

Small Scale Intelligent Vehicle Final Design Report

Preparers:

Sarah De Rosier

sderosie@calpoly.edu

Dominic Riccoboni

driccobo@calpoly.edu

Paul Rothhammer-Ruiz

prothham@calpoly.edu

Sponsor:

Charles Birdsong, PhD

Acknowledgements

The ProgreSSIV team would like to thank Dr. Peter Schuster for his advice, patience, and for keeping the whole team sane throughout the duration of this project. They would also like to thank Charlie Revfem for his willingness to share his time and expertise. Charlie put almost as many hours advising this project as we put into it.

Abstract

The objective of the Small Scale Intelligent Vehicle Project is to develop a small scale, semi-autonomous vehicle platform for use in an intelligent vehicles course that is to be taught at Cal Poly. This document provides an in-depth description of the scope of the project, its finalized design, testing results, and recommended future work. This project is in collaboration with the project sponsor, Dr. Charles Birdsong, and the project advisor, Dr. Peter Schuster.

The project was begun last year by the μ Laren team. They did a considerable amount of work in adding complex hardware to an RC car. The additions to the car this year were mostly on the software side, with some hardware improvements and sensor selection and integration. The project began with a lot of research and benchmarking, outlined in the Background and Benchmarking sections of this document. This led us to a preliminary design where we selected sensors to purchase, investigated new motors, investigated the necessary hardware changes, and began the process of designing the control algorithms that are to be implemented.

The sensor selection, motor selection, and hardware changes were finalized in the Final Design section of this document, along with the firmware architecture and controller designs. The Manufacturing section details a lot of the firmware, sensors, and controller implementation. Testing results are summarized in the Design Verification section. Each team member's roles are outlined in the Project Management section. Future work and recommendations are outlined in the Project Management section.

Table of Contents

1 Introduction	1
2 Background	2
2.1 Benchmarking	2
2.2 Mechanical Background	3
2.3 Vehicle Modeling	4
2.3.1 The Bicycle Model	5
2.4 Sensors	10
2.5 Firmware/Software Background	12
2.5.1 High Level System Architecture	12
2.5.2 Firmware	13
2.6 Control Algorithms	14
2.6.1 Acceleration control	14
2.6.2 Adaptive Cruise Control	17
2.6.3 Lane Departure Assist System	22
2.6.4 Stability Control	26
3 Objectives	29
3.1 Customer Needs	29
3.2 Engineering Specifications	30
3.3 Boundary Sketch and Boundary Diagram	35
3.4 Functional Decompositions Diagram	37
4. Concept Design Development	38
4.1 Mechanical Concept Development	38
4.2 Sensor Concept Development	44
4.3 Controller Design Development	50
4.3.1 Acceleration Control	51
4.3.2 Adaptive Cruise Control	54
4.3.3 Lane Keeping Assist	61
4.3.4 Stability Control	65
	1

4.4.1	Teensy Firmware Design	72
4.4.2	Simulink Software Considerations	75
4.5	User Interface Concept Development	76
4.5.1	User Interface Hardware Ideation	77
4.5.2	User Interface Hardware Pugh Matrix Result	78
5.	Final Design	78
5.1	Motor Selection	79
5.2	Vehicle Model	82
5.3	Sensor Integration	84
5.3.2	LiDAR Sensor	86
5.4	Control Algorithm Structures	88
5.4.1	Adaptive Cruise Control	88
5.4.2	Lane Keeping	94
5.4.3	Stability Control	96
5.5	Firmware Design	97
5.5.1	Teensy to Raspberry Pi Firmware Changeover	97
5.5.2	Firmware Architecture Design	99
5.5.3	Firmware Design Updated	99
5.6	Bill of Materials and Budget	102
6	Manufacturing	104
6.1	Firmware Manufacturing	105
6.1.1	Raspberry Pi Firmware Tasks and Functions	105
6.1.2	Mastermind Task	105
6.1.3	Functions	107
6.1.4	Firmware Manufacturing Updated	108
6.2	Simulink Communication Subsystems Manufacturing	115
6.3	Sensor Manufacturing	118
6.4	Controller Manufacturing	124
6.4.1	Lane Keeping Controller	124
6.4.2	Stability Controller	127
7	Design Verification	128
7.1	Design Verification Plan	128

7.1 Sensor Testing	129
7.1.2 LiDAR and GPS	129
7.1.4 Encoder	131
7.2 Controller Testing	132
7.2.1 Lane Keeping Control	132
7.2.2 Yaw Control	133
7.2.4 Vehicle Model	135
7.3 Usability Testing	136
8 Project Management	136
9 Conclusion	137
9.1 Firmware and Simulink Recommendations	137
9.2 Controller Recommendations	139
10 Works Cited	141
Appendices	143

1 Introduction

Dr. Charles Birdsong, a member of the mechanical engineering faculty at Cal Poly, San Luis Obispo, has proposed the development of a course in Advanced Driver Assistance System (ADAS) feature design. As a part of the course, students will learn how to design ADAS features, and then practice implementing them on a small-scale vehicle platform designed for this purpose. Last year, he served as the sponsor for a senior project team by the name of μ Laren, and they began to develop this vehicle platform. This project is a continuation of their efforts, and will focus on improving the manufacturability and mechanical performance of the vehicle platform, while also adding features such that it is customized for use in the classroom. Much of the technical information presented in the Background section of this document was sourced from the μ Laren Final Design Report, which is listed here as a reference [Miley, Phillips, and Grant]. The target students of this course do not have a background in mechatronics, so this vehicle platform must be able to be operated by a controls student without previous mechatronics experience. The vehicle platform should provide this functionality in a form that is modular, expandable, and portable to future technological developments to ensure the longevity and usefulness of the platform.



Figure 1. The vehicle platform developed by last year's senior project team. This year's team, ProgreSSIV, intends to expand upon the work that was done last year in order to customize the vehicle for use in an intelligent vehicle controls course at Cal Poly, San Luis Obispo.

In addition, Dr. Birdsong is interested in sharing this platform with other universities who want to do autonomous research on a small scale, without having to spend the time and money to develop a platform themselves. To facilitate this, the platform should be easily manufacturable and as low cost as possible without sacrificing performance.

2 Background

This project is unique in that there has been a lot of previous work on this topic in the form of senior projects and several different master's theses over the years. This section is a compilation of our research on the technical background of this project, compiled from the uLaren team's Final Design Review and a few different master's theses from Cal Poly, as well as our own independent research. The background information is organized by technical topic.

2.1 Benchmarking

The benchmarking for this project is primarily concerned with investigating small-scale vehicles in the market that have been developed to test ADAS and autonomous algorithms. The three cars that were studied are UC Berkeley's BARC (Berkeley Autonomous Race Car), MIT's RACECAR (Rapid Autonomous Complex-Environment Competing Ackermann-steering Robots), and Donkey Car's Raspberry Pi controlled RC car. The most important features of these cars that were considered were the material and part selection, sensor integration, mechanical structure, motors, and the controller.



Figure 23. Images of the three platforms- BARC (left), RACECAR (middle), and Donkey Car (Right)

The BARC is a 1/10th scale vehicle platform developed by students at UC Berkeley to test autonomous features including: drifting, lane changes, and obstacle avoidance. Like to our project they sourced some parts from an RC car package, but also manufactured some custom parts. Because they are only using the stock single RC car motor to drive all the wheels, their product requires a lot less modifications to the hardware. The structure is made from water jet cut aluminum that encloses the internal wiring, giving the car a cleaner look. The BARC also has a similar hardware architecture in that it has a computer chip similar to the Raspberry Pi processing the controls and then a lower level controller similar to the Teensy sending out motor commands. Instead of using Simulink, the BARC platform uses ROS, which stands for Robot Operating System. This is an open source software library that includes many autonomous robotic algorithms for easy integration into a system. For the feedback, the car uses: a camera, an ultrasonic sensor, an IMU, and encoders on each wheel. The platform has been used for many projects including: Autonomous Lane Keeping and Obstacle Avoidance, Autonomous Drifting, and Learning Model Predictive Controller for Racing. The platform website includes a bill of materials and documentation explaining how to assemble all the parts. Its cost per unit is \$500. [Gonzales, J.]

MIT also has a small-scale intelligent vehicle that they have developed. Their car, RACECAR, is also built around an RC car platform. Like the BARC, the RACECAR only uses one motor, so a lot of the modifications to the RC car were done to create a platform for the Graphical Processing Unit (GPU) and fixtures for the sensors. The RACECAR is controlled by an NVIDIA Jetson GPU running parallel machine learning algorithms that output steering and throttle commands based on the input from the sensors. The sensors include: a LiDAR sensor, a stereo camera, an IMU, and a depth camera. A bill of materials with all components and prices was not found, but based solely on the sensors being used and the RC car kit, it is estimated that the setup costs at least \$3600. The platform is being used for an undergraduate course in autonomous robotics titled: “Robotics, Science, and Systems”. The course includes topics such as: motion planning, geometric reasoning, kinematics and dynamics, state estimation, tracking, and map building. They have developed very thorough documentation on building the car and running it with different algorithms.

The Donkey Car is a modified RC Car designed by DIYRoboCars to allow hobbyists to learn and develop autonomous robot algorithms. DIYRoboCars is an organization and community of autonomous robot enthusiasts that meet up to race and show off their smart RC cars. The Donkey Car is specifically designed so that not a lot of manufacturing methods are needed to put it together so that a hobbyist can build it in their garage. The only manufacturing required is a 3D printed base and a 3D printed elevated fixture for the camera. Similar to our platform, the Donkey Car is also controlled by a Raspberry Pi (without Simulink) and uses a Pi Camera as feedback. The DonkeyCar is advertised as only costing \$200. The DonkeyCar does not use a Traxxas RC car, but instead is built around a “Magnet Car”.

See the Quality Function Deployment (QFD) in Appendix B for more information on how the current state of the vehicle compares with these benchmarks. The QFD will be in the Objectives section.

2.2 Mechanical Background

The main objective of the uLaren team was to adapt an RC car into a vehicle that could be used as the platform for a modern controls course in the application of intelligent vehicles. The challenge in achieving this was allowing for independent drive of each wheel. One of the key features of the course lab will be implementing a multiple input, multiple output (MIMO) control system on this car in order to control each motor and the steering. Given that RC cars on the market only offer models with a single motor to control either all four wheels or the rear two wheels, an RC car needed to be configured to have each wheel driven by a separate motor. This required designing motor housings, motor shaft couplers, a suspension frame, and a chassis. The main mechanical focuses of the uLaren team was to integrate these manufactured parts with the stock parts of the RC car. Our main mechanical objectives are to bring down the cost of the vehicle and improve the manufacturability.

According to the uLaren team’s bill of materials, the cost of one of the car units in parts approaches \$900. The motors and motor drivers, however, have a total retail value of \$5200, bringing the total cost of the vehicle to \$6100. Other than the cost of the motors and motor drivers, the most expensive collection of parts is the RC car kit which costs \$450. This kit

includes all the parts needed to construct an RC car. Many of these components, however, do not end up being used in the car because they are not needed—for example, the motor that is included in the kit. The uLaren team recommended that next we should consider purchasing the individual RC car parts that are needed instead of buying the whole kit. This would cut down the per unit cost considerably.

Currently, it takes a minimum of four manufacturing methods to construct all the parts needed on the car. These manufacturing methods include: a water jet to cut the nylon chassis and aluminum suspension supports, a 3D printer to print out the motor mounts and sensor fixtures, a mill to drill the chassis holes, and a CNC to cut the shaft couplers. It is preferable to have fewer manufacturing methods so that it takes less time to complete the manufacturing of a unit and also, so it is easier to replace parts if they were to break.

Aside from the adjustments to the manufacturing methods and the parts selection, the mechanical portion of the senior project would also consist of improving the fixture and positioning of some of the components. One of the areas for improvement is the steering mechanism which has a mechanical interference that occurs when the wheels are steered far right or far left. Another area for improvement is the camber of the front wheels which changes when the suspension is compressed. Lastly, there is significant play in the steering of the wheels. For example, when the steering servo is held constant so that it does not rotate, the wheels can still be slightly rotated in either direction. It is likely that many of these defects arose because of the integration of manufactured parts and RC stock parts. As a result, it may be difficult to completely overcome these issues, and design decisions will have to be made on what tolerances are permissible given the accuracy requirements of the control system. These mechanical issues have to be solved, however, in order for the vehicle modeling to reflect the actual dynamics of the vehicle.

2.3 Vehicle Modeling

In order to implement semi-autonomous vehicle control, a model of the vehicle must be developed. This vehicle model will act as the “plant” in all control loop operations involving the vehicle sensors, and is critical to developing semi-autonomous functionality. Its level of accuracy in representing the vehicle’s true dynamics will determine the level of accuracy that will be able to be achieved with the vehicle controls. There are many different vehicle models that have been developed for this exact purpose, and they deal with differing levels of complexity. The level of complexity of a vehicle model is measured by the degrees of freedom, or axes of motion, of the vehicle model. Different types of vehicle models and the degrees of freedom that they model are listed below in Table 1 [Schramm].

Table 1. Degrees of freedom of different vehicle models.

Model Type	Number of Degrees of Freedom
Single Track Model, Linear	2
Single Track Model, Nonlinear	3-7
Twin Track Model	14-30
Complex Multibody System Model	>20
Finite Element Model	>500
Hybrid Model	>500

The Single-Track Model is also referred to as the bicycle model, and is the simplest to implement of the models listed. It is explored in the section below. The other models listed are much more complex than the bicycle model, and consequently it would be impossible to implement them in the course of this year-long project. As a result, only the linear and non-linear bicycle model will be considered for use in this project.

2.3.1 The Bicycle Model

Since modelling a vehicle with thousands of parts can be exceptionally complex, the bicycle model provides a balance between complexity and basic functionality that would be appropriate for a small-scale product such as our vehicle. The bicycle model analyzes a vehicle such as a car by “collapsing” it into a two-dimensional model of a bicycle. To do this, the front and rear tires are collapsed together, forming a two-dimensional representation of the car. This is illustrated in Figure 2 below.

