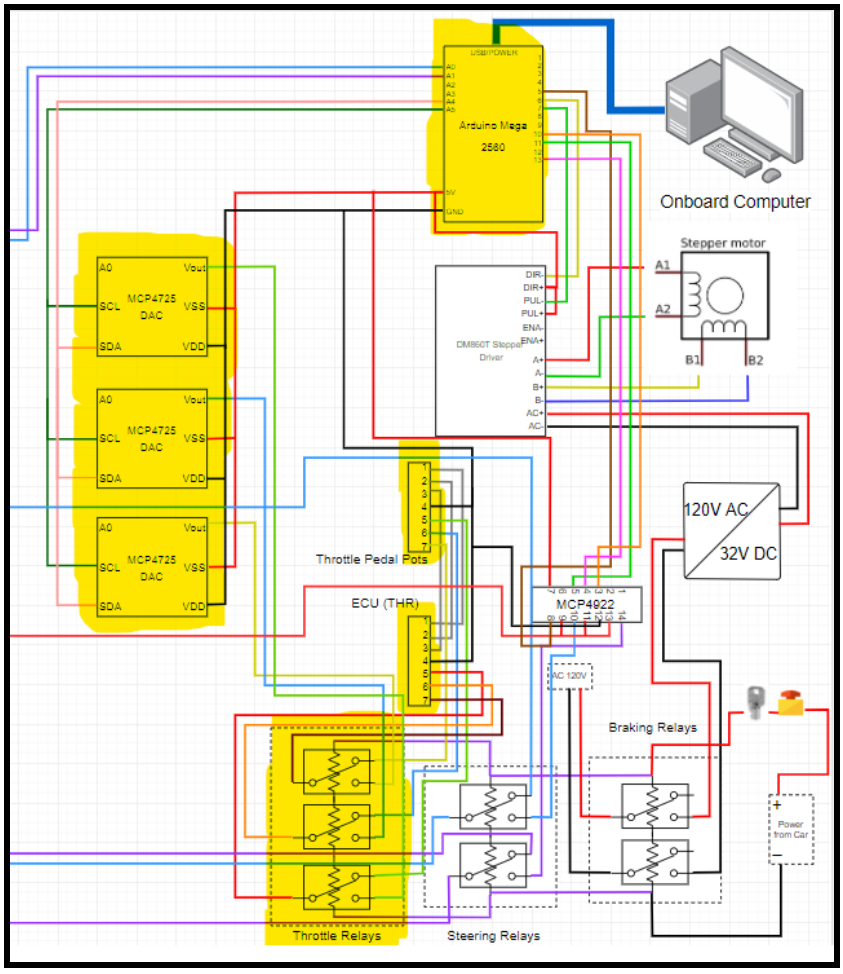


Fig. 8. Throttle Electrical Diagram

Our throttle code tells the DACs exactly what voltages to output in order to achieve a specific throttle percentage from 1-100. The three throttle signals each have their own unique idle voltages and voltage ranges that the ECU then uses to compute a throttle percentage. These are depicted in Figure 9.

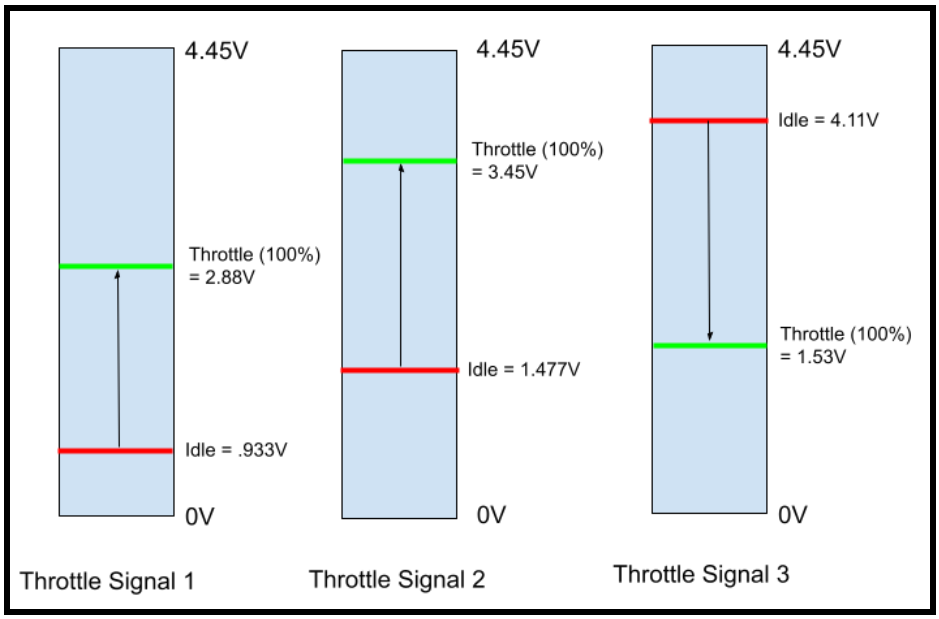


Fig. 9. Throttle signals sent to ECU for idle and throttle pedal fully pressed

These values were discovered experimentally by measuring the signals directly. Our code to control throttle takes these ranges and applies a linear distribution to the outputs from our DACs. So, the percent of throttle requires the three correct voltage values to be sent from each DAC's output channel.

```
void throttle(float thr_percent){
  float accelerator_command = 1800 * (thr_percent / 100);
  dac1.setVoltage(3730 - (1.3*accelerator_command), false);
  dac2.setVoltage(1320 + accelerator_command, false);
  dac3.setVoltage(815 + accelerator_command, false);
}
```

Fig. 10. Throttle Signal Mimic Function (in bits)

A three channel relay is able to switch the car between autonomous mode and manual mode by switching the voltages that the ECU receives from our DAC outputs to the throttle pedal outputs. In order to ensure accurate and safe acceleration, this system will eventually be integrated with and controlled by a PID controller. A feedback loop with the throttle messages

from the car's CAN bus will allow us to make sure that the car is autonomously throttling as expected even when changing slopes.

4.3 BRAKE-BY-WIRE SYSTEM

The braking subsystem is the only subsystem controlled by external actuation rather than internal signal mimicking. The team uses a stepper motor driven by a DM860T stepper driver and controlled by arduino code through ROS via our operator.py script. A power board inverts 120V AC to 32V DC in order to power the motor. The relays for braking switch the power on and off to the motor, so no external actuation is possible without them flipped. This system features a PID control loop using the brake's CAN bus status to ensure that the brake reaches the correct position in the most efficient and smooth manner with little oscillation. This is motivated by error factors such as the lack of precise positioning with the stepper motor and slack in the pulley cable.

The CAN bus feedback is also used in the zeroing function for braking. As mentioned, the stepper motor does not record position. This means that if the motor slips, the wire connecting it to the brakes may become taut and the brakes may become compromised. To ensure there is no slack in the pulley when the car is put into autonomous mode, the brake zeroing function spins the stepper motor until the CAN bus registers that the brake is pressed. Once this occurs, the function slowly steps back the motor until the CAN bus registers the brakes as released. This zeroing function is imperative to ensuring a taut cable and immediate braking. The relevant electrical components are highlighted in Figure 11 as well as CAD models of the pulley system.