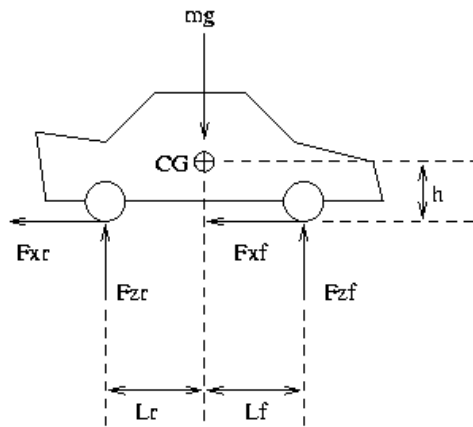


Figure 2. Image of the bicycle model. The bicycle model imagines a vehicle as a two-dimensional object by assuming the steering angles are the same for each set of wheels at the front and rear of the car [Bicycle Model]

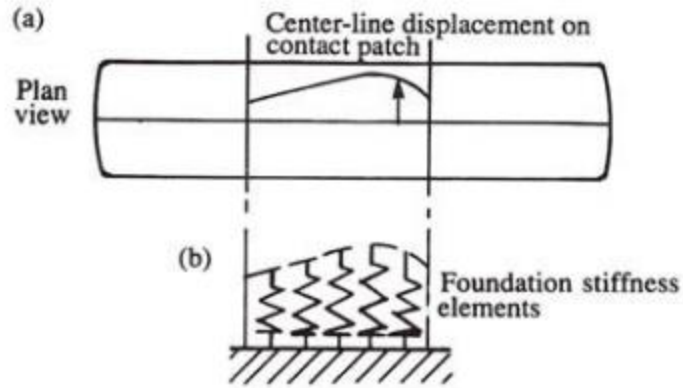


Figure 5. The elastic tire model models the tire as a collection of independently-acting elastic components [Stevens].

A commonly used empirical tire model is known as the “Magic Formula”. The Magic Formula is used extensively in industry and in other applications in order to model lateral forces that a tire experiences under driving conditions. The advantage of an empirical model is that it works over a wider range of tire conditions, and therefore can more accurately capture the behavior of a system. The Magic Formula is given below.

$$F_{yi} = D \sin\{C \tan^{-1}[B\alpha_i - E(B\alpha_i - \tan^{-1}(B\alpha_i))]\}$$

where $D = \mu F_{zi}$
and $B = \frac{C_{\alpha i}}{CD}$

The coefficient i represents differentiation between the front and the rear tires, and would be replaced appropriately with either an “f” or an “r”. D represents the influence of the coefficient of friction between the tire and the road--this controls the peak lateral force that the tire is able to withstand. B is the stiffness factor--this controls the linear region of the lateral force curve, and is related to the shape factor C . The curvature factor E controls the curvature of the lateral force curve.

The Magic Formula is used to generate a curve that relates the lateral force that a tire experiences to the slip angle of a tire. This curve is used to predict the level of grip that a tire has. The grip of a tire is influenced by both the lateral and longitudinal forces that it experiences. The relationship between lateral and longitudinal tire forces and their influence on grip can be described by a friction circle.

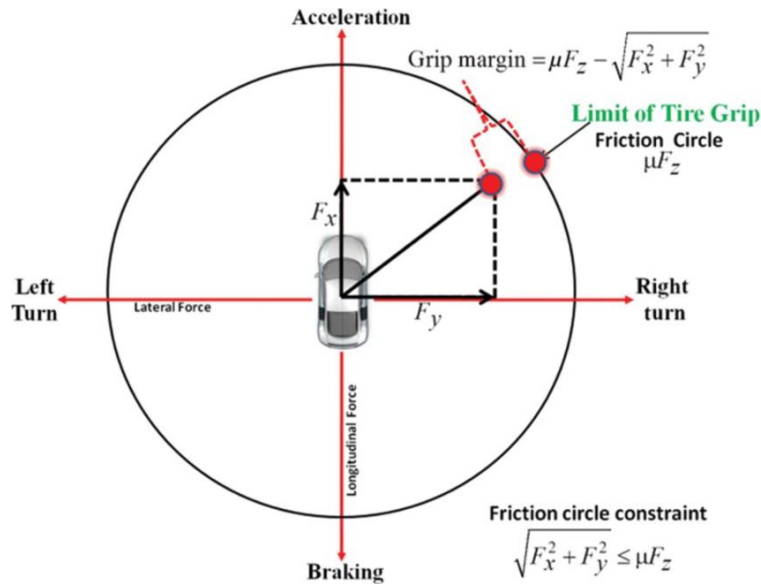


Figure 6. The friction circle provides a graphical representation of the limits of grip of a tire in terms of the g's it experiences under different driving conditions [Singh].

The friction circle provides a graphical description of how the forces experienced by a tire's contact patch influence its dynamic behavior. If the forces are such that they lie within the tire's friction circle, the tire will maintain contact with the ground and not slip. If they lie on the border of the circle, or outside of the circle, then the tire begins to slide. The behavior of the tire is determined by how far outside of the friction circle it lies at any instant in time [Stevens].

Motor Modeling

In order to model the drive torques, a model of the vehicle drive system is required. The current vehicle platform is powered by four Maxon flat brushless motors. The performance curve for these motors is shown below in Figure 7. The red lines represent different voltages at which the motors can be operated. On the y-axis is the motor RPM, and the x-axis represents the motor torque. The black curves represent the performance of the motor torque over an RPM range, both when the motor is operating at steady state and when the motor is accelerating.

A common way of modelling a brushless DC motor is with a linear model that neglects internal static friction. This model provides a basic way to model motor performance, without sacrificing too much in the name of accuracy.

The linear model of a DC Brushless Motor is represented by the following equation:

$$T_m = \frac{-K_b K_t}{R_a} \omega_m + \frac{K_t}{R_a} E_a$$

where $K_b = \frac{1}{K_v}$

The tractive force at the driven wheels can then be found using the following equations,

$$F_{x,i} = \frac{T_m G}{R}$$

where $i = f, r$. [Stevens]

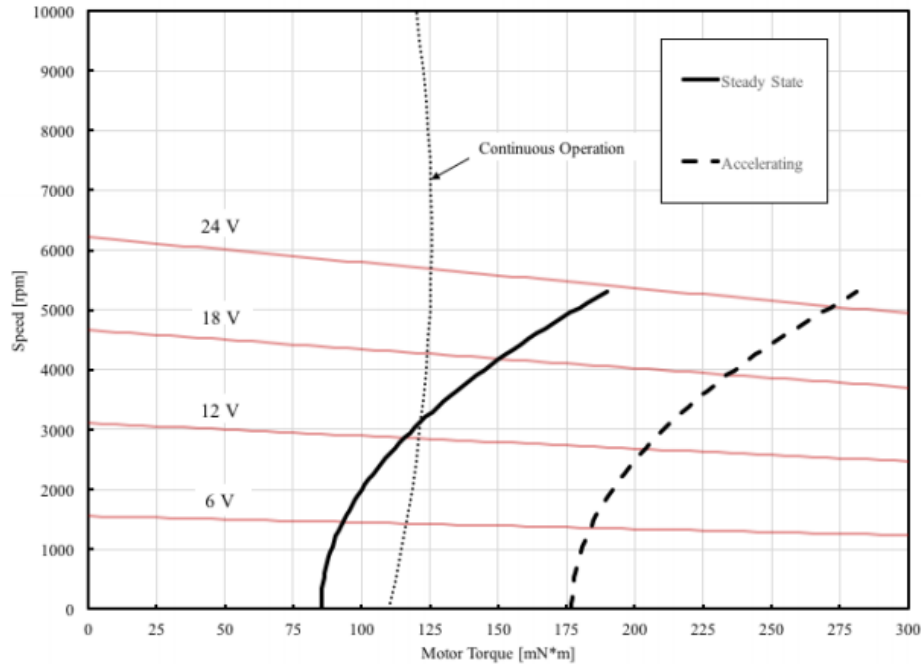


Figure 7. This performance curve specifies the torque characteristics of the motor as related to the voltage supply and the motor RPM [Maxon Motor].

With the tire forces and motor torques specified, and applying the principles of the conservation of linear and angular momentum, the final equations of state can be found [Schramm].

The vehicle model will be used in the ADAS features that will be developed for the course. The operation of the ADAS features will require a controller to modify the vehicle plant, or model, in order to carry out the ADAS behaviors. The various control algorithms used for each of the ADAS features are explored in section 2.5.

2.4 Sensors

Semi-autonomous technology would not be possible without the use of sensors. Sensors enable a vehicle to keep track of its own motion and path relative to the environment around it. As semi-autonomous technology in mainstream vehicles is becoming more popular, vehicles are being equipped with more sensors. For example, the newest Tesla model has 8 cameras, ultrasonic sensors around the perimeter of the car, Inertial Measurement Units (IMUs), GPS sensors, and a radar sensor.

The current vehicle platform contains an IMU, a Time of Flight sensor distance sensor, and a camera. In this project, we will need to filter the IMU data in order to refine perception of the environment and add real-time lane detection algorithms on the Raspberry Pi. These features would provide the feedback needed for students in the course to construct Simulink models implementing control algorithms such as lane keeping and adaptive cruise control.

One of the most important sensors in semi-autonomous cars is the distance tracking sensor, which assists in collision avoidance. Tesla uses Stereo Vision to find the distance to obstacles. Stereo Vision requires two cameras that are pointed in the same direction but have a distance between them. Using triangulation geometry, the distance to the object can be calculated based on where the object appears in each image. A simple way to visualize this concept is to close your left eye and look at a close object in front of you. Next, open your left eye and close your right eye. Notice how the object's apparent location shifted. Both left and right eye "images" are sent to the brain where the true position of the object is determined. The drawbacks to Stereo Vision are that it performs poorly in the dark, is computationally intensive, and the region that it can scan for objects is limited by what appears in the images.

The Google autonomous car relies on a LiDAR sensor for obstacle detection, rather than cameras. LiDAR (Light Detection And Radar) works on the principle of reflected light. The LiDAR device emits a laser beam of light and determines the distance based on the time of flight—that is, the time it takes for the laser beam to hit the object and return to the sensor. Many LiDAR models achieve full 360° range by spinning the sensor in a circle. Since the sensor is capable of receiving readings at a very high frequency, it can be spun rapidly without compromising resolution of the data. As a result, the LiDAR sensor has extremely high resolution and great range. However, the sensor's performance drops during certain weather and lighting conditions. It is also prohibitively expensive. One of the early LiDAR models, the Velodyne HDL-64E, cost around \$75,000 when it was introduced. Although the prices have dropped, they remain somewhat high (Velodyne® now offers a model for \$8000), leading companies to search for alternative methods with performance similar to the LiDAR sensors. While there are cheaper sensors based on LiDAR technology, they are not capable of the range of the full-scale LiDAR sensors. These sensors are sometimes referred to as ToF (time of flight) sensors or 1D LiDAR sensors. One example is the ST Electronics VL53LOX. With a price tag of just \$15, it is much more accessible than a full-scale LiDAR sensor [Adafruit].

An ultrasonic sensor emits sound and counts the amount of time it takes for the wave of sound to return. Ultrasonic sensors do not give very high resolution with regards to where an object was detected. They simply output a distance corresponding to an object that was within the ultrasonic sensor's range cone. The benefit of ultrasonic sensors is that they are cheap. The drawbacks are that the sensors noisy and do not perform well on materials that absorb sound such as fabrics.

Unfortunately, the current camera that is on the vehicle platform cannot give three-dimensional distance readings, but it can be calibrated to find the distance of any point on the ground in front of it based on where that point appears on the image. This may not be useful in terms of collision avoidance, but can be very useful in determining the distance to an upcoming curve in the lane.

Of course, before finding the distance to lane lines, one must first detect the lane lines. A popular technique for detecting lane lines is the Hough Transform. The Hough Transform takes in a black and white image that has gone through an edge detection algorithm. A popular edge detection algorithm is the Sobel Edge detection algorithm. It outputs pixel points that correspond with the location of a lane line. Both the Hough Transform and the Sobel edge detection algorithm can be implemented using Simulink blocks. [Gonzalez, Rafael]

2.5 Firmware/Software Background

In this section, the high-level system architecture of the current platform as designed by the uLaren team is briefly explained. This architecture consists of the orientation of the hardware components how they relate to and communicate with each other. The Firmware portion describes the current firmware architecture and how it may not ultimately be suitable for a final product. A previous intelligent vehicle design thesis is investigated to help illustrate this point and to assist in framing the firmware and software design challenge ahead.

2.5.1 High Level System Architecture

The system architecture of the vehicle (see Figure 8) was designed in such a way as to distribute the operational and computational burden among various nodes. This scheme facilitates the modularity and testability of the system by isolating and grouping various related functionalities. High level control of the vehicle is to be done by a Simulink model embedded on a Raspberry Pi single-board computer. A general-purpose microcontroller called the Teensy houses the firmware and serves as the middleman between the high-level control and the various vehicle components. Currently this microcontroller manages sensor data acquisition and communication of data among the nodes, but it will likely take on additional functionality as the capabilities of the vehicle are expanded. Lastly, four motor controllers perform closed loop control of the motors in either torque or velocity mode. Due to the nodal design of the system architecture, the hardware components are linked by communication buses, transferring data through protocol such as I2C, CAN, SPI, and Serial. Some potential challenges of this arrangement are discussed as part of the Firmware section below.

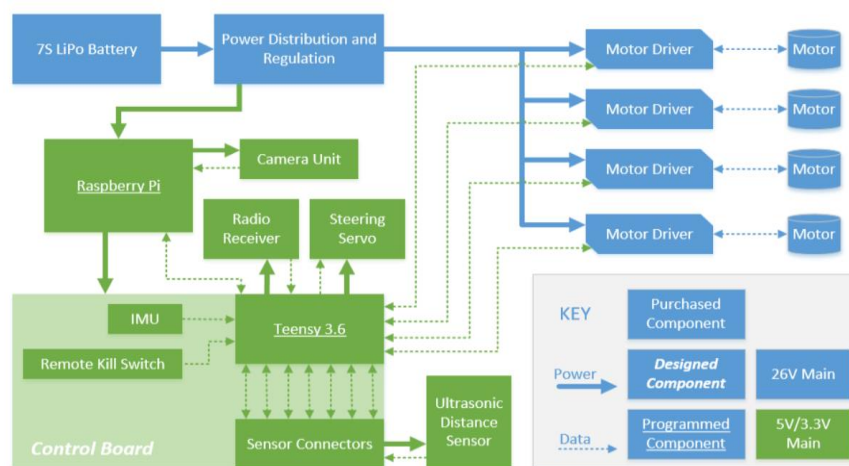


Figure 8. Current computer architecture of the vehicle, as developed by the uLaren team [Miley, Phillips, and Grant].