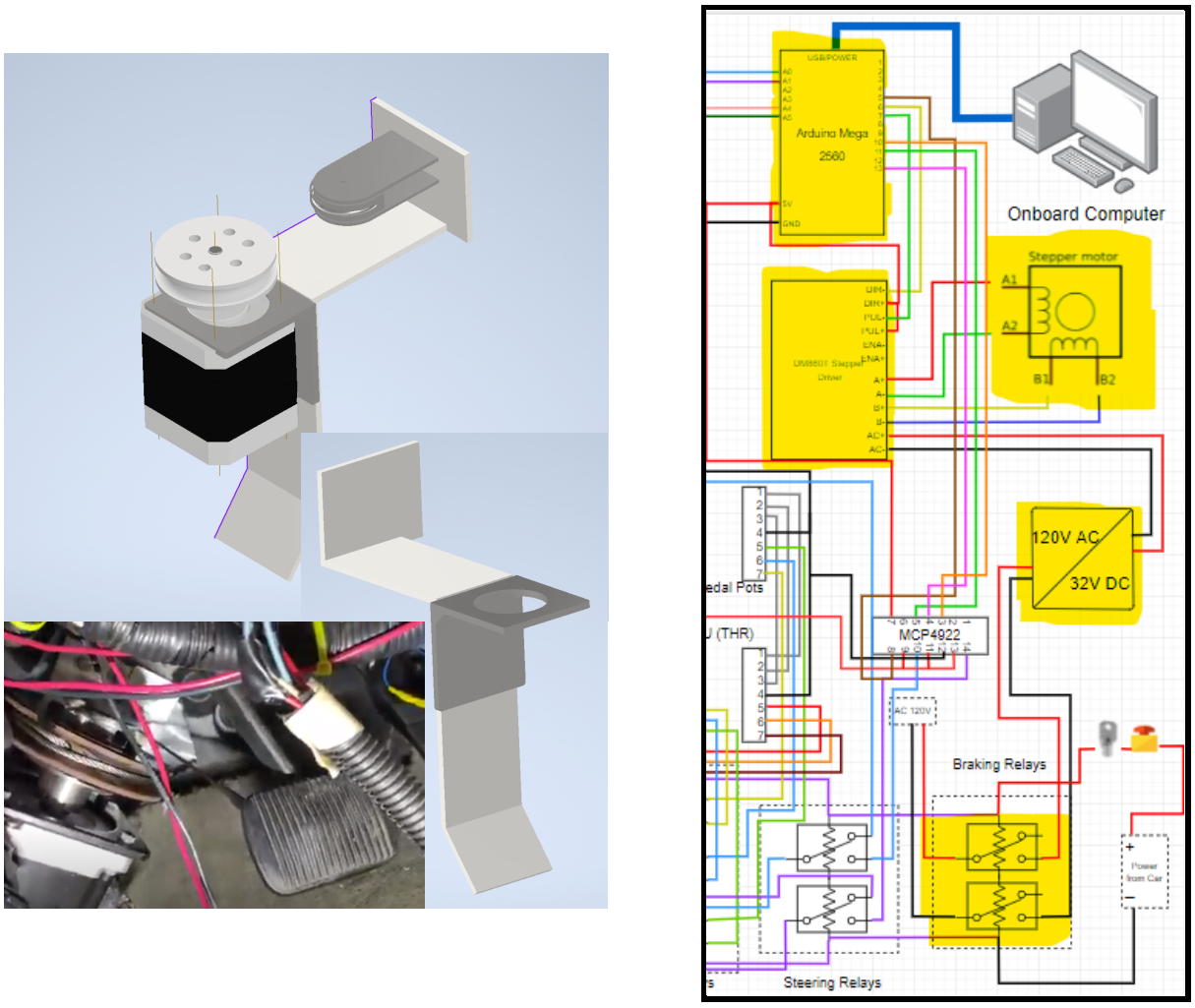


Fig. 11. Electrical (right) and physical (left) diagram of braking components

4.4 STEER-BY-WIRE SYSTEM

Electric power steering uses a motor to assist driver steering when a torque is applied. The torque sensor sends two voltages ranging from zero to five volts. These voltages stay around 2.5V when no torque is being applied and then spike or drop by similar amounts when torque is applied. Understanding and recording these values is required to emulate the signals sent by the torque sensor.

The steering control system's final design works quite similarly to the throttle control system. It uses a DAC to emulate the voltage signals for torque to the ECU, mimicking what the torque sensor in the steering column would give it to assist a steering motion from the user. The steering system uses a single dual channel DAC called the MCP4922 which can output two different voltages. These signals are sent to the ECU when the car is in autonomous mode and the relevant electrical components are highlighted in Figure 12 below.

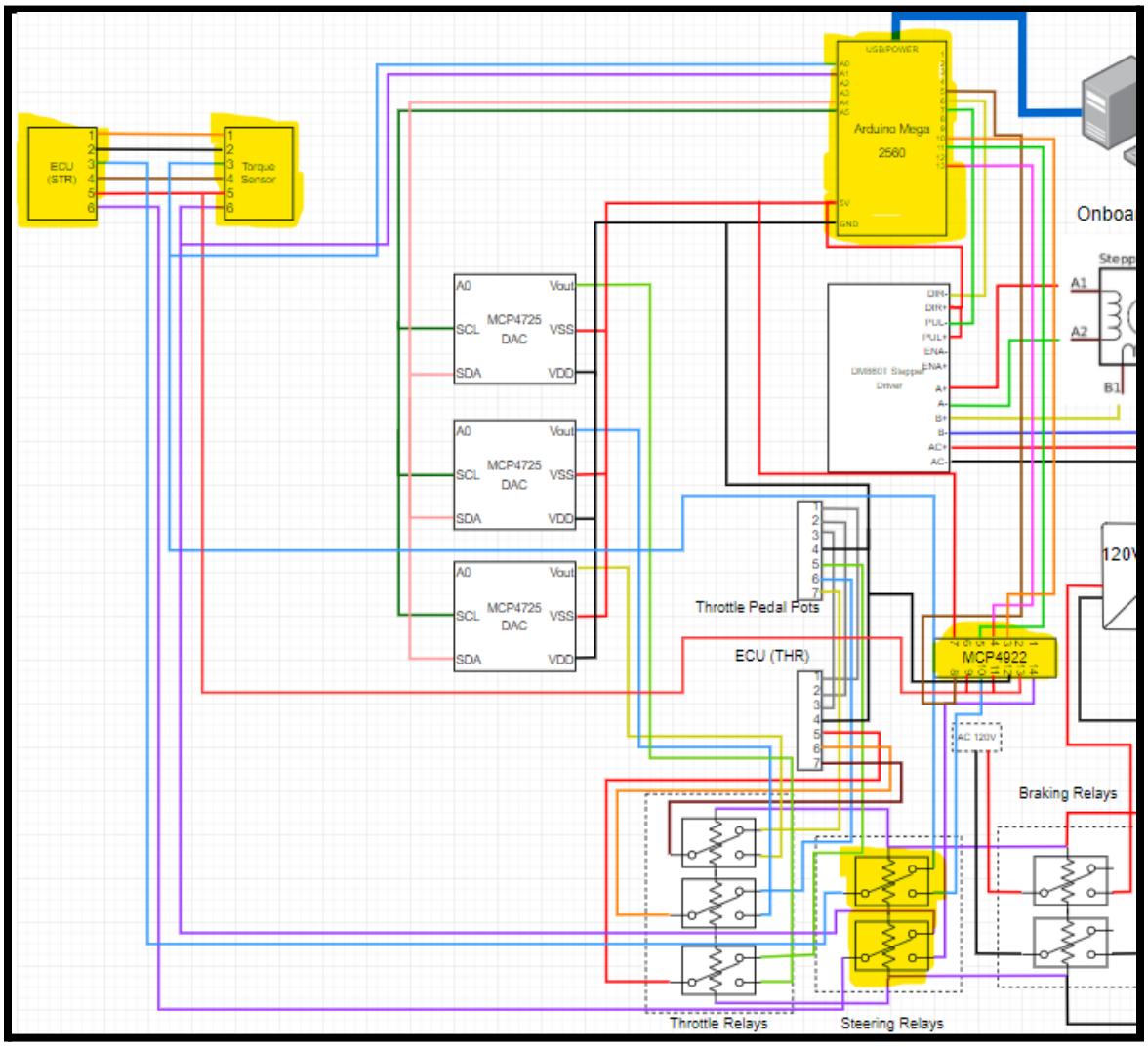


Fig. 12. Electrical diagram (steering components)

The steering signals that are being recreated are torque signals, so they spike when torque is applied and then return to a zero torque value. There are two torque signals to be mimicked, as

opposed to throttle where there are three position signals. Figure 13 explains how the torque signals move when torque is applied.