2.5.2 Firmware

Firmware is code that is embedded on a system—typically for its entire lifetime. It is held in non-volatile memory such as Electrically Erasable Programmable Read-Only Memory (EEPROM), or flash, and is generally used for lower level control of a device's hardware and operational logic. In less complex devices it can serve as the complete operating system. In our case, the firmware refers to code that will exist on the Teensy microcontroller that will not be available for students or other users of the device to edit. In the current design the firmware can either run in Manual (RC mode) or Simulink Mode. In Manual mode a handheld remote is used to send throttle and steering commands and the Teensy relays them directly to the motor drivers and servos. There is no interaction between the Teensy and the Raspberry Pi in this mode. Simulink Mode, however, allows the vehicle to run via Simulink computations alone—the Teensy is not currently able to receive throttle inputs from the user while receiving actuations from Simulink. This is something that may need to change if the course curriculum requires it or if it is something deemed helpful to researchers in general.

Something that was not developed last year is an outline of the current design of the firmware. An outline is necessary to investigate if the present setup will meet the project requirements. One of the first tasks, then, as a part of this senior project was to develop a diagram that would identify the names and responsibilities of tasks within the specific subsystems or device drivers, and the details of the interfaces between subsystems [Barr]. This diagram is shown in Appendix A. Currently, the code cannot be well represented by a collection of independent tasks - rather its dynamic functions are written one after the other without clear distinction or grouping within a repeating while loop. This composition diminishes modularity and multitasking ability because it does not group code of similar purpose distinctly and it does not provide for timing considerations other than the loop's cycle time. Additionally, many of the sub-processes occur within infinite loops that can halt the program or produce unpredictable timing behavior. Products requiring firmware will often have explicit time requirements that ensure that explicit deadlines are met [Barr]. In the current version, when important messages are sent and when sensor information is acquired is not deterministic. If a process were to be implemented into the current framework, significant delays in cycle and response time and problems with data validity and relevance would occur.

We referred to the firmware architecture described in Thomas Steven's master's thesis to gain insight into what needs to be done to produce a successful product. Although the current architecture of our platform is different than the one he implemented, there are aspects of his design that are relevant, such as the principles Stevens used to develop the software aspects of his collision avoidance system. Among these are responsiveness and timing requirements, task and state machine code structure, and the porting of a Simulink designed model with that code structure [Stevens].

The approach he took was to create a real-time embedded system on a single-board computer running Linux. For a real-time control system, it is more important to have high responsiveness rather than high throughput. Stevens notes that embedded control systems must have “a high level of determinism and meet operational deadlines by guaranteeing a system response from an

event within strict time constraints.” Our system will likely have similar constraints if our vehicle is to perform low latency ADAS functions.

Stevens designed the architecture to comprise of multiple tasks, each encoded as finite state machines. The finite state machine and task-based approach allows for the “flexibility to add, delete, or change the flow of the program without impacting the overall system code structure” [Stevens]. Timing requirements and priority are managed by a Real-Time Operating System (RTOS) that can perform context switches between tasks in accordance with a scheduling policy. The extensibility described here will be essential to a long-lasting platform that can be expanded upon by future researchers. Implementation of a real-time operating system that encompasses all elements of our system is most likely not possible for this project, as important elements are separated by communication buses that have inherent limitations such as lag. However, it is possible that one proves necessary – if so, it will have to occur on a particular component independent of the others i.e. the Teensy and the Raspberry Pi both having separate real-time operating systems.

The entire collision avoidance system presented in Thomas Stevens’ master’s thesis, including the path planning algorithms and the vehicle dynamic control was prototyped in Simulink—another parallel to our vehicle. A key difference however is that students of the future intelligent vehicles course must be able to simply design a model and upload it to the Raspberry Pi for immediate testing. Stevens had to manually port over his Simulink controller into C++ and fuse it into the real-time operating system in a manner that preserved the control loop. While this was time consuming, it allowed him to meet the timing needs mentioned above. We, however, have to work within how Simulink ports the code—we cannot alter this relationship. A challenge we will face is to provide adequate control capability to the architecture when there is less control over timing and limited synthesis between the Raspberry Pi and the rest of the components. This will be an issue even if both the Teensy and the Raspberry Pi run an RTOS. The course will likely deal with much simpler control algorithms, so this may not prove to be an issue for students, but other users may put a higher burden on the platform.

2.6 Control Algorithms

Since the primary function of this vehicle is to be used in a vehicle controls course, the control algorithms that are relevant to the course will have to be designed such that they perform their intended functions. It is the goal of the future instructor of this course, Dr. Birdsong, that the vehicle be programmed for: acceleration control, adaptive cruise control, lane-keeping assist, and stability control. Sections 2.5.1-2.5.4 explain how each of these control algorithms will influence the behavior of the vehicle, and the different ways that they could each be designed. The main design decision will be what kind of controller to use. This controller may be different for each algorithm, because the performance needs of each algorithm will be different.

2.6.1 Acceleration control

An acceleration control algorithm is designed to get the vehicle to reach a desired acceleration. In the case where acceleration is produced from electric motors, this means varying the input voltage to the motors in order to control their acceleration.

A number of different controllers have been used in this application. The simplest is Proportional-Integral-Derivative (PID) controller. PID controllers have each a proportional gain, integrative function, and derivative function that modify the input. They can be represented by the following equation:

$$u(t) = K_p e(t) + K_i \int e(t) dt + K_d \frac{de}{dt}.$$

The constants K_p , K_i , and K_d are the multiplying factors, or gains, for each input modification. In PID control, these gains are fixed, and their values can greatly influence the behavior of the system. A block diagram of a basic PID controller is shown in Figure 9 below.

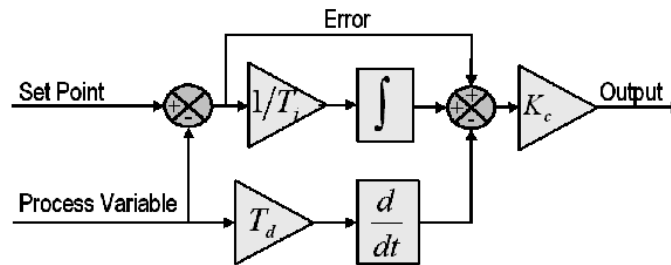


Figure 9. Block diagram of a basic PID controller [National Instruments].

Applied to acceleration control, the input to the PID controller would be the desired acceleration, and the output would be the measured acceleration. While this controller is easy to implement and robust, it doesn't handle the nonlinearities of a vehicle system well [Zhongpu and Dongbin].

One way of modifying the PID's performance is by combining it with Fuzzy control. The Fuzzy control essentially modifies the inputs and outputs to the PID controller, in order to make it more adaptive to nonlinear systems. A diagram of the interaction between the Fuzzy controller and the PID controller is shown below in Figure 10.

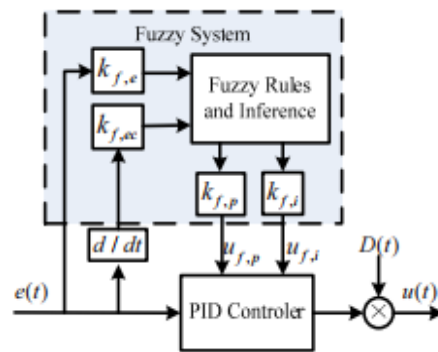


Figure 10. Block diagram of a hybrid Fuzzy PID controller [Zhongpu and Dongbin].

The Fuzzy controller operates according to a set of rules that determine the modifying factors for the PID, $k_{f,p}$ and $k_{f,i}$. By doing this, the Fuzzy control can alter the proportional and integral control constants of the PID instantly, allowing the PID to be more responsive to changing circumstances. For example, if the system were to experience large error, it could increase the value of K_p , allowing the system to have a more dramatic response to correct the error, while simultaneously reducing K_i in order to avoid saturation. When the error is then reduced, the Fuzzy controller can reduce the value of K_p and increase the value of K_i in order to improve precision of the system [Zhongpu and Dongbin].

One way of customizing the response of the system is by tuning the input and output modification values of the Fuzzy controller, $k_{f,e}$, $k_{f,ec}$, $k_{f,p}$, and $k_{f,i}$. One method of tuning these values is with Particle Swarm Optimizer (PSO). The PSO is a fitness function in which one can modify the performance parameters of a step response function: the rise time, or response time of the system, the percent overshoot, or the amount by which the controller “overshoots” the input step function value, and the integral of the absolute error. The PSO applies weight factors to each of these parameters such that the user can weight their importance to system performance. The PSO allows a final layer of system customization within the Fuzzy-PID hybrid controller [Zhongpu and Dongbin].

Another means of implementing acceleration control is with Model Matching Control (MMC). MMC utilizes a feed-forward loop in order to improve time response, and a feedback loop in order to reduce error. A block diagram of an MMC controller is shown below in Figure 11.

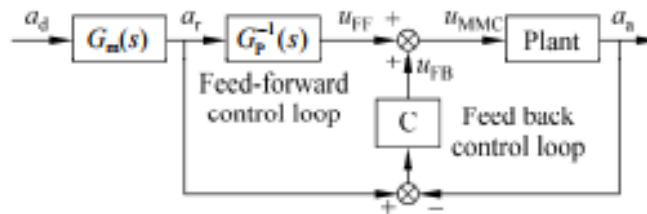


Figure 11. Block diagram of a Model Matching Controller [Yuejian, et al].

Often, the feedback control is simply a PID controller. However, in order to improve the transient response and reduce error, the feedback loop can be incorporated with Sliding Mode Control (SMC). In SMC, a sliding surface is defined such that

$$S = \dot{e} + a_s e + b_s \int e ,$$

where e is the error. Essentially, the sliding surface is just a PID controller. The surface, however, can only exist when

$$\dot{S}S < 0 .$$

The addition of the SMC in the feedback loop improves the transient response of the system over the PID controller in the feedback loop, and reduces error [Yuejian et al]

When designing the acceleration control algorithms, a selection will have to be made between a simple PID controller, a hybrid PID-Fuzzy controller, a hybrid PID-Fuzzy controller with PSO, a MMC-PID controller, and a MMC-SMC controller. More about the advantages and disadvantages of each of these methods and the final design selection can be found in section 4.3.1.

2.6.2 Adaptive Cruise Control

Adaptive cruise control (ACC) is a driver assistance technology in which the cruise control system can adapt to the traffic environment. Using a radar, ultrasonic, or laser sensor, the vehicle can detect other vehicles in the ACC vehicle's path. If those vehicles are slower moving than the ACC vehicle, the ACC vehicle will slow down to maintain a specified distance between itself and the slower vehicle. Once the ACC detects that there is no longer a slower-moving vehicle in front of it, it accelerates back up to cruising speed.

ACC works by maintaining distance control. It has a set following distance that it determines to be safe, and continuously measures the actual following distance. If the actual distance is greater than or equal to the safe distance, then the ACC operates in normal cruise control mode, where it maintains the cruising velocity set by the driver. If the actual distance is less than the safe distance, then the vehicle operates in distance control, or ACC mode. Figure 12 gives a visual of the operation of ACC.

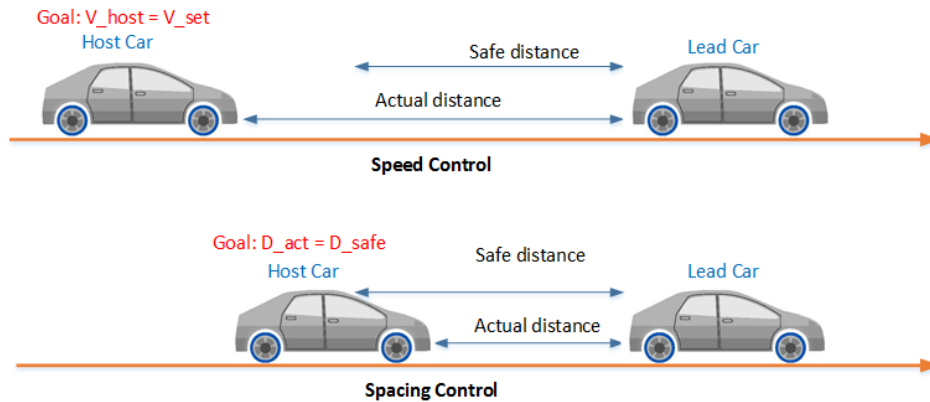


Figure 12. Diagram of the operation of Adaptive Cruise Control as a distance controller [Adaptive Cruise Control System Using Model Predictive Control - MATLAB & Simulink].

In order for ACC to work, it must be able to communicate with many of the subsystems of the vehicle. Once the cruise control is set, communication must be established between the sensor, the engine control module, the instrument cluster where the driver sets the cruising speed, and the brakes. The diagram in Figure 13 below demonstrates the communication routes between each of these entities.

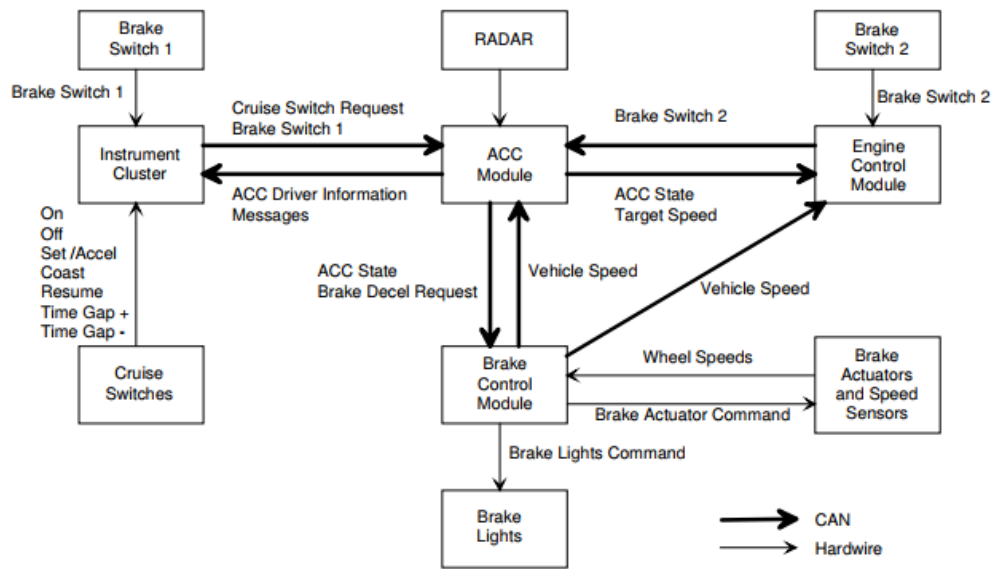


Figure 13. Control system diagram of Adaptive Cruise Control (ACC) [U.S. Software System Safety Working Group]

The difference between the control system interface in Figure 13 and the one that will be designed for the small-scale vehicle platform is that the vehicle platform doesn't require an instrument cluster in order for the user to set the cruising speed, and the vehicle is not equipped with brakes. Without those two elements, the ACC system simply becomes a communication route between the sensor, the ACC module, and the engine control module.