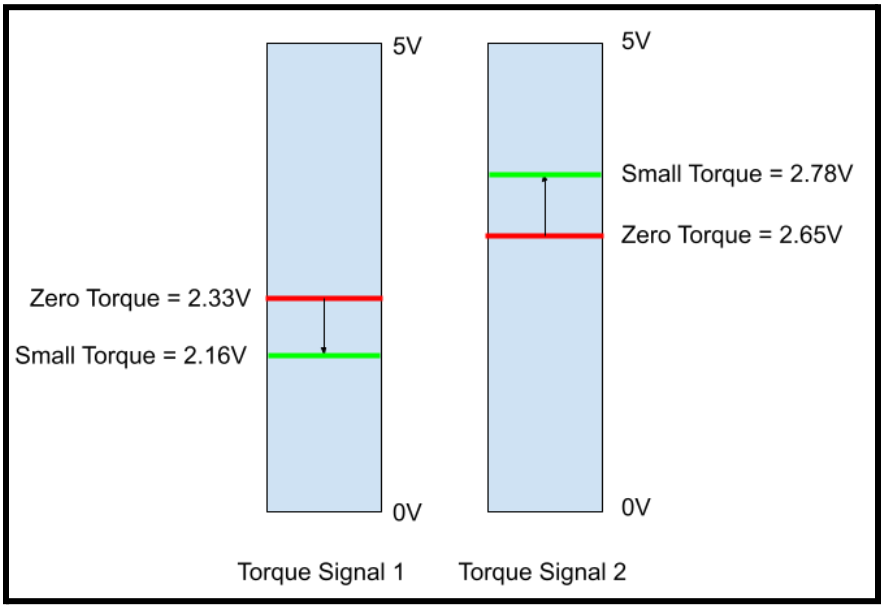


Fig. 13. Steering signals sent to ECU for zero torque and a small experimental torque

Understanding how these messages work was required to write the subsequent control code. This code takes a single input of a negative or positive bit value to offset each signal by. This is the input that we will then use to map the left stick of the remote controller. Since this is giving a torque command, PID control is necessary to give the steering wheel positional control, so when the user lets go of the stick the steering wheel will return to its straight position.

```
void turn(int C){
if (C >= 0){
  torque(1900 + C, 1700 - C);
//  delay(100);
//  torque(1900,1700);
//  delay(200000000);
}
if (C < 0){
  torque(1900 + (4.5*C), 1700 - (4.5*C));
//  delay(100);
//  torque(1900,1700);
//  delay(200000000);
}
}
```

Fig. 14. Steering Arduino Function

4.5 EMERGENCY STOP AND AUTONOMATION SWITCH

The car will have the ability to switch between autonomy and manual control via two methods of input on the center console: the emergency stop (E-stop) button and the autonomy switch. These are connected in series with the relay circuit, so that users will need to actuate both inputs to enter autonomous mode, but just one to disable it. All signals being sent to the ECU and the power for the stepper motor run through relays. Flipping these relays is the most direct way to return the car to manual operation, giving the user control over all subsystems. A future improved version of the emergency system might entail an immediate full-braking command or a pull over and stop sequence. These are some semi-autonomous processes to be further developed by future teams. A picture and diagram of the E-stop and autonomy switch are provided in Figures 15 and 16.

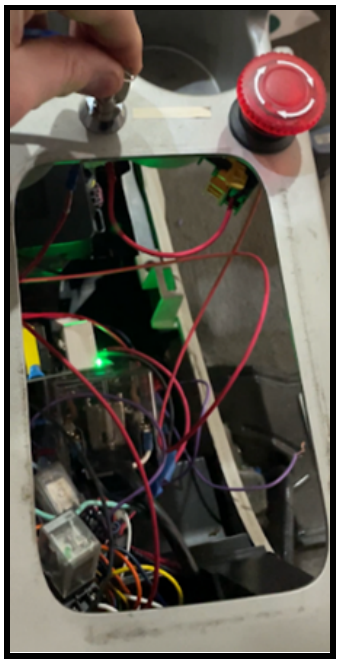


Fig. 15. Emergency stop inputs: autonomy switch (left), E-stop button (right)

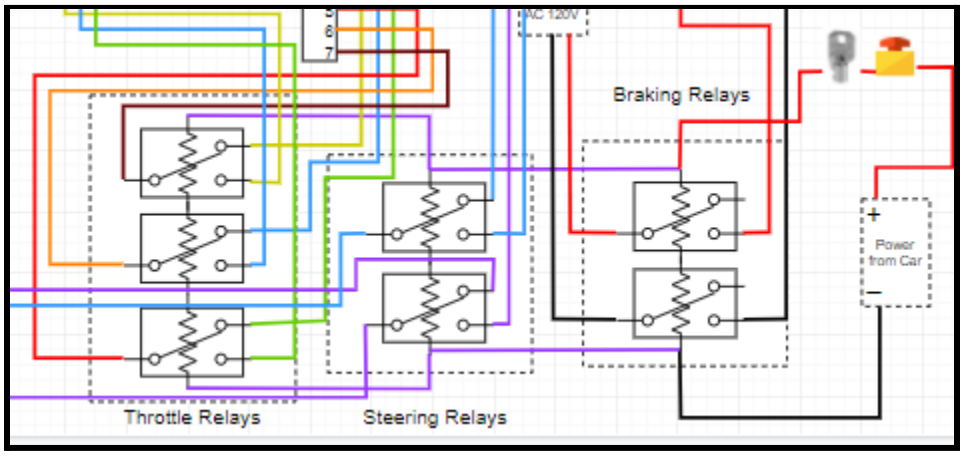


Fig. 16. Diagram for autonomy relays and E-Stop inputs

4.6 SENSOR SUITE

The Ford Escape came with two Blackfly S USB3 FLIR cameras, a 64 point LiDAR system and two GPS units. The cameras are able to form point cloud images and will be used by

future teams to gather data to avoid obstacles, vehicles and pedestrians. The next step for the cameras is to create a point cloud system represented as matrices from each camera's feed. A python code can be used to do this. The 64 point LiDAR system was replaced with a 128 point system made by Ouster for more accurate recordings. The Ouster system uses a 24V power supply and connects to the main computer using a USB to Ethernet cable. The LiDAR sensor should remain unplugged when not in use, because if it remains plugged in it will continually scan its surroundings and wear itself out. The sensor communicates to the computer using IPv4. When the LiDAR is fully connected, open a terminal window and use the command: *ping -4 -c3 os-122148000210.local* to activate it.

If this command runs and a package is sent and received, this indicates the sensor is connected to the computer and is able to send and receive data. Once the sensor is connected it needs to be configured, and will need to be reconfigured whenever disconnected from the main power. The configure command is:

python3 -m ouster.sdk.examples.client os-122148000210.local configure-sensor

Once the sensor is configured, it is ready to collect and receive data using the command:

python3 -m ouster.sdk.examples.client os-122148000210.local live-plot-signal

This will broadcast a black and white image that will use the live data from the sensor. The Ouster sensor data is able to be used by rvis and broadcast live data that will be used when the car is made autonomous. To do this first enter catkin_ws/src. Run the command:

source ../devel/setup.bash

Then the command:

roslaunch outster_ros ouster.launch sensor_hostname:=10.5.5.50 udp_dest:=10.5.5.1 metadata:=metadata.json

Open rvis in a new terminal, add a point cloud by topic and change the fixed frame to “os_sensor” this will pull up a live feed from rvis. Figure 17 shows the final imaging when the above prompts are initialized successfully.