There are many different options for designing an Adaptive Cruise Control system for this application. The ones studied here are PID control, Model Predictive Control, and Fuzzy predictive control. The following paragraphs will describe each of these in detail.

With PID control, the ACC system is split into two different loops that perform different functions. The Outer Loop Control (OLC) performs distance tracking control. It uses a simple proportional controller to measure the discrepancy between the detected distance between the ACC vehicle and the leading vehicle and the desired distance between the two. It then uses this discrepancy to calculate the new reference speed for the ACC vehicle. The Inner Loop Control (ILC) then tracks how the ACC vehicle adheres to the new reference speed set by the OLC. A schematic of the relationship between the two control loops is shown below in Figure 14 [Sivaji and Sailaja].

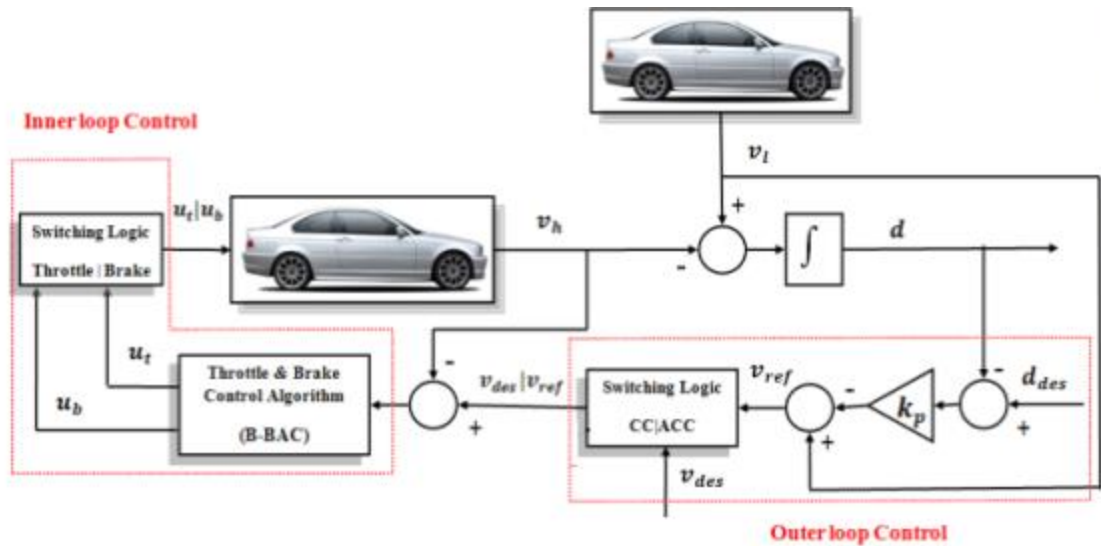


Figure 14. Schematic of relationship between Outer Loop Control and Inner Loop Control of PID-controlled Adaptive Cruise Control [Sivaji and Sailaja]

Switching logic is used to determine whether the vehicle will be adjusting the speed with throttle or brake control, and whether the vehicle is in cruise control mode or ACC mode. A Simulink model of this ACC system was constructed to test its performance. The model is shown below in Figure 15 [Sivaji and Sailaja].

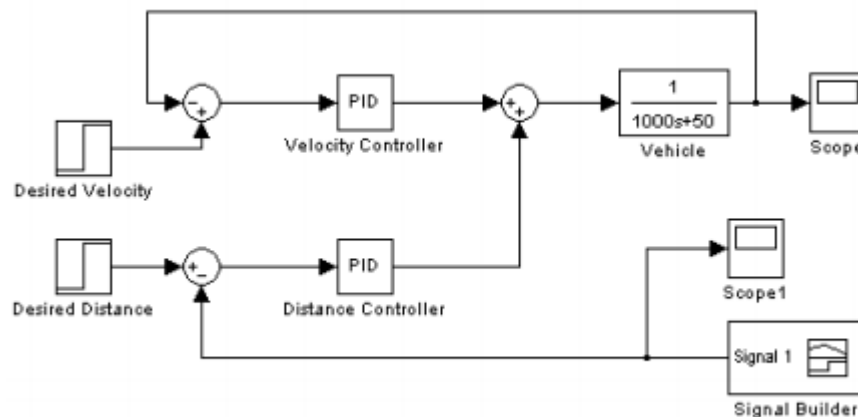


Figure 15. Simulink model with PID controllers for ACC system [Sivaji and Sailaja]

Two PID controllers were used for velocity control and distance control. The performance of this method is discussed in Section 4.3.2.

Another controller used in the Adaptive Cruise Control application is Model Predictive Control (MPC). A model predictive controller consists of three different parts: the MPC controller, the predictive model, and the optimization process. The MPC controller consists of two levels of hierarchy: the higher level calculates the desired acceleration or deceleration of the vehicle using the optimization process, and the lower level determines whether throttle or brake action is required, and then sends the commands to a Proportional Integral (PI) controller, where they're executed. The MPC controller that was studied is a linear MPC controller. The vehicle dynamics are represented within this controller using a state space represented, where the states of the model are the distance between the vehicles, the speed of the ACC-equipped vehicle, the relative speed between the ACC vehicle and the leading vehicle, the acceleration of the ACC vehicle, and the change in the acceleration of the ACC vehicle for each timestep, called the vehicle jerk. The acceleration of the leading vehicle, is introduced into the model as a disturbance. The model outputs the relative distance error, the relative velocity of the two vehicles, and the acceleration and jerk of the ACC vehicle [Kural and Guvenc].

The MPC uses the prediction model in order to extrapolate the future behavior of the model. The predicted state variables are calculated along the future horizon using future control inputs. These are predicted using the current state of the model, as well as the relative velocity and acceleration of the ACC vehicle. The prediction model for the current timestep is also modified by correction factors calculated from the output errors of the prediction model from the previous time step [Kural and Guvenc].

The optimization process optimizes the control objectives and applies vehicle constraints. It is also designed to regulate the control input in order to achieve a smooth control action. It performs a cost function that is minimized in each time step, where it uses weighting matrices to tune the relative importance of the output vector elements. The constraints are separated into two categories: objective and state constraints. The objective constraint is whether or not the vehicle has achieved a safe following distance, while the system constraints are the performance constraints of the vehicle. The controller must ensure that it doesn't ask the vehicle to perform outside of its capabilities, so the system constraints take into account the maximum velocity, acceleration, and jerk that the vehicle is capable of. The constraints defined as the distance between the two vehicles and the speed of the ACC vehicle are the prerequisites for which the controller attempts to solve the problem, while the performance parameters can be considered to be design parameters [Kural and Guvenc].

The cost function as it is subjected to the constraints listed above are then formulated as Quadratic Programming (QP), which is solved for each time step. The results of this function, called the control vector, are then input into the predictive model to determine the future step. The results of the control vector in the current time step are then fed to the lower-level throttle and brake controllers as the design point. These lower-level controllers then generate signals that are fed into a PI controller with closed-loop feedback in order to generate throttle and brake commands [Kural and Guvenc]. The performance of this controller as compared to the PID and other methods is discussed in section 4.3.2.

The next controller that was studied for an Adaptive Cruise Control application is an adaptive neuro-fuzzy predictive controller. With an adaptive neuro-fuzzy predictive controller, the sensors

obtain data from the leading vehicle and feed it to the Takagi-Sugeno (T-S) fuzzy model. The T-S fuzzy model consists of IF THEN rules that are able to translate the sensor data into a model of the leading vehicle. It then computes a discrete time model in order to generate a predictive model of the leading vehicle. Simply by subtracting the safe following distance from the predictive model of the leading vehicle, the predictive model of the ACC-equipped vehicle is generated [Lin et al]

Adaptive laws are established to generate tuning weights to improve performance of the model. These are generated using an energy function that describes the error between the actual state of the leading vehicle and the state predicted by the T-S fuzzy model [Lin et al].

The actual controller in the adaptive neuro-fuzzy control network is designed simply to minimize a cost function. The generation of this cost function in a form in which it can be minimized, however, requires a Fuzzy Neural Network (FNN). This network can be “trained” in order to improve its performance by using an energy function that computes the error for each discrete time step. The FNN uses IF THEN rules and the input predicted state of the ACC vehicle from the T-S fuzzy model in order to generate the estimated cost function in such a form that the controller can minimize it. Figure 16 below shows a diagram of the interactions between the various elements of the adaptive neuro-fuzzy predictive controller in an ACC application [Lin et al].

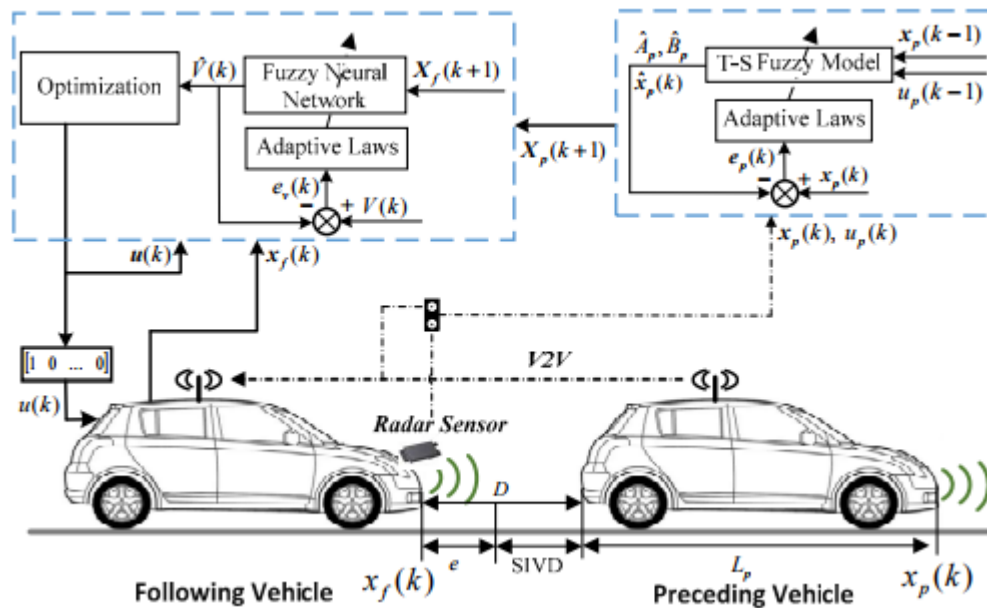


Figure 16. Interaction between various elements of the adaptive neuro-fuzzy predictive controller in an Adaptive Cruise Control application [Lin et al].

In this figure the T-S fuzzy model in combination with the adaptive laws are shown to be predicting the behavior of the leading vehicle, and then sending that information to the FNN, also acted upon by adaptive laws. The output of the FNN is then sent to optimization, where the

cost function is minimized, and the result is output to the lower-level controller. The lower-level controller then sends commands to the engine control unit and brake actuation unit, if necessary [Lin et al].

Each of these controllers: PID, MPC, and Adaptive Neuro-Fuzzy Predictive are evaluated based on their performance and level of appropriateness to this application in section 4.3.2.

2.6.3 Lane Departure Assist System

Lane Departure Assist Systems (LDAS) are a driver-assistance technology that monitor the vehicle's position within a lane. When activated, the system provides steering input in order to ensure that the vehicle stays in the lane. Vehicles with LDAS are equipped with cameras that can detect the vehicle's distance to the lane lines, as well as vehicle dynamics sensors that provide data on longitudinal and lateral velocity, yaw rate, and heading angle. See Figure 17 for a diagram of these vehicle measurements [Hoehener et al].

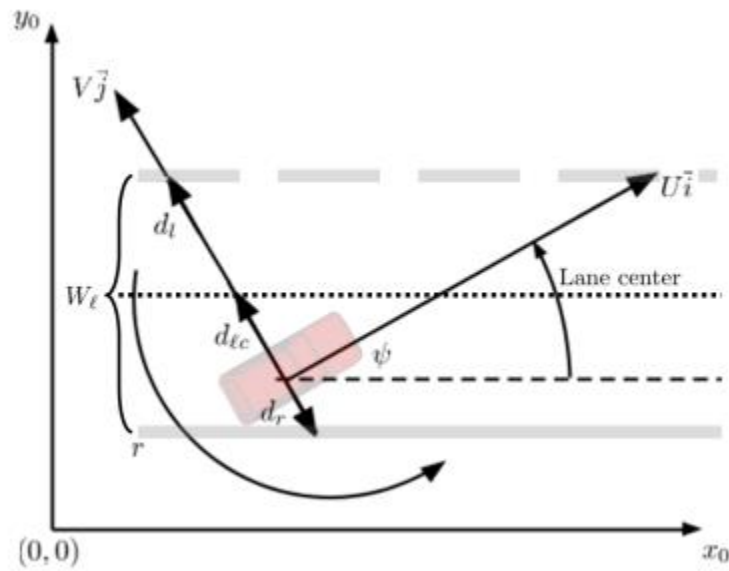


Figure 17. Diagram of vehicle dynamics measurements in LDAS. Longitudinal velocity is represented by U , lateral velocity by V , yaw rate by r , and heading angle by ψ [Hoehener et al].

These semi-autonomous systems take in data on the vehicle dynamics and position and determine whether lane departure is imminent with the current driver input. The system then overrides the driver input in an effort to avoid lane departure. These systems must guarantee that the car remains in the limits of the lane at all times, while simultaneously being conservative in ensuring that the driver's input is only overridden when absolutely necessary [Hoehener et al].

The controller methods that will be investigated here include PID control, controlled invariance, and state feedback. The following paragraphs will go over each of these methods in detail.

Section 4.3.3 will then go through a selection process for determining which of these methods is best suited to the needs of this project.

The first method to be explored is the PID method. In this study, a PID controller was used in two nested control loops that modified the vehicle's yaw rate. The input to this system is the steering wheel angle. The yaw rate, measured by a gyroscope, and the lateral offset, or the distance from the center of the vehicle to the center of the road. The lateral offset is measured from the vehicle's vision system. A diagram of the system is shown below in Figure 18 [Marino et al].

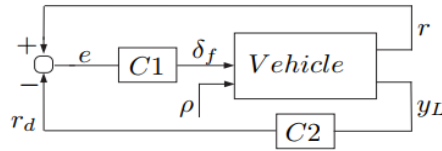


Figure 18. Diagram of nested PID controller design in LDAS [Marino et al].

The controller $C1$ is a PI controller, and the controller $C2$ is a PID controller. The measured yaw rate is represented by r , and the desired yaw rate is represented by r_d . The steering angle is input into the vehicle model as δ_f , and ρ represents the road curvature. The lateral offset is denoted with y_L . $C2$ takes in the lateral offset signal and generates the desired yaw rate. This is then sent to the controller $C1$, which attempts to minimize the error between the desired and output yaw rate. Section 4.3.3 evaluates this controller based on its performance in a number of simulations [Marino et al].

In the controlled invariance approach, the vehicle gathers data from its camera and vehicle dynamics sensors in order to gain an understanding of the current state of the vehicle. It then collects all the possible vehicle states that have a steering input that would successfully keep the vehicle in the lane—this is called the safe set. The controller then analyzes the vehicle state to see if it falls within this safe set. To do this, the model must assume that the vehicle is always travelling in a straight line. It collects data on the vehicle's longitudinal and lateral speed, yaw rate, and heading angle, as well as the vehicle's distance from the right and left lane lines. The inputs to the system are the wheel torque, τ_w , and the steering input angle, δ_f .

The task of the system is to be able to find a feedback strategy such that x is within the safety set. This decision process is shown below in Figure 19 [Hoehener et al].

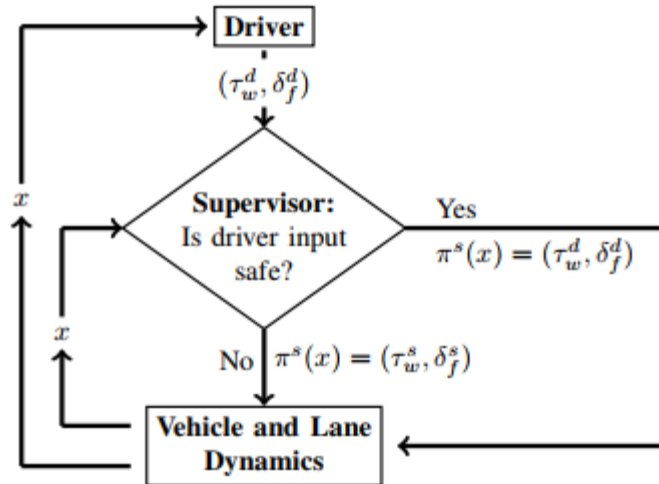


Figure 19. Diagram of the interaction between the safety supervisor and the driver. The driver's inputs are wheel torque, τ_w^d , and steering angle, δ_f^d . The supervisor's feedback control is represented by π^s [Hoehener et al].

The actions of the safety supervisor are accomplished in a set of three tasks. The first task occurs when the driver activates the LDAS. At this moment, an initial check is performed on the system. The purpose of this check is to ensure that the driver is not performing any extreme maneuvers at the point of initialization. If this check is successful, the system is activated and maximum and minimum values for vehicle speed and steering inputs are set. The next task is a status update. In this task the system monitors the vehicle speed and the steering inputs. If at any point the driver inputs a steering angle or vehicle velocity outside of the limits, the LDAS is shut off. The third component is the safety supervisor, which continuously monitors the vehicle's state and determines if steering input is required in the case of an imminent lane departure [Hoehener et al].

The safety supervisor acts by first defining the range of operation of the system. It is then able to separate the tasks of keeping the vehicle to the left of the right lane, and to the right of the left lane. It separates these states in order to define the separate states for the necessary steering trajectories. The vehicle state and steering trajectories are sampled at every time step. But since the LDAS needs to be continuously monitoring the vehicle's position in the lane, it is necessary to ensure that at the end of each time step there is a steering input that can maintain the vehicle's position in the lane. The safety supervisor acts after satisfying all three of the above conditions. It then uses a forward Euler approximation to compute the state that would result from the current driver inputs, and then performs integration to check whether it is in the safety set. The results from this method are presented in section 4.3.3 [Hoehener et al].

The third method that was studied was state feedback control. In the example shown here, the vehicle states are defined from data obtained from the sensors. The state feedback controller then aims to minimize error in certain key parameters. In this example the camera sensor is used to detect the location of the lane line and define it relative to the vehicle's position. The vehicle's

position relative to the lane line is used to determine the lateral offset. The vehicle's onboard Inertial Measurement Unit (IMU) is then used to find the vehicle's current yaw angle and yaw rate. The controller then minimizes the error in the lateral offset and the yaw angle. A diagram of the interactions between the sensors and controller is shown below in Figure 20 [Son et al].

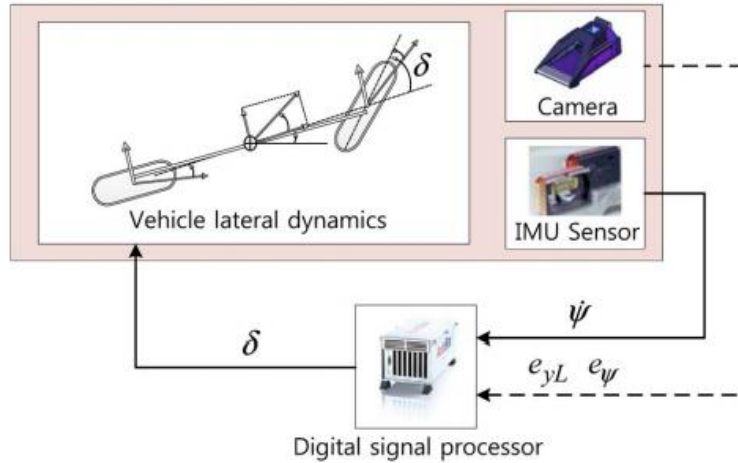


Figure 20. Diagram of interaction between vehicle sensors [Son et al]

The measured parameters from the IMU are then used to determine a parabolic trajectory of the vehicle. In order to do this, the model assumes a constant speed and yaw rate of the vehicle. In addition, two other polynomial trajectories are generated that represent the vehicle's relation to the left and right lane lines. The desired trajectory of the vehicle is then obtained by averaging the two polynomials that relate to the left and right lane lines, such that the desired trajectory is in the center [Son et al].

The state feedback controller aims to minimize the error between the desired yaw angle and lateral offset, and that which actually occurs. A look-ahead distance is selected that determines how far in front of the vehicle's path the camera sensor data will be sampled. This is an important parameter to select precisely as the look-ahead distance must be within the performance umbrella of the camera while simultaneously ensuring that the controller is stable at that sampling distance. A look-ahead distance that isn't far enough in front of the vehicle will influence the root locus of the controller and cause it to be unstable. The state feedback is designed as a proportional controller, as shown below

$$u(t) = -K_c e(t),$$

where $e(t)$ represents the error function. The optimal proportional gain, K_c , is selected using a Linear Quadratic Regulator that minimizes a cost function in order to minimize the error between the desired and actual yaw angle and offset [Son et al].

The performance of this controller in an experimental setting is described in section 4.3.3.

2.6.4 Stability Control

There are many different types of stability control. The two most common types are Active Steering (AS) systems and Direct Yaw moment Control (DYC) systems. In AS systems, the controller actively alters the driver's steering input to the vehicle. This is in an attempt to control the slip angle at the wheels—if the driver is performing a steering maneuver that increases the instability of the vehicle, the AS system steps in to try to correct it. AS systems come in a number of different configurations. Active Front Steering (AFS) is only applied to the front wheels and is good at increasing stability at high speeds. Active Rear Steering (ARS) is effective at improving a vehicle's performance in low speed and tight cornering maneuvers. Four Wheel Active Steering (4WAS) combines the benefits of both, such that the vehicle is stable at low and high speeds. Slip angle control with AS, however, is only half of the equation. AS helps to prevent the tires slipping, but once they are slipping, it can't do much to improve vehicle stability. This is where DYC comes in. DYC is effective in the linear and nonlinear regions of the tire—that is, it improves stability even when the tires have lost grip and are sliding. DYC systems come in the form of active braking, where the system selectively applies brake pressure, and torque distribution, where the system varies the torque output to the wheels. Torque distribution DYC is especially effective in electric vehicles in which a single motor powers one wheel. The vehicle platform for this project currently has a setup where four electric motors power the four wheels. In setups such as these, it is easier to customize the torque output to each wheel [Mousavinejad et al].

The first stability controller that will be studied here is a DYC torque distribution controller applied on a rear-wheel drive electric vehicle with two motors—one for the left rear wheel and one for the right rear wheel. While this configuration is slightly different than that of the current vehicle platform for this project, the principles should be applicable between the two.

This stability controller has a two-part control strategy: the first part is an Acceleration Slip Regulation (ASR) controller, which uses fuzzy logic in order to minimize slip at the wheels. The second part is the dynamic torque distribution controller, which regulates torque output to either of the rear wheels to improve vehicle stability.

The vehicle model was used to determine the sideslip and yaw rate of the vehicle given its speed and steering input. This data was then processed in each of the controllers. An overview of the control system is shown in Figure 21 [Zhang et al].

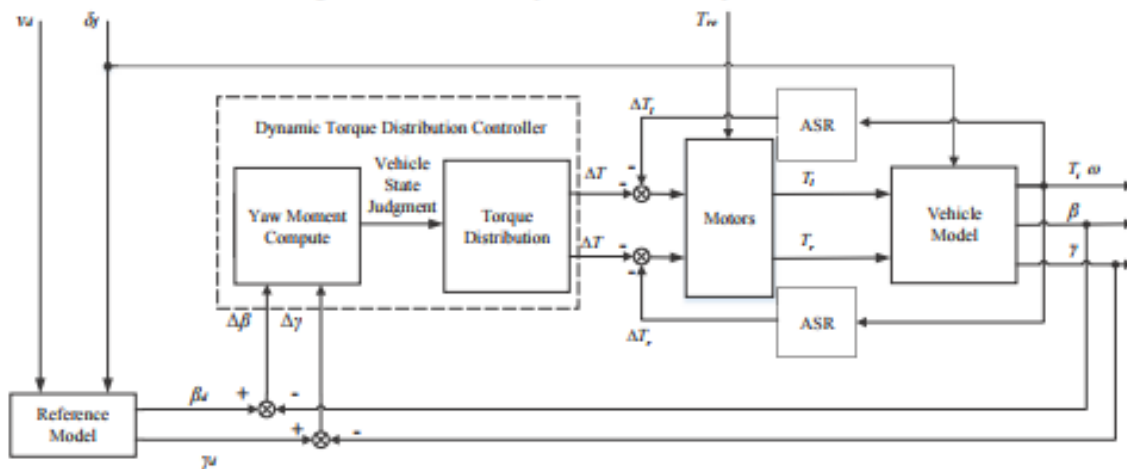


Figure 21. Overview of control system that includes one Acceleration Slip Regulation (ASR) controller per wheel, and a Dynamic Torque Distribution Controller [Zhang et al].

Note that there are two ASR controllers—this is because there is one ASR controller for each wheel. The ASR controller has inputs of angular velocity of the wheel, ω , and reference slip ratio, λ . It uses fuzzy logic to output the desired torque to the wheel [Zhang et al].

The dynamic torque distribution controller, on the other hand, stabilizes the vehicle by regulating its yaw moment, or the vehicle's tendency to rotate about its central axis. The controller has two parts: The first part uses fuzzy logic to find the desired yaw moment from the sideslip angle error and yaw rate error, and the second part uses rules to determine the output torque to each wheel, given the desired yaw moment. The torque distribution rules are shown in Table 2 below. The performance of this controller is examined in section 4.3.4.

Table 2. Torque distribution rules of dynamic torque distribution controller

Desired Yaw Rate	Desired Yaw Moment	Vehicle Condition	Torque Distribution
Positive	Positive	Left turn understeer	Decrease torque of left rear wheel
Positive	Negative	Left turn oversteer	Decrease torque of right rear wheel
Negative	Negative	Right turn understeer	Decrease torque of right rear wheel
Negative	Positive	Right turn oversteer	Decrease torque of left rear wheel

The next controller that will be studied is a simple PID controller that performs *DYC* torque distribution control. This was also studied in an application with independent wheel drive of an electric vehicle. The purpose of the PID controller is to control the yaw moment of the vehicle. The vehicle dynamics and vehicle model are used to find the yaw moment of the vehicle from the driver inputs such as steering angle and vehicle velocity. The yaw moment is then used to apply a torque to each wheel, in order to preserve stability [Wang et al].

This is accomplished using a tiered controller. The higher level controller computes the yaw moment from the vehicle dynamics. This information is fed to the lower level controller, which contains the PID control that regulates the yaw moment. In this study, performance was compared with lower level control with a single PID controller and lower level control with two PID controllers. Their relative performance is discussed in section 4.3.4. Figure 22 below demonstrates the process flow diagram of this controller [Want et al].

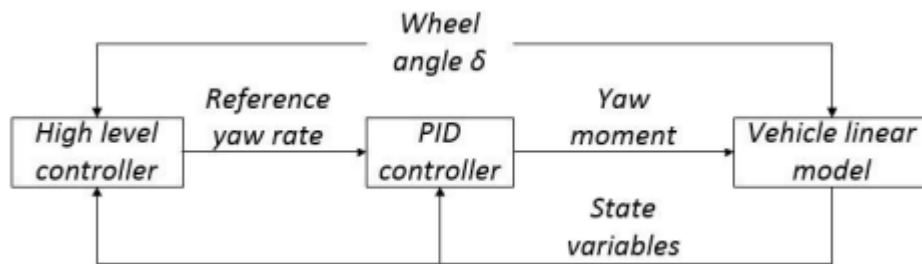


Figure 22. Process flow diagram for the tiered PID stability controller [Want et al].

Here it is shown that the higher level controller takes in the wheel angle and other state variables and generates the reference yaw rate, which the PID controller translates into a yaw moment that is then fed back into the vehicle model to generate a desired wheel torque [Wang et al]. The next few paragraphs will discuss an optimal torque distribution *DYC* controller.