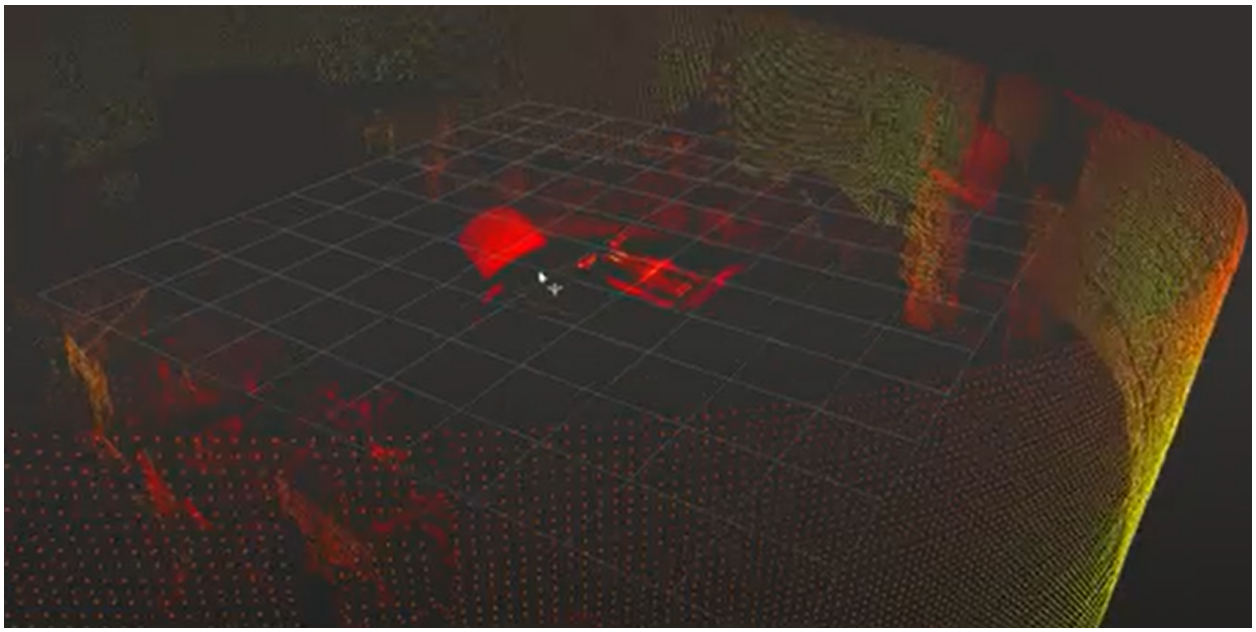


Fig. 17. LiDAR imaging taken in reactor room where car is worked on

4.7 SOFTWARE

The software suite on the Escape is meant to combine all of the concepts above into one centralized computing environment. This was accomplished with the Robot Operating System (ROS) middleware suite, which handles the operation of programs and the communications between different programs in the suite. Doing so allows for quick and clear communication methods between Python and Arduino files, as well as an external joystick and the car’s onboard CAN bus system.

The software suite can be broken up into the 3 groups of hardware they reside on: the CAN bus, the onboard desktop computer, and the Arduino Unos. The CAN bus is the car’s

onboard computer that allows the car's many microcontrollers and devices to communicate. This can be read using a PCAN to USB adapter that translates the CAN bus' messages into hexadecimals that the computer can read. The onboard computer houses many of the programs that will be run for the operation of the car. This includes scripts such as the CanBusReader.py script, which will decode the CAN messages, the joystickControl.py script, which will take input from a user-controlled joystick, and the operator.py script, which will take the decoded CAN messages and joystick input and decide what messages to send to the Arduinos accordingly. Lastly, the Arduinos convert the messages from operator.py into voltages, which then are sent to each subsystem to operate them. Once these systems change, their CAN messages will as well, and operator.py compares the new CAN values with the joystick input to make sure everything is operating properly. This is done using PID tuning, and is present on both the braking and steering subsystems.

Using these feedback loops, every subsystem can be checked for accuracy. This is especially important with the stepper motor for the brake, because, unlike the electrical systems like the torque sensor for the steering or the potentiometers for throttle that are relatively easy to set, this is a physical system. This means it is much more prone to error, such as a bad step from the motor or a loose or slipped cable from the motor to the brake. If an event like this happens, it will be immediately clear, as the CAN bus value for braking will not match the input value, allowing the software to adjust the brake as necessary. A full software logic tree is provided in Figure 18.

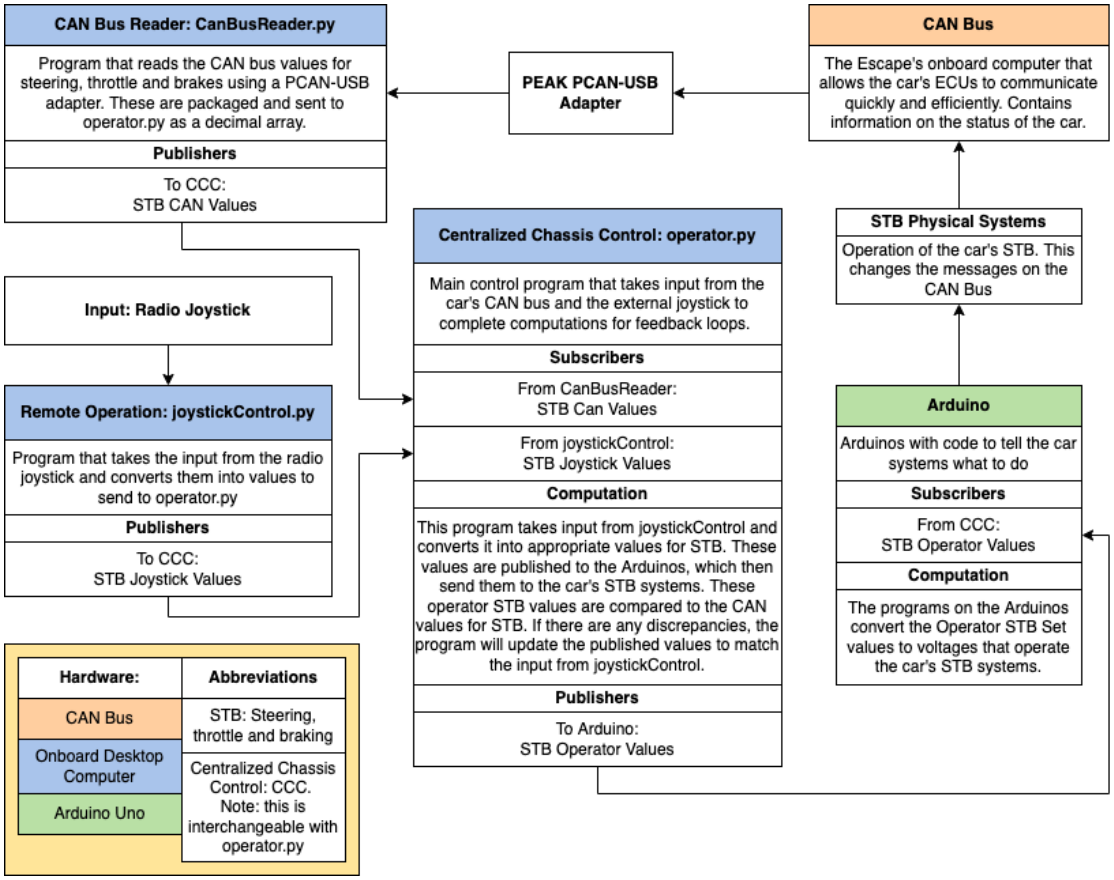


Fig. 18. Software system diagram

5 MATHEMATICAL AND NUMERICAL ANALYSES

5.1 MOTION MODEL COMPARISON

In order to validate the car's motion systems, students decided to compare reactions to a computer motion model of the car. Luckily, the Compositional Systems Lab at the University of Arizona created a Gazebo plugin called "catvehicle" as a testbed for autonomy. The model used is a Ford Escape (shown in Figure 19). This plugin features a controllable car model complete with sensors. Both the motion model and the actual car were fed acceleration step inputs for throttle, and the velocity of each system was measured in response. Trials were run at with inputs

of 8% throttle, 10% throttle, and 12% throttle, and the reaction was plotted for easy comparison in Figures 20, 21 and 22 respectively.