The optimal controller studied in this section performs torque distribution *DYC* control on an Electric Vehicle (EV). It creates a disturbance feed-forward signal based upon the input steering angle, and outputs the yaw rate and lateral velocity through state feedback. The goal of the control system is to minimize the error in the actual and desired yaw rate. The yaw rate is found from measurements of the vehicle speed and steering angle input. This minimized yaw rate error is called the performance index. The performance index is also constrained by the physical limits of the vehicle such that it doesn't tell the system to apply impossible amounts of torque [Esmailzadeh].

The feedback and feed-forward gains of the system are found through a Linear Quadratic Regulator (LQR). These gains vary with the vehicle speed, as well as a weighting factor that is determined from the limits of the yaw moment. The weighting factor ensures that the yaw moment that the controller demands from the vehicle is less than the maximum possible yaw moment that can be applied. This controller was tested in a constant-velocity simulation on

different road surfaces. The results of the simulation are discussed in section 4.3.4 [Esmailzadeh].

This section discusses the various control strategies for stability control that were considered in the design of the stability control scheme for this vehicle platform. The design process for the stability controller is discussed in section 4.3.4

3 Objectives

The primary goal of this project is to create a customized semi-autonomous vehicle platform for use in a course setting. The existing vehicle platform, modified from an RC car, will be improved upon, and functionality will be added to it that will allow it to run control algorithms, behave predictably in an experimental setting, collect and process data in such a way that it can be used as an instructional tool, and come equipped with enough sensors such that students can have the freedom to expand upon the existing capabilities of the vehicle. Additionally, this vehicle will be designed for use by students without a mechatronics background.

This vehicle platform will also be a tool that can be used by students of the course and student researchers for many years, so it is also a goal to develop documentation that is thorough enough that the users will be able to take advantage of its capabilities. For students of the intelligent vehicle design course, user manuals will be developed so they can understand how to use the vehicle platform for the purposes of the course. Technical manuals will also be provided that can be used by student researchers to understand the design process and decisions, so the vehicle platform can be replicated and updated in the future.

3.1 Customer Needs

To better understand our customers' needs and how best to satisfy them, we developed a Quality Function Deployment, or QFD. In our QFD, we organized the customer needs, the engineering specifications, the tests to evaluate our design, and the performance of competitors' designs. Our main customer is Dr. Birdsong. He will be instructing a course on the control of intelligent vehicles, for which our product will serve as the learning tool. In addition to Dr. Birdsong, our customers also include the students of Dr. Birdsong's course, students researching intelligent vehicles at Cal Poly, and other universities wanting to teach courses on intelligent vehicle. These customers were included to ensure that we consider the specific needs of all potential users of the vehicle platform. Specifically, other universities were included after Dr. Birdsong discussed his desire to share the finished product with universities looking to offer courses relating to intelligent vehicles. Although Dr. Birdsong is the primary customer, students of his course will be the primary users of the product we put together. Apart from students of the course, we decided to add student researchers at Cal Poly as a customer, as intelligent vehicles have been a topic of interest in recent years in theses and senior projects. The needs of Dr. Birdsong were derived from our discussions with him. The needs of the students of the future course, Cal Poly graduate researchers, and other Universities, however, were determined by considering what sorts of features of the vehicle platform they would most benefit from. The Figure 24 shows the different needs of our four customers.

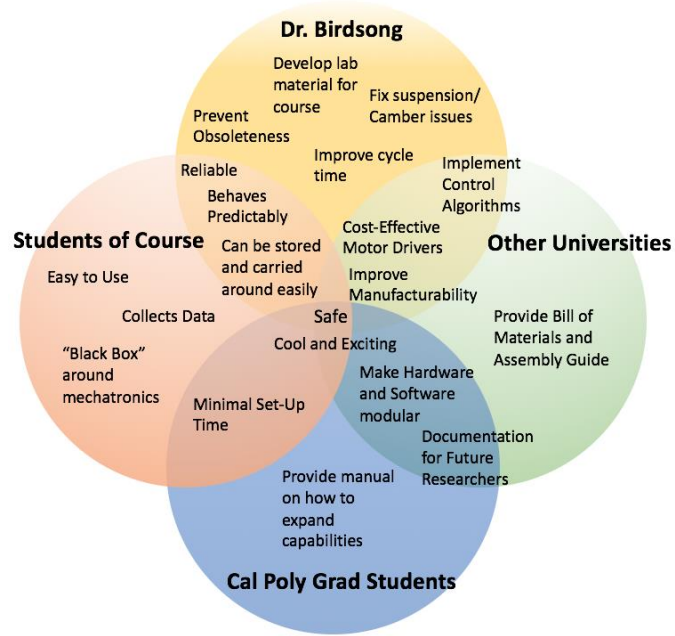


Figure 24. Customer Needs Diagram

3.2 Engineering Specifications

After coming up with the customer needs, the next step was to determine specifications that would address each of the needs. The specifications that were chosen can be found in Table 3 below.

Table 3. Engineering Specification Table

Spec #	Parameter	Target [units]	Tolerance	Risk	Compliance
1	Measured Camber Angle	0 [degrees]	+/- 2	L	T, I
2	Simulink Cycle Frequency	60 [Hz]	+/- 10	M	T
3	Signal Communication	1 [% packet loss]	+/- 0.5	M	T
4	Cost	\$1,500	+/- 250	M	A
5	Hours to Manufacture	8	+/- 2	M	T
6	# of Manufacturing Processes	3	+/- 1	M	T
7	Top Speed of Vehicle	6.7 [m/s]	+/- 1	L	T
8	Maximum Acceleration	1.4 [m/s ²]	+/- 2	M	T, A
9	Rollover at Top Speed	None	---	M	T, A
10	Works with Simulink	Meet	---	H	I
11	Protected Electronics	Meet	---	M	T, I
12	Reparability	All parts able to be replaced individually	---	M	I
13	Ease of Use	Yes	---	H	T
14	Battery Life	3 [hrs]	Min	L	T, A
15	Works with image processing	Meet	---	M	T
16	Integrates with LiDAR sensor	Meet	---	M	T
17	Integrates with stereo camera	Meet	---	M	T
18	Chaser car for ACC	Meet	---	M	T, I

1 – Measured Camber Angle: In order for the vehicle platform to behave predictably in an experimental setting, it needs to be mechanically sound. The way the suspension on the current vehicle is set up, vertical compression of the suspension changes the camber angle, or the angle that the vertical axis of the wheels makes with the ground. This kind of suspension travel may affect the car’s performance, especially at high speeds or in cornering. To ensure that the vehicle

model and the vehicle controllers can accurately affect the vehicle's behavior, this kind of suspension travel should be minimized.

2 – Simulink Cycle Frequency: The cycle time requirement refers to the time through one iteration of the Simulink control loop running on the Raspberry Pi. The controller outputs and actuations calculated from the current state of the system will exist discretely at a given frequency if Simulink is running at fixed time-steps. If the frequency is too low, the response of the controller to the dynamics of the system will be too slow to provide adequate control.

3 – Signal Communication: The signal communication requirement ties in strongly with the need for fault proof firmware on the Teensy microcontroller. Messages between components of the system architecture need to be transmitted with a high enough success rate to prevent errors, loss of dynamic response, and to help ensure complete test data. Percent packet loss refers to the percentage of the total messages sent that reached their destination.

4 – Cost: A future goal of this vehicle platform is that it can be shared with other universities for use in similar undergraduate courses in intelligent vehicles. To make this product attractive to other universities, the cost of the components should be minimized. The goal cost of \$1500 was generated from looking at the prices of other benchmarked vehicles that could accomplish the same tasks as our vehicle.

5 – Hours to Manufacture, 6 – Number of Manufacturing Processes: There are currently five manufacturing processes needed to construct the vehicle. The vehicle is composed of aluminum shaft couplers that have been fabricated using a CNC, a nylon chassis that has been laser cut, aluminum suspension mounts that have been cut using a water jet, and motor housing and bumper supports that have been 3D printed. Additionally, some holes and slots were made on the chassis using a mill. Below, Table 4 shows the different process needed for the components of the vehicle.

Table 4. Current and goal manufacturing processes for each manufactured vehicle component.

Vehicle Component	Material	Current Manufacturing Process	Goal Manufacturing Process
Shaft couplers	Aluminum	CNC	3D Print
Motor housing	PLA	3D Print	<i>No change</i>
Steering posts	Aluminum	Lathe	Purchase Part
Turnbuckle mounts	Aluminum	Mill	3D Print
A-Arm mounts	Aluminum	Mill	<i>No change</i>
Chassis Mounts	Aluminum	Water Jet & Sheet Metal Bending Brake	Water Jet
Front and Rear suspension mounts	Aluminum	Water Jet	<i>No change</i>
Chassis	Nylon	Water Jet	<i>No change</i>
Sensor Array	PLA	3D Print	<i>No change</i>
Steering Linkage	PLA	3D Print	<i>No change</i>

7 – Top Speed of Vehicle: The top speed of the vehicle was set with the goal for the vehicle to have a wide range of performance, while still having vehicle dynamics that are realistic to full-scale vehicles. For a 1/7th scale vehicle, a top speed of 6.7 m/s corresponds to a full-scale top speed of 46.9 m/s, or 104.9 mph. This is close to the top speed of a compact, low-horsepower vehicle.

8 – Maximum Acceleration: The maximum acceleration specification of the vehicle was also set to mimic the behavior of a full-scale vehicle. A maximum acceleration of a 1/7th scale vehicle of 1.4 m/s² corresponds to a full-scale acceleration of 9.8 m/s², or 1g of acceleration. This is the acceleration achieved by a Tesla Model S P85D.

9 – Rollover at Top Speed: Since the students will in some ways be designing the vehicle’s behavior, it is to be expected that some of the vehicle commands will go awry in that process. While the exact behavior of the students is impossible to predict, the vehicle platform should be able to withstand some amount of unpredictable programming. One of the ways in which to mitigate damage to the vehicle in these situations is by minimizing the vehicle’s capacity to roll when given a command to turn at speed. The vehicle should be able to stay upright when given a maximum steering input at top speed.

10 – Works With Simulink: The platform’s ability to run control algorithms designed in Simulink will be a continued requirement. It is essential to the success of the platform that this

ability does not go away as students without a mechatronics background should be able to design and implement control algorithms on the vehicle. Currently the vehicle is able to run in either “Deployed Mode”, where it runs independently of a PC, or “External Mode” where it connects through WiFi to a personal computer and gives live feedback.

11 – Protected Electronics, 12 – Reparability: It is important that the vehicle remain operable throughout the quarter and over the years. The vehicle is likely to crash or be mishandled at some point during its usage. Avoiding crashing and mishandling is difficult given the nature of the course. However, a lot can be done to reduce the likelihood that an accident put the car out of commission. We have addressed this possibility with three specs: reparability, life cycle, and protected electronics. In the case that the vehicle is damaged, we want the vehicle to be easy to repair. This will be achieved by making sure that all the custom mechanical components can be individually swapped out. Additionally, there will also be a repair guide that will describe assembly of the vehicle and common repairs that may be needed. The life cycle specification will address how the vehicle is designed to prevent catastrophic damage. The vehicle should contain some sort of bumper in the front, rear, and sides, that will prevent any of the electronics and mechanical components from being damaged. With regards to the protected electronics specification, all of the electrical components and wiring should be securely fastened and housed or otherwise protected. The electronic wiring should be protected in such a way that connections do not short, and the wiring does not get caught on anything and get disconnected.

13 – Ease of Use: Students and researchers should be focused on learning and developing control algorithms and a long, tedious setup procedure will get in the way of this. Users should be able to seamlessly transition from controller design and simulation to implementation and testing on the vehicle. A well thought out user interface will likely be a good portion of the total design work. This specification will be tested using a survey of mechanical engineering students that will be conducted once adequate capability and procedural documentation have been produced. The goal of the survey will be to determine whether a new user of the vehicle will be able to get a control algorithm from a completed Simulink design on the computer to running within 10 minutes using a one-page guide.

14 – Battery Life: The battery life of the vehicle platform must be able to last through an entire three-hour lab session while in use, without needing to be recharged. Since it is likely that the vehicle platform will never be ran for three hours straight, if the vehicle platform meets this specification its performance will be adequate for use in the lab.

15 – Works With Image Processing: One of the topics of Dr. Birdsong’s future course will be introducing sensors that are currently being used on intelligent vehicles. In order that the students get a better familiarity with the sensors that will be introduced in the class, it would be beneficial if they gain exposure integrating the sensors on the vehicle platform during the lab.

16 – Works With LiDAR sensor: Dr. Birdsong’s plan is to offer a LiDAR sensor and a Stereo Camera sensor to two lab teams. It would be expensive to equip each lab vehicle with a LiDAR and stereo Camera sensor, so at least initially, there will be one LiDAR sensor and one Stereo Camera sensor so that they can at least be used for a lab project. Much investigation will need to be done in order to understand how these two sensors with large data outputs can be integrated

into the architecture. It is possible that an additional microcontroller or perhaps an NVIDIA Jetson may be needed.

17 – Integrates with Stereo Camera: One of the primary sensors will be the camera. The camera will be used to get feedback on lane lines so that the car has the functionality to perform lane keeping. The lane detection algorithm will be run on the Raspberry Pi using Simulink. This will allow students the opportunity to adjust the algorithm or perhaps attempt their own.

18 – Chaser Car for ACC: Adaptive Cruise Control (ACC) works by controlling the vehicle's speed relative to another vehicle in front of it. To adequately test the vehicle platform's ACC performance, a "chaser car" needs to be developed that is of the necessary size and shape and can serve as a speed-regulation test for the ACC algorithm. This "chaser car" would be controlled manually while the ACC vehicle follows it, in order to determine how well the ACC vehicle performs speed regulation.

These engineering requirements were developed from the list of customer requirements that we developed. We will use these engineering specifications as measures of how well we have met the customer requirements. These were developed using the Quality Function Deployment method, or QFD. In the QFD method, each customer is listed separately, and a list of their requirements for the project is developed. These requirements are then weighted, taking the viewpoint of each customer, and a list of engineering specifications is generated such that each requirement lines up with a means of measuring its success. Then a benchmarking study is done to ensure that there are no existing products that meet the current customer requirements. The QFD spreadsheet for this project can be found in Appendix B.

3.3 Boundary Sketch and Boundary Diagram

Since last year's senior project team has already developed a vehicle platform, most of what we will be changing will be on the software side, with some on the hardware side. The boundary sketch in Figure 25 below shows a student interacting with our device in a classroom setting. The dotted line draws a boundary around what lies within the scope of this project for us to change.