Fig. 19. Catvehicle plugin on Gazebo

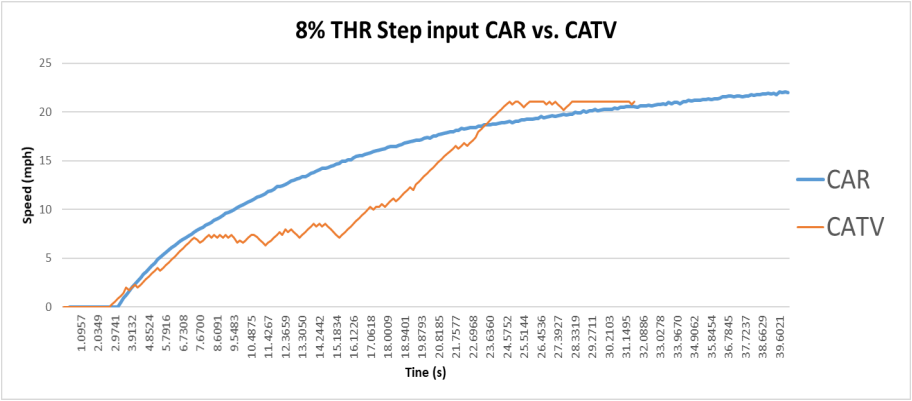


Fig. 20. Trial with 8% Throttle

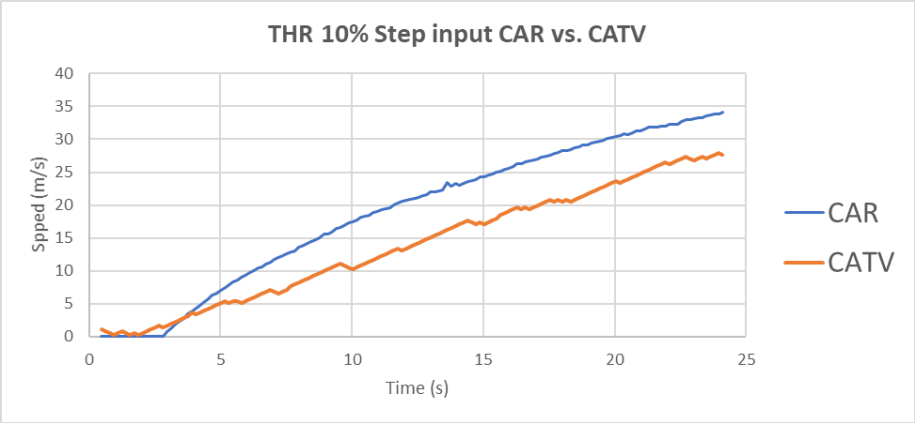


Fig. 21. Trial with 10% Throttle

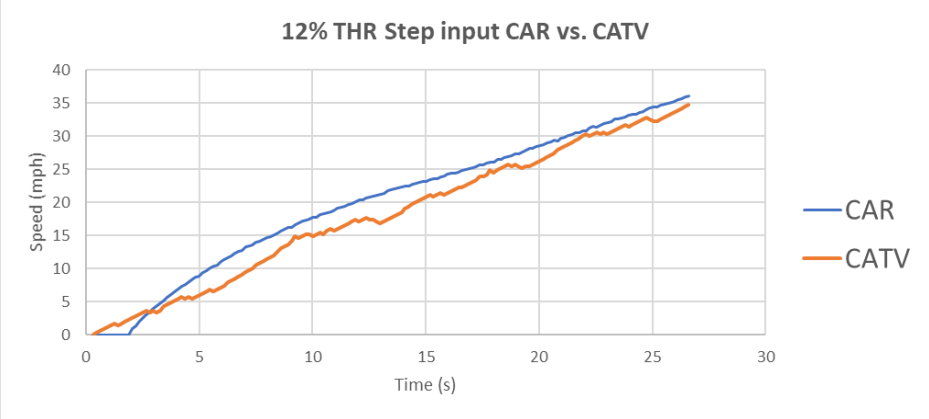


Fig. 22. Trial with 12% Throttle

Interestingly, the results showed that the real car has a smoother acceleration than the simulated catvehicle. Overall, the car performs very similarly to the motion model, however there is much more internal noise in the modeled system. The systems are expected to converge more closely toward at cruising speeds because external factors like static friction have been overcome and air friction has plateaued. This behavior is true of the 8% and 12% trials, however the two systems failed to converge on our 10% trial.

5.2 PID TUNING

In order to minimize the error in brake position and eliminate jitter from the overall motion, a PID controller was implemented. PID controllers require tuning to achieve the desired motion, meaning that our Kp, Ki, and Kd constants must be adjusted in the calculation of the output command. In order to tune our system, the brake controller was fed a constant step input of reaching a brake position of 124 from an unpressed position of 127. There are several methods of tuning a PID controller, but our team chose to follow the Ziegler-Nichols method. This consists of first finding a system's ultimate gain (Ku), or the Kp value that causes constant undamped oscillation. Several equations relate Ku and its period of oscillation to the suggested values for the constants. The relation used in our braking system can be found in figure 23 below.

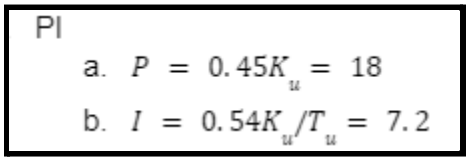


Fig. 23. Ziegler-Nichols equations for a P and I term only

Results were recorded for a PID controller using just a proportional (Kp) term (Ki=Kd=0), and a proportional and integral term (Kd=0). The trial with all three terms led to a worse result than just P and I and, therefore, is not shown. Figure 24 shows the attempt of the system to reach a brake position of 124 with a Kp (left) and a Kp and Ki value (right) found from the Ziegler-Nichols method with a Ku of 40 and period of 3 seconds.

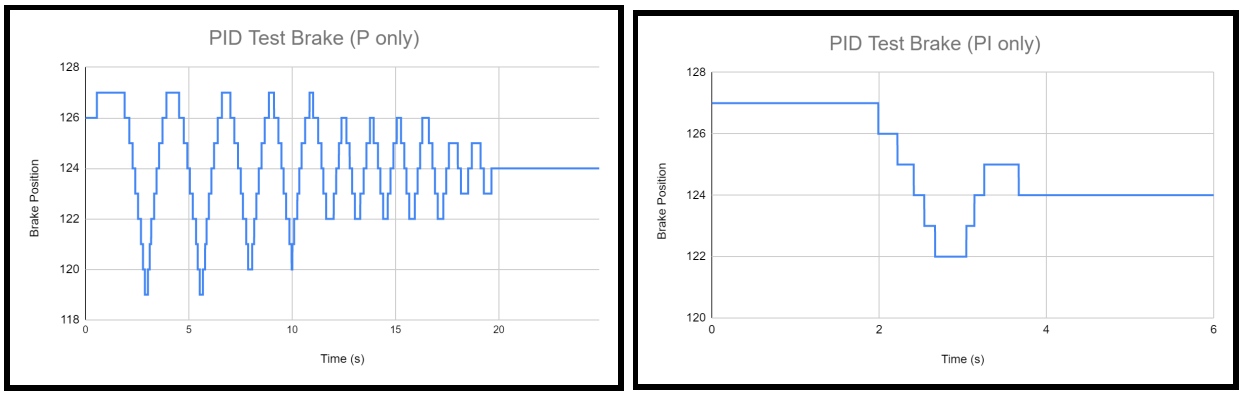


Fig. 24. PID Brake Test (124 position step input), Left: P only, Right: P & I only

The results show the improvement of the system dampening the oscillation after adding in the integration term. Only two oscillations before reaching the setpoint is fairly good, although further tuning may be beneficial for future teams to pursue.

6 EXPERIMENTAL VALIDATION

The braking and throttle systems were tested for response times in order to ensure that the target specifications have been met for the car to be “safer than a human operator” as previously quoted from the customer. Step inputs for a brake position of fully pressed and a throttle value of 25% were fed into both systems and the time of the response was measured in Figures 25 and 26.

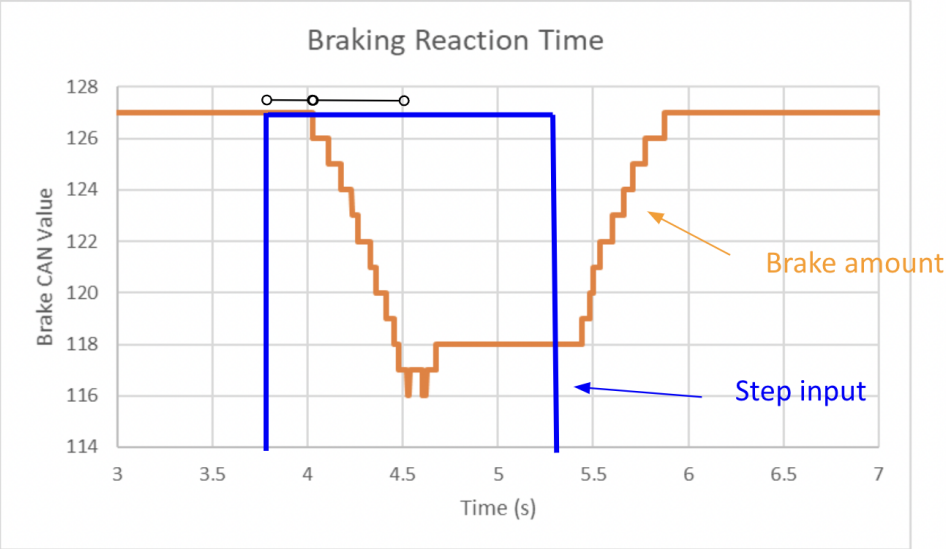


Fig. 25. Braking test results

The black lines and dots above the data illustrate the two separate response times of the system. The first two dots represents the time between the input given to the system and the beginning of the system response. For braking, this response time was 0.13 seconds. The second span of dots illustrates the time for the system to reach its desired state. These two response times were summed to show the total response time from user input until system completion. The total braking response time was 0.7 seconds. In fact, the human response time (moving foot from throttle to brake) is 0.75 seconds, which makes the system safer than a human operator.

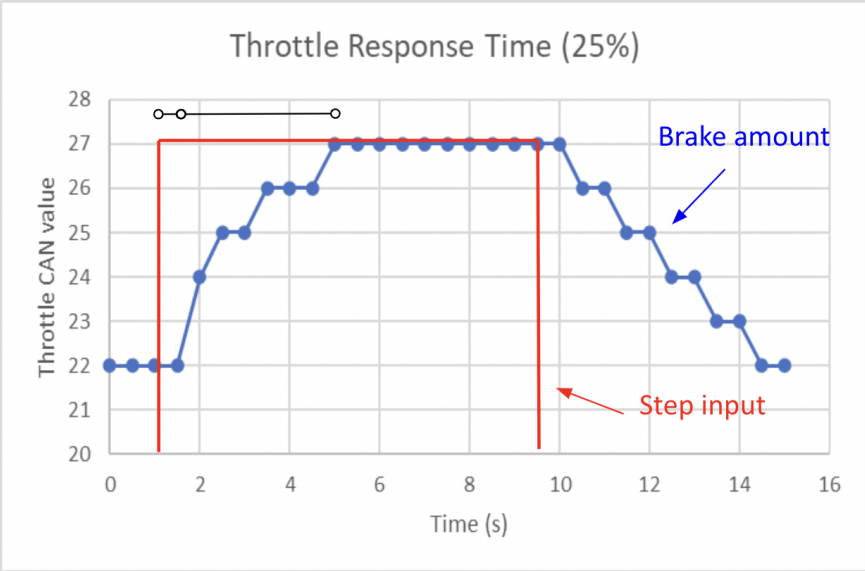


Fig. 26. Throttle test results

This test was repeated on the throttle system. The system response time was .72 seconds (user input until system reaction), and the total response time was 5.22 seconds (user input until reaching 25% throttle). Due to the nature of throttle response, it takes more time for the system to reach the value specified by the step input.

7 OPERATIONS MANUAL

As a division of VICTOR Lab, we have developed this operations manual in order to record the general operating procedures necessary for the safe handling of the drive-by-wire 2008 Ford Escape Hybrid outlined in the document. This document has been designed to outline fundamental structure, development, and deployment of this system.

Before operating the system, it is essential to ensure that the 12V battery in the front of the car is connected, and that the high voltage (HV) battery in the trunk of the car has the

disconnect switch turned to the locked position. The locked position will have the crossbar handle of the switch sitting horizontal, as can be seen by the lock arrow close to the switch.

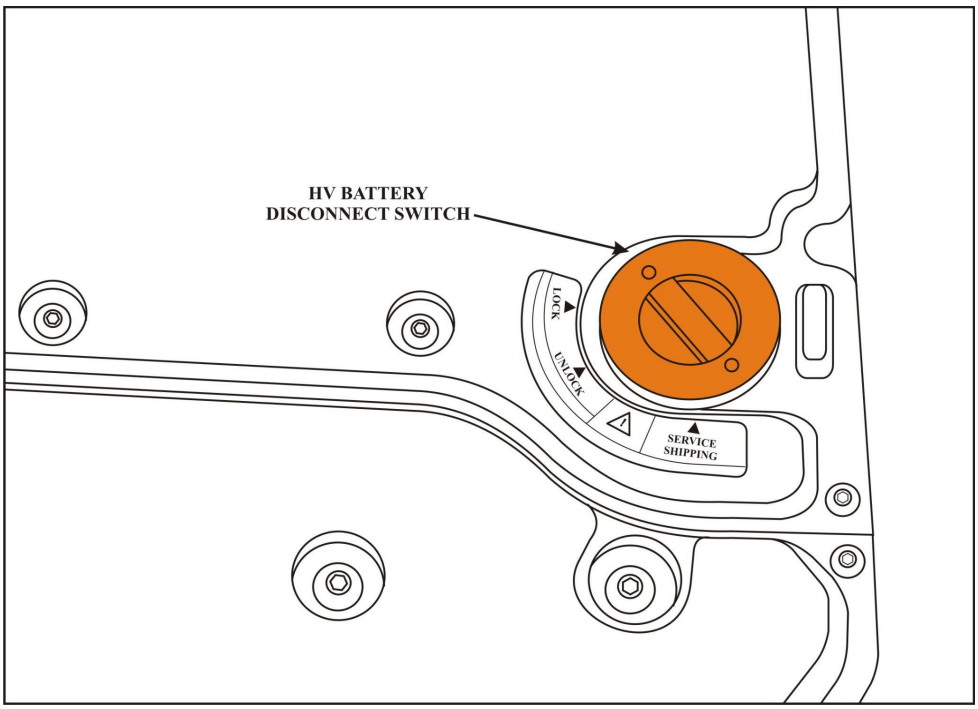


Fig. 27. Hybrid battery disconnect switch diagram

For the purpose of this description, it will be assumed that the car's non-native electronics are plugged into wall power via the power strip in the back of the car. To switch over to battery power, first locate the strip, the computer, the power inverter (labeled 'Pure Sine Inverter'), the battery, and the switch in the back of the vehicle. Before unplugging the power strip, make sure the computer is properly shut down. After this, unplug the strip from the wall and plug it into the outlet on the side of the power inverter. The battery is connected to a pink plastic clip, which will connect to a matching red clip connected to the car that should be located in the back of the vehicle. Plug the battery's pink clip into this red clip. Lastly, turn the safety switch (small gray box) on using the red key zip-tied to the switch. Once this is complete, you should be able to turn

the inverter on using the small circular power button located next to the outlet, which will supply power to the system.

Fig. 28. From left to right, front to back: Power strip, computer monitor, joystick, power inverter, battery-kill switch, 12V battery, computer.

Next, make sure the Arduinos are connected to the car systems. There are 4 total pin connections that need to be plugged into the arduinos. The pin connector with the red, yellow, and green wires is for the braking system and should be connected to the arduino with a single connector. The connector with the blue, yellow, black, and green wires is the throttle connector and should be connected to the only 4-pin connector on the other arduino. The remaining two 3-pin connectors are both for steering. The one with a blue, purple, and black wire should be

connected to the middle pin connector. The last connector has blue, purple, and red wires and should be connected to the final left 3-pin connector. Finally, the two blue cables attached to extenders from the computer should both be connected to the arduinos.

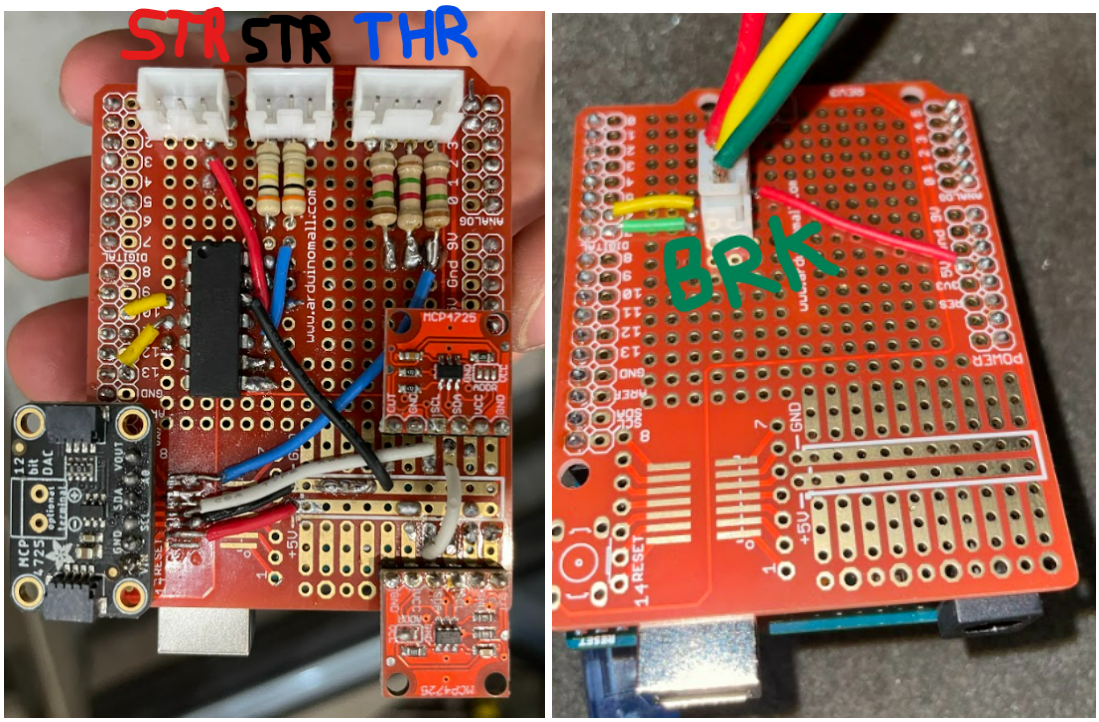


Fig. 29. Pin connector locations

Make sure the CAN bus reader (white cable with a small box on it and a red light) is plugged into the car near the driver seat and the computer as well. The system should now be independently operable.