Figure 25. This boundary sketch shows everything that it is within the scope of the project to change. This project will alter the vehicle hardware, software, and develop literature that a student can use in the intelligent vehicle design course.

The scope of this project extends to altering the vehicle software (represented by the Simulink file on the laptop), the vehicle hardware (represented by the mess of wires on top of the vehicle), the students understanding of the vehicle and learning experience, and the literature that is available to them to learn from (represented by the open book).

In addition to the Boundary Sketch, a Boundary Diagram is included in Figure 26 that gives more detail about the customer needs that our work will address. Several of the items originally selected to be within the scope of the project have changed as time progressed. Instead of implementing cost effective motors, we have decided to recommend a cost effective alternative and leave their implementation up to future iterations so that we can focus on providing lab functionality, user interface, etc. We have also decided to remove the shaft coupler design from our scope.

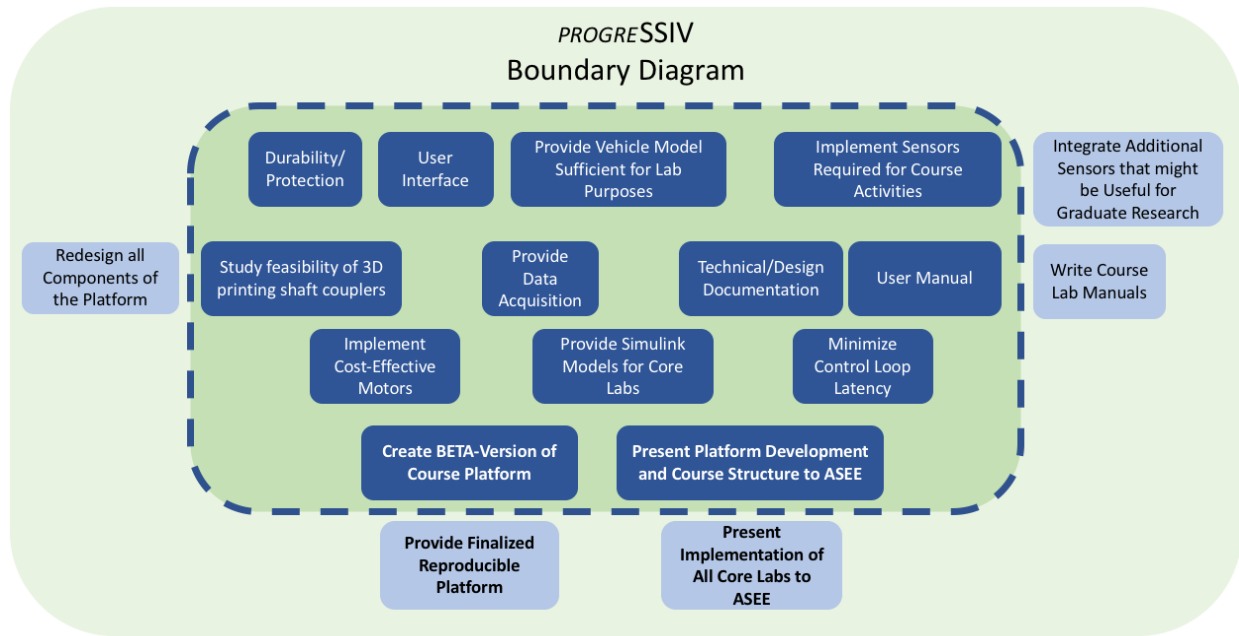


Figure 26. This Boundary Diagram explicitly shows what needs we will be addressing in our senior project and what needs we will not be fulfilling.

The next section discusses the use of a functional decomposition diagram to relate functions (derived from the above information and sponsor correspondence) to specific attributes of the platform and tasks to be completed.

3.4 Functional Decompositions Diagram

The customer needs, engineering specifications, and boundary sketch and boundary diagram guided the creation of various vehicle functions that were formed into a functional decomposition diagram (Appendix C). These functions included three related to the big picture goals of the project - two being subtasks of the most important:

1. Serves as instructional platform for intelligent vehicle controls.
 - a. Provides functions required by learning objectives of Dr. Birdsong's course.
 - b. Serves as more general research platform for researchers and other universities.

These were linked to a set of primary functions that encompass most elements of the boundary diagram. These include:

1. Perform the four core labs envisioned for the course.
2. Provides system identification capabilities.
3. User interface facilitates operation of the vehicle in a manner that is user friendly and conducive to lab progress and structure.
4. Provides organized test data to users.
5. Documentation instructs unskilled students in vehicle operation.
6. Technical manual informs researchers how to expand the platform.
7. Embeds control algorithms written in various languages on its hardware.

These primary functions cascade down into secondary functions, tasks, and attributes of the platform such as software and firmware functions, vehicle modeling, sensor selection and

mounting, and others. Many of the primary functions as well as some crucial secondary functions served as topics of ideation and concept development. The functional decomposition diagram provides the content for idea generation, serves as a road map linking the specifics to the big picture, and when fully completed may serve as a complete checklist of tasks to be completed before the vehicle project can be considered a success.

4. Concept Design Development

The above research, requirements, scope, and specifications were used to guide team and individual ideation sessions and decision matrices. A functional decomposition diagram (see Appendix C) was first developed whose functions would serve as topics for ideation. The topics included mechanical improvements, sensor integration, controller designs, software and firmware architecture, and user interface. Once ideation was done, the feasible solutions for each of these functions (not including software and firmware architecture for reasons that will be discussed in that section) were narrowed down through the Pugh matrix method to several top ideas. These were not integrated morphologically into an overall system concept for the platform, however, each of the functions could be decided upon independently of the others - a byproduct of both the modularity of the original uLaren design and our desire to further test the effectiveness of that modularity. This process is outlined in more detail in the sections below.

4.1 Mechanical Concept Development

This section discusses the concept development process that led to the selection of a final design for the mechanical system. This process began by investigating all the mechanical functions required by the vehicle in order to satisfy the needs of the customers (see Figure 24). From this investigation, we were able to come up with two primary functions for which we would brainstorm design solutions. These two functions included: protection of the electrical components and mounting and housing of the sensors. Appendix D shows sketches of the different solutions that were generated from the ideation process. These solutions included: a roll cage, a fiberglass shell, sensors that slide on rails (to adjust the positioning), modular housing (protective cover for each individual component), and a double decker (elevated platform on top of chassis to mount additional sensors). The criteria used to evaluate these designs can be found in Table 5 below.

Table 5. Criteria used to evaluate the mechanical concept designs.

Criteria	Weight
Electronics securely mounted	5
Protection from outside	5
Wiring not exposed	5
Easy to manufacture	5
Sensors have good range	4
Durable	4
Sensors can be adjusted/mounted	3
Easy to add/fit new electronics	3
Cost effective	3
Cool	2

A Pugh Matrix, shown in Appendix E, was constructed to determine the best design solution for the project based on the constraints. It was determined that the fiberglass shell and the double-decker platform would best address the needs of our customer. The roll cage solution was also strongly considered; however, it was decided that the manufacturing would be too complicated. Apart from the manufacturing, having the shell instead of the roll cage would serve as a good target for when the adaptive cruise control lab would be implemented. Specifically, the time of flight sensor that will be used for the distance tracking would have a large surface area that it could detect. The concept of adaptive cruise control as it applied to our vehicle will be explained in section 4.3.2.

Once the shell and double-decker platform were established as the mechanical design solutions, concept models were built to verify the concept. These concepts can be seen in Figure 27 below.

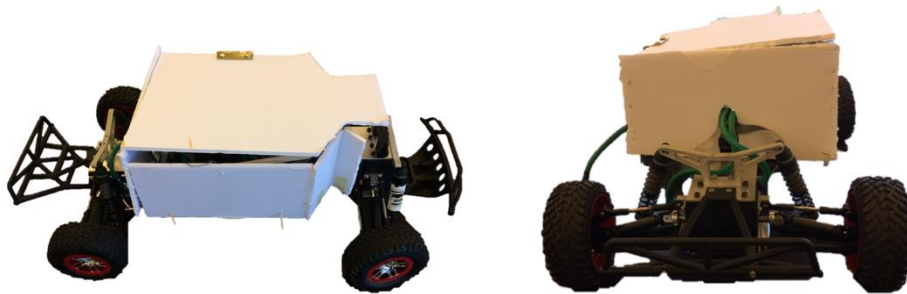


Figure 27. Shell concept developments. The perspective on the right shows the large target area on the rear of the car.

After the concept models were made, more elaborate prototypes were designed and built to better visualize how the mechanical additions would fit in with the current hardware. Figure 28 below shows both the double-decker and the shell prototypes.

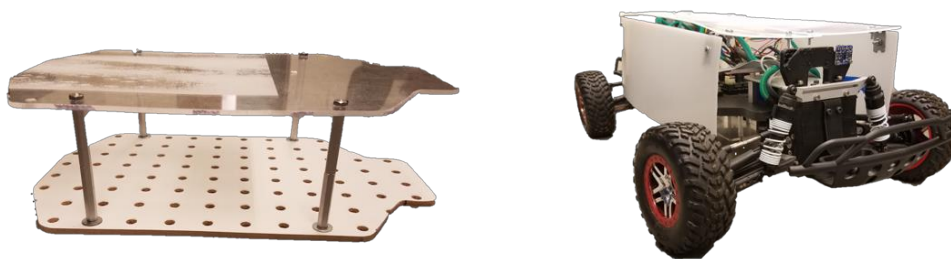


Figure 28. Double-decker and shell build prototypes.

The last step in the initial design process was to develop a CAD render to begin to put more detail into the dimensions and the overall appearance of the designs. The results of these

renderings can be seen in Figures 29 and 30 below. Please note that the CAD of the vehicle, that is, everything that was done except for the top platform, the LiDAR sensor, and the shell was created by the uLaren team [Miley, Phillips, Grant].

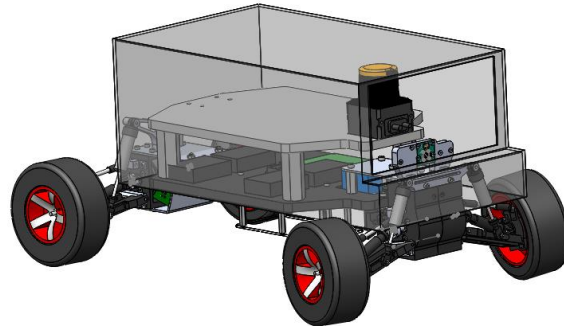


Figure 29. CAD rendering of the fiber glass shell concept. The shell would protect the internal components. The shell would also act as a good “target” so that other cars would have a large target to detect with their time of flight sensors.

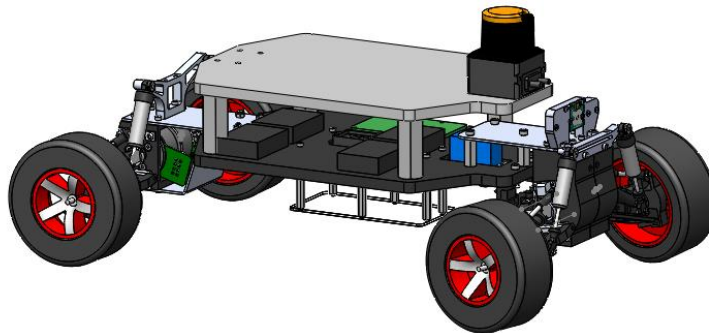


Figure 30. CAD rendering of the double-decker concept. A LiDAR sensor is included to show how the top platform would allow for additional space for sensor and electrical component placement.

The two CAD models represent how well the current design will benefit from the mechanical additions. Specifically, the double-decker platform will allow for additional space to place sensors. It will also serve to create a more “black-box” environment around the electronics, in that the wires and controllers would be hidden beneath the top platform. We could also construct acrylic sides where the rectangular openings currently are, in order to completely remove access to the controllers and wires, with the exception of some ports such as the USB connection port. The platform could also serve as a means of changing the location of the center of gravity—it could be set up with a grid that students could attach weights to at different locations. They could

then study how changing the location of the center of gravity affected the vehicle’s performance in different driving scenarios.

The shell will benefit the current system in that it will keep debris out of the inside and ensure that the wires do not get caught on anything in the environment. The shell would also serve as a last barrier in the case of a crash. As mentioned before, one of the motivations for the shell was so that any car affixed with a shell could serve as a chaser car. This is a similar concept to a senior project sponsored by Daimler that was worked on during the 2016-2017 school year. In this project, a large target was constructed and placed over a platform so that the Daimler truck could test out its object detection on the target. The target for the truck needed to be large in order to be comparable in size to a car that the truck may will encounter on the road. Similarly, a chaser-car would need to have a large target so that the car that is following behind it has a large target area to detect.

The two design solutions that were generated from the concept development do not satisfy all of the mechanical needs of the customers. The needs not addressed by these concepts are improving manufacturability and reducing the cost.

To address the need to improve manufacturability, we intend to test out which mechanical components can be properly manufactured using 3D printing. Below, Table 6 shows the different process needed for the components of the vehicle.

Table 6. Current and goal manufacturing processes for each manufactured vehicle component.

Vehicle Component	Material	Current Manufacturing Process	Goal Manufacturing Process
Shaft couplers	Aluminum	CNC	3D Print
Motor housing	PLA	3D Print	<i>No change</i>
Steering posts	Aluminum	Lathe	Purchase Part
Turnbuckle mounts	Aluminum	Mill	3D Print
A-Arm mounts	Aluminum	Mill	<i>No change</i>
Chassis Mounts	Aluminum	Water Jet and Sheet Metal Bending Brake	Water Jet
Front and Rear suspension mounts	Aluminum	Water Jet	<i>No change</i>
Chassis	Nylon	Water Jet	<i>No change</i>
Sensor Array	PL	3D Print	<i>No change</i>
Steering Linkage	PLA	3D Print	<i>No change</i>

The highest priority component to switch over to 3D printing is the motor shaft coupler. This part is currently being manufactured through a CNC. The reasons that it would be desirable to switch the process from CNC to 3D printing is that the parts could be made quicker, cheaper, and without the need for a licensed CNC tech. Also, it may be a more compelling product, as seen by other universities, if the components could mostly be 3D printed, since not all universities have CNCs. The CAD model of the motor shaft coupler is displayed in Figure 31 below.