All software needed for the operation of the vehicle is located in the *catkin_ws/src/car_chassis_controller_s22* directory of the computer. All directories named hereafter will be found in this directory, and so the *catkin_ws/src/car_chassis_controller_s22* part will be left out of its location for brevity. The software is handled using the ROS middleware suite, which means you do not need to be in this directory to launch the commands *except* for the

CanBusReader.py script, which has not been integrated into the launch file. Because of this, you must be in the directory of CanBusReader.py to run the script.

To do this, turn the computer on and open a terminal window. Use the change directory command (*cd*) to get into CanBusReader.py's directory:

cd catkin_ws/src/car_chassis_controller_s22 /src/pcan

Open another terminal window in this directory by right-clicking on the current terminal window and selecting "New Tab" or "New Window.". You should now have two terminal windows opened in the *pcan* directory. This is done because the CanBusReader.py script is not killable with Control+C, so the terminal it is running in must be closed every time by right-clicking on it and selecting "Close window" manually. To avoid having to change into CanBusReader.py's directory each time you close and reopen the terminal, you can *cd* into the directory in one terminal, then use it to open another in the same directory.

Next, open the Arduino IDE. Open the Brake_Control and Steering_Throttle_Control files, which are located in the *src/arduino* directory. Make sure that each of these files is uploaded to the proper Arduino by uploading them and checking to make sure that the proper Arduino flashes for each upload. The Steering_Throttle_Control file should be uploaded to the Arduino with many DACs on it and the Brake_Control file to the Arduino with a single pin connector for the stepper driver.

The next step is to make sure that the emergency stop button and autonomy switch are disengaged. These are located in the center console of the car. The left E-stop button must be in the up position (twist until it pops up) and the autonomy switch must be towards the right. Both of these switches are wired in series so both need to be in the right position to send signals to the car. Engaging either system cuts power to the relays and returns manual control of the vehicle.

The E-stop button doesn't automatically stop the car; user intervention is necessary. The following is the most important instruction in the user manual. If the E-Stop needs to be flipped during operation, it is imperative to not turn the autonomy back on without first turning the car off. The E-stop and autonomy switch need to be in the correct positions before the car is turned on. Turning autonomy on while the car is on sends a voltage spike to the systems that can burn out the motor in the power steering assembly.

In one terminal, write but do not run the command *roslaunch car_chassis_controller_s22 car.launch.* In the other terminal, write but do not run *python CanBusReader.py.* The launch file will run all necessary programs for teleoperation including operator.py, rosserial for Arduino communication, and joystickControl.py. CanBusReader.py will decode the CAN messages and send them to operator.py.

Run the *car.launch* command. You should see ROS start up, some information text, and then a line saying "*Controller Not Connected: Press A."* Before pressing A, press both triggers a few times and spam a few buttons like the D-pad and joysticks. This helps to establish the communication between the joystick and the computer and reset the potentiometers in the joystick. After this, press A. The message should change to "*BRAKE NOT ZEROED."* Now you can run the *CanBusReader.py* command. You should see this terminal window begin to print CAN values, and eventually the other terminal window will say *"Brake Zeroed."* Once this is printed, the car should be ready for teleoperation. Remember to press the brake before taking the car out of park. The operation of the car uses the left trigger for brake control, right trigger for throttle, and the left stick for steering.

To stop the autonomous driving mode, stop the car using the brakes and put it into park. Kill the *car.launch* terminal with Control+C, and then close the *CanBusReader.py* terminal by

right clicking and selecting “Close.” To return the car to non-autonomous mode, press the emergency stop and flip the switch from autonomous mode to non-autonomous mode. Again, it is imperative to turn the car all the way off before switching it back into autonomous mode.

8 CONCLUSIONS AND FUTURE WORK

The team was able to accomplish its main goals and successfully completed a test drive of the system. External electromechanical and wiring systems that allow for the control of the car’s steering, brake, and throttle systems have been installed. Software to interface with the external control system, the joystick controller, and the car’s internal CAN bus for the teleoperation of the car have been created. PID controllers to allow for feedback control of the steering and braking systems using data from the CAN BUS have been designed. The sensor suite for LiDAR, GPS, IMU, and depth cameras have been mapped and designed. All subsystems are controlled together, meaning the car is now fully drive-by-wire, and is able to drive on the road, albeit slowly and safely in a parking lot.

Future work on the system involves several small fixes to make the system more reliable with more safeguards to prevent damage. Adding the functionality to account for user input in the steering wheel during operation and adding components to the relay circuit to prevent the spike in voltage when the system is turned on are some examples that will help future teams. However, future work on the car will primarily involve making the car fully autonomous. The first step of this process will be in creating more reliable PID controllers for steering, throttle, and braking. Once these subsystems can be more accurately controlled with an external controller, sensors can begin being integrated into the system. This sensor data can be used to

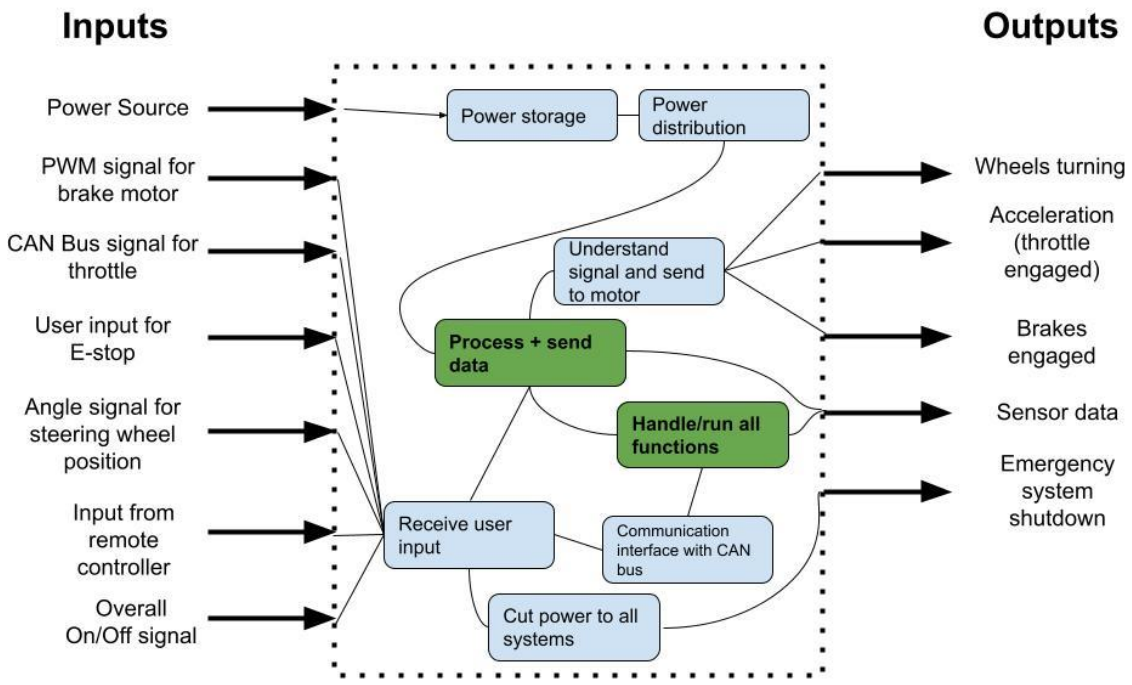
create semi-autonomous processes such as self-parking and speed control by interpreting data from the outside world. Once enough semi-autonomous control loops are integrated into the system future teams can create more autonomous features and eventually a fully autonomous vehicle. It is imperative for future teams to more accurately control the integral driving subsystems and convert the data points recorded by the sensors into a complete image of its surroundings in order to perform logical processes.

Appendix

Appendix A: QFD Chart

	Importance (Customer Needs)	SPECIFICATIONS:	Turning resolution	0 - 60mph acceleration time	Response time for Braking	Response time for throttle	Obstacle avoidance distance	Response time for steering	braking distance from 60mph - 0mph	Throttle (speed changing) resolution
			1	2	3	4	5	6	7	8
Direction of Improvement			↓	↓	↓	↓	○	↓	○	↓
CUSTOMER NEEDS:										
Drive-by-wire system	5		4	4	3	3		4	3	4
Must have steering and override mechanical or electrical capabilities	4			3	5		4	3	5	2
Strong transparancy required of the autonomous system to relay its "thought" to passenger	4		3		4	3	4	3	4	3
Data can be collected for GPS mapping, but should not be used outside of vehicle or to improve vehicle's driving capabilities	3			1	1		1			
Physical controller	3		2	2	2	2		2	3	3
TECHNICAL IMPORTANCE, DIFFICULTY, AND PRIORITY:										
Technical importance (calculated, absolute)			38	41	60	33	35	50	60	49
Technical importance (calculated, percentage)			10.4%	11.2%	16.4%	9.0%	9.6%	13.7%	16.4%	13.4%
Technical importance (calculated, rank)			6	5	1	8	7	3	1	4
			0.2 0.15 0.1 0.05 0							
Units			deg	s	s	s	ft	s	ft	mph
Marginal Target Value			3	10	0.5	0.5	5	0.5	200	3
Ideal Target Value			1	8	0.1	0.1	10	0.1	100	1
Technical Difficulty			2	2	3	3	1	3	4	2
Technical Priority (calculated)			76	82	180	99	35	150	240	98
Technical Priority (calculated, rank)			7	6	2	4	8	3	1	5
			250 200 150 100 50 0							

Appendix B: Concept Generation- Functional Decomposition



Appendix C: Concept Generation- Different Solutions for Braking

Sub-Functions	Solutions				
Actuator	Stepper Motor	Servo Motor	DC Brush Motor	DC Brushless Motor	Hack power-braking system within car
Power	12V Car battery stepped up to	Power Supply	Solar Panel	Portable Power Cell	Wall power
Input control	GUI on onboard computer	Wired Potentiometer	Code positions looping in code	External joystick Controller	
Microcontroller Board	Arduino Mega/Uno	Texas Intruments Microcontoller	FPGA board (XEM8350)	ASM Microcontroller	Parallax P2 Chip
Sensors (feedback control methods)	Sensor suite for measuring speed	Ultrasonic distance sensor mounted behind pedals	Quadrature encoder	Absolute encoder	CAN Bus brake pedal position

Appendix D: Concept Generation- Different Solutions for Throttle

Sub-Functions	Solutions					
Method	Stepper Motor to pull throttle	Servo Motor to pull throttle	Send DigiPot signals to emulate signals from throttle potentiometers	Replace actuator for opening throttle air intake with our own mechanism that we control	Pneumatic actuator to push throttle	
Power	12V Car battery	Power Supply	Solar Panel	Portable Power Cell	Wall power	
Input control	GUI on onboard computer	Wired Potentiometer	Code positions looping in code	External joystick Controller		
Microcontroller Board	Arduino Mega/Uno	Texas Intruments Microcontoller	FPGA board (XEM8350)	ASM Microcontroller	Parallax P2 Chip	
Sensor (Feedback control method)	Triple potentiometers imbedded in throttle pedal	Ultrasonic distance sensor mounted behind pedals	Quadrature encoder	Pitot tube inside throttle assembly to measure air intake	CAN Bus speed reading (spedometer)	Sensor suite (Radar, Lidar, GPS) for measuring speed

Appendix E: Concept Generation- Different Solutions for Steering

Sub-Functions	Solutions				
Method	Motor built in line with steering column	Send a torque sensor message to the ECU to emulate power steering reading	Exterior motor attached to a belt wrapped around the steering wheel	Replace power steering module with a controllable hydraulic power steering module	
Power	12V Car battery	Power Supply	Solar Panel	Portable Power Cell	Wall power
Input control	GUI on onboard computer	Wired Potentiometer	Code positions looping in code	External joystick Controller	
Microcontroller Board	Arduino Mega/Uno	Texas Intruments Microcontoller	FPGA board (XEM8350)	ASM Microcontroller	Parallax P2 Chip
Sensor (Feedback control method)	Triple potentiometers imbedded in throttle pedal	Sensor suite (Radar, Lidar, GPS) turning radius	Absolute encoder on steering wheel	Accelerometer mounted on car to measure turning	CAN Bus steering wheel position reading

Appendix F: Concept Generation- Other Solutions Info

The **Carnegie Mellon Boss**, a Chevy Tahoe, did have steering, throttle, and brakes controlled with actuators. It was mentioned that normal driving controls are available in case of emergency override, so it is implied that sensors to detect override are present. For internal processing and data computation, the Boss uses a CompactPCI Chassis with 10 Core2Duo Processors.

The **Stanford Stanley**, a Volkswagen Touareg R5 did have steering, throttle, and brakes mechatronics controlled with actuators. Processing was done internally in the car with a system of external computers running Linux software hidden in the trunk. Linux was used due to its excellent networking and time sharing capabilities. Race software was executed on three of the six computers, with the fourth computer to log race data, leaving two idle computers. One of the race computers handled solely video processing and the other two race computers executed all other software polling the sensors and controlling the

brake, steering, and throttle. There was no mention on whether sensors to detect manual override were present (other than an E-stop button), if there was an intuitive graphical interface to show processing, or a physical controller.

MIT's Talos vehicle, a Land Rover LR3, includes servos for throttle and brake actuation. The car has a "Situational Planner" interface, which does show vehicle decision making ahead of time. Processing is done on a Quanta blade server computer system. It was mentioned that there is a "proven and safe method" for switching between manual and autonomous driving, but it is unclear whether it is some sort of E-stop or sensors on the throttle, brake, or steering themselves. There was no mention of a physical controller.

Virginia Tech's Victor Tango car is a Chevy Volt. Very limited information was found about the car but from what was gathered, the steering and throttle/brake systems are not controlled with actuators, but computing was done internally.

Some autonomous vehicles that are currently in the industry include **Volvo's driveMe** system and **NuTonomy's driverless taxi**. Both had steering, brake, and throttle systems controlled by actuators, as well as internal secure processing (nuTonomy used machine learning) but did not have sensors to detect override other than an E-stop.

REFERENCES

Kottayil, N. K. (Ed.). (2021, August 24). *What is an autonomous vehicle? - definition from Techopedia*. Techopedia.com. Retrieved March 21, 2022, from https://www.techopedia.com/definition/30056/autonomous-vehicle

Laukkonen, J. (2020, February 13). *Take a ride in Waymo's self-driving car*. Lifewire. Retrieved March 29, 2022, from https://www.lifewire.com/waymo-self-driving-cars-4171314

PID theory explained. National Instruments. (2020, March 17). Retrieved May 8, 2022, from https://www.ni.com/en-us/innovations/white-papers/06/pid-theory-explained.html

Schwall, M., Daniel, T., Victor, T., Favarò, F., Honhold, H., (2020). Waymo public road safety performance data. https://storage.googleapis.com/sdc-prod/v1/safety-report/Waymo-Public-Road-Safety-Performance-Data.pdf

Shipley, D. (2021, November 17). A Trucker Shortage? Bring on the Robots. Bloomberg.com. Retrieved February 24, 2022, from https://www.bloomberg.com/opinion/articles/2021-11-17/autonomous-vehicles-could-solve-the-u-s-truck-driver-shortage

Wevolver. (2020). 2020 autonomous vehicle technology report Wevolver. Retrieved from https://www.wevolver.com/article/2020.autonomous.vehicle.technology.report

Jmscslgroup. (2013). *Jmscslgroup/catvehicle: A macroscopic multivehicle tesbed and hardware-in-the-loop simulator for autonomous driving*. GitHub. Retrieved from https://github.com/jmscslgroup/catvehicle