ITEM NO.	PART NUMBER	DESCRIPTION
1	225A	SHAFT COUPLER SIDE
2	225B	SHAFT COUPLER SIDE
3	231	M2.5 X 0.4 12MM SOC

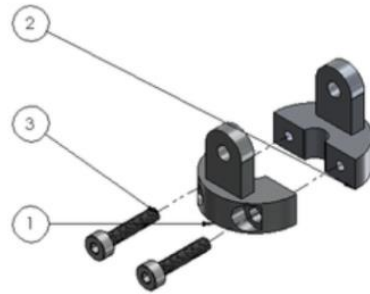


Figure 31. Current shaft coupler manufactured by CNC.
[Miley, Phillips, and Grant]

The last customer need that will be addressed through the mechanical improvements is reducing the cost. The most expensive components by far are the motor and motor drivers. Together, they cost around \$4800. The possibility of switching to lower-cost motors and motor drivers has been a topic of research. The motor options that have been considered are all Maxon Motors, just like the current motors. They also belong to the same class of motors: the “Flat EC” motors, which are compact brushless DC motors, as shown in Figure 32.

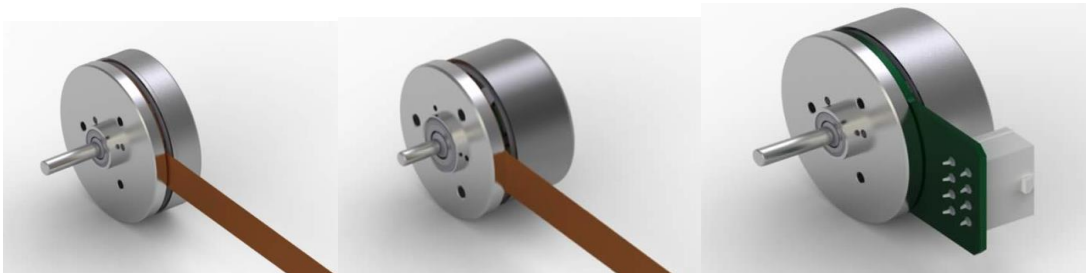


Figure 32. Maxon’s EC Flat class of motors.
(<https://www.maxonmotorusa.com/maxon/view/catalog/>)

Only compact brushless DC motors were considered in the motor selection because most other motors would be too large to mount back to back between the motors as seen in Figure 33 below.

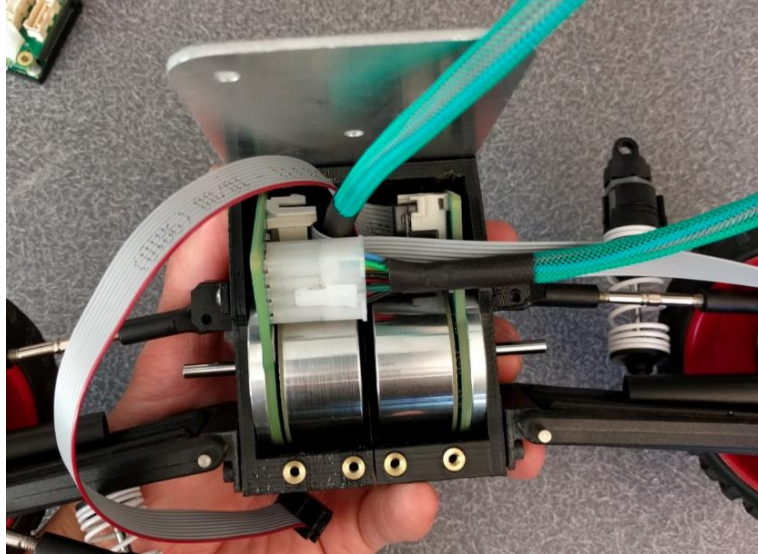


Figure 33. Motor configuration. [Miley, Phillips, and Grant]

Having the motors back to back between the wheels eliminates the need for gears or a belt drive, which would greatly increase the complexity of the mechanical system. Currently the Maxon motor chosen is a 70-Watt motor with both hall sensor and encoder feedback. It is believed that a less expensive Maxon motor with less power would still be adequate for the performance needs of the car. In addition to selecting a new motor, we intend to install a new motor driver. We are working with graduate student Charlie Refvem, who has agreed to design a motor driver for our senior project that would perform the functionality needed for the car. Assuming a motor is chosen with hall sensors or with an added encoder, Charlie's board would read the feedback and perform commutation to output the voltage signals that would yield the desired shaft position of the motor. Charlie's board is shown in Figure 34 below.

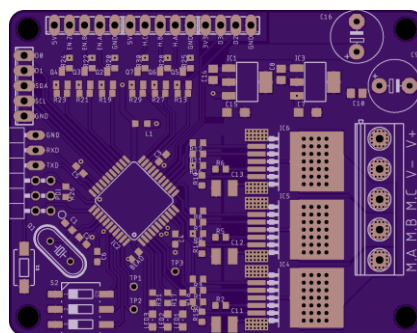


Figure 34. BLDC Motor Driver designed by Graduate Student Charlie Refvem.

Table 7 below shows how the cost would change if we chose one of the lower power Maxon motors and used Charlie's motor driver.

Table 7. Proposed Bill of Materials of Motors compared to Current

Component	Description	Price
<i>Current BLDC Motor</i>	<i>70W w/Hall & Enc.</i>	<i>\$400 X4</i>
Proposed BLDC Motors	50W w/ Hall	\$147 \$103 each (min. 20) x 4
	30W w/ Hall	\$100 \$74 each (min. 20) x 4
	30W w/o Hall	\$92 \$67 each (min. 20) x 4
	15W w/o Hall	\$82 \$61 each (min. 20) x 4
<i>Current Motor Driver</i>	<i>Maxon Motor Driver</i>	<i>\$800 x 4</i>
Proposed Motor Driver	Charlie's Motor Driver	\$55 x 4
Proposed Encoder	CUI Encoder	\$35 x 4
Total Cost per Car	\$948 > X > \$604	
<i>Current Cost per car</i>	<i>\$4800</i>	

The next steps for the mechanical improvements are to investigate the best material and method to manufacture the shell and double-decker. Initially, the shell was to be made out of fiberglass, but after more research was done it was determined that it would make more sense to make the shell out of plastic. There is a vacuum-forming device on campus that can be used to mold plastics into desired shapes. More investigation will be done to see if this will be a viable option. For the selection of motors, analysis will be done on the current vehicle to measure parameter, such as weight, tire diameter, max acceleration desired, and top speed desired. Some similitude calculations will need to be performed as well so that the performance of the small-scale car is equivalent to the performance of a full-size car.

4.2 Sensor Concept Development

One of the primary constraints on the functional capabilities of the car are the sensors that are being used. It is important to include enough sensors in the car so that it is capable of performing the functionality required of the core labs, as well as the functionality desired by the students in their lab projects.

The car currently has four sensors- the Inertial Measurement Unit (IMU), the Time of Flight sensor, the motor encoders, and the camera. Under this setup the car can perform lane detection, rough odometry, and limited object detection. The acceleration control lab requires accurate readings of acceleration. This can be accomplished with the IMU that is currently integrated into the car. The stability lab requires information about the slip angle (see section 2.1) of the vehicle. There has been a lot of research into using a GPS sensor along with an IMU sensor to yield the slip angle [Yoon]. The lane keeping lab requires data of the distance to the lanes. This can be estimated with a single camera when the assumption is made that the road is flat. The camera must be pre-calibrated so that the distances at the different pixel coordinates are known. The adaptive cruise control lab requires feedback of the distance to the chaser-car or object in front of it. This could be accomplished with the current Time of Flight sensor on the car, although it would require that the object that was being tracked had a large target for the sensor to detect, and additionally, it would require that the target be directly in the path of the sensor's detection cone. In other words, the sensor has very poor "peripheral vision".

Apart from the core labs, the car should be able to accomplish additional functionality that students may wish to achieve for their lab project. Some example projects may include: obstacle avoidance, navigation from point A to B, path planning, automatic parking, and vehicle to vehicle communication. Many of these projects and others not mentioned would require additional feedback that is not available from the sensors currently on the vehicle.

Although it is not possible to know in advance what projects students will choose during the course, research and analysis was done to decide upon additional sensors to add to the vehicle. Table 8 below gives a list of the different sensors that were looked at.

Table 8. Sensors investigated for usage on car

Sensor Function	Sensor	Details	Price
3D Scanning	Nerian Stereo Camera	Processes up to 30 FPS @ 640x480 resolution. Requires 2 USB connections (1 for each camera).	\$1,500
3D Scanning	Zed Stereo Camera	Processes up to 100 FPS @ 1344x376 resolution. Single USB 3.0 interface.	\$450
3D Scanning	DUO DDK Stereo Camera Kit	Processes up to 320 FPS @320x120 resolution. Includes embedded computer that processes video stream. WIFI and Ethernet connection.	\$995
2D Scanning	Hokuyo LiDAR	Can be interfaced with through USB or RS232 (2 pin serial connection).	\$2,000
Vehicle Position and Velocity	Indoor GPS/IMU	Kit that gives feedback on position in indoor environments with a resolution of +/- 2cm. SPI, I2C, and UART.	\$70
Vehicle Velocity	Optical Flow Camera	Any type of web camera or PI Camera can be programmed to solve Optical Flow. Horn-Schunck algorithm included in Simulink.	\$35
Motor Velocity	CUI Capacitive Encoder	Compact capacitive encoder that can be mounted to many motors different motor shafts.	\$35

Two main categories of sensors were looked at– depth sensors and motion and position sensors. The depth sensors themselves were broken up into stereo camera sensors and LiDAR sensors (refer to section 2.4). There stereo camera sensors considered were the Nerian Stereo Camera, the Zed Stereo Camera, and the Duo M Stereo Camera.

The Nerian Stereo Camera, shown in Figure 35, captures frames at up to 30 FPS at a resolution of 640x480. The camera sends the captured images in real time to the host computer through two USB connections (one for each of the two cameras). The Nerian Stereo Camera requires a host computer to process the images unless an embedded computer is purchased from them. The embedded computer is capable of calculating the image disparity and outputting the 3D distance data through Ethernet. The camera and computer system together would cost \$3400 (\$1500 for the camera and \$1900 for the embedded computer). It is not well known how this system would integrate into the current hardware and software architecture. The Raspberry Pi does contain an Ethernet port, however data being transmitted through the Ethernet port is not currently supported in Simulink and creating a driver for this would be an extensive task.



Figure 35. Nerian Stereo Camera [<https://nerian.com>]

The Zed Camera, shown in Figure 36, is able to process images at 100 FPS at a resolution of 1344x376. Similar to the Nerian Stereo Camera, it requires a host computer to take care of the processing in order to get the 3D map. Additionally, the host computer must be equipped with an NVIDIA graphical processing unit or GPU. There is a programming architecture known as CUDA (Computer Unified Device Architecture) that allows for highly parallelized computations and this architecture is supported by NVIDIA GPUs. This is the reason that the Zed has that requirement--in fact, some Matlab libraries require CUDA enabled hardware as well. Unlike the Nerian Stereo Camera, Stereo Labs (the makers of the Zed Camera) do not provide an embedded processor to take the place of a laptop or desktop. However, the Zed Camera is supported by the NVIDIA Jetson, which is a GPU that is widely used in embedded projects. MIT's smart RC car, RACECAR, uses a ZED camera running on an NVIDIA Jetson. One of the benefits of the Zed Camera is that the company has provided example code on how to access images and calculate image disparity using MATLAB. The company also provides a software package that simplifies tuning the camera and outputting the 3D data.

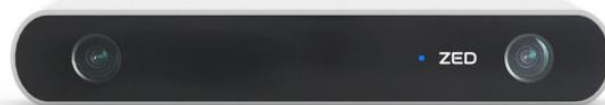


Figure 36. Zed Stereo Camera [<https://www.stereolabs.com>]

The DUO DDX stereo camera kit shown in Figure 37 includes a DUO MLX stereo camera and an embedded computer to process the video footage. The DUO MLX is capable of 320 FPS at a resolution of 320x120. The company, DUO3D, provides software that can be used to run the

stereo camera and determine the distance at each pixel. The embedded computer can be communicated with through Ethernet or WIFI.



Figure 37. DUO DDX Kit including the Stereo Camera and embedded computer
[<https://duo3d.com/product/duo-minilx-lv1>]

Aside from stereo cameras, 3D scanning can also be achieved by LiDAR sensors. The LiDAR sensor that was researched was the Hokuyo UBG-04LX-F01 LiDAR sensor. This sensor was used by Cal Poly graduate student Thomas Stevens in his thesis. The sensor can be communicated with through USB interface or RS-232 serial interface. The current hardware and software architecture on the vehicle is able to communicate through the serial interface, but because of the large amounts of data coming from the LiDAR, it remains to be found out if the current microcontroller, the Teensy, can store and process all the data.

The motion and position sensors that were looked at include: the indoor GPS sensor, the optical flow camera, and the CUI Capacitive Encoder. In contrast with the previous sections on sensors, the following sensors do not overlap in function as much. In fact, all of the following sensors could all be included on the car without too much redundant data.

The Indoor GPS sensor shown in Figure 38 is a positional sensor made by Marvel Mind Robotics. The system is composed of two stationary beacons: a beacon mounted on the moving object, and then a router that facilitates the communication between all of the beacons. Although this system uses a similar method of triangulation to GPS satellites, the system uses ultrasonic sound waves as opposed to the microwaves used by GPS. There is the option to purchase the beacons with an IMU built in. Both the GPS positional data and the IMU feedback can be accessed through UART, SPI, or I2C, allowing it to fit into the current architecture of our system. Marvel Mind claims that their system achieves +/- 2cm accuracy of positional data. This system is used in the BARC (UC Berkeley's smart RC car). [Gonzalez J.]



Figure 38. Ultrasonic GPS sensors. A starter kit can be purchased that includes five beacons and one router. [<https://marvelmind.com>]

Optical flow is a method of finding the velocity vector of an object by comparing two image stills of an object that has moved. The velocity vector is generated by comparing the position of the object in the first image to the position of the object in the next image. The difficulty arises in recognizing the moving object as a single object, in order to avoid influence on the velocity vector from other moving objects or even the stationary environment. There are a few different algorithms that perform optical flow. One of the algorithms, the Horn-Schunck algorithm, is supported as a Simulink block under the Computer Vision Toolbox. Conceivably this algorithm could be run using video stream from the Raspberry Pi camera or webcam. Most likely, the best results would arise when the camera is pointed downward towards the ground to minimize the noise from other moving objects. A diagram of how a tracking system with optical flow would work is shown in Figure 39.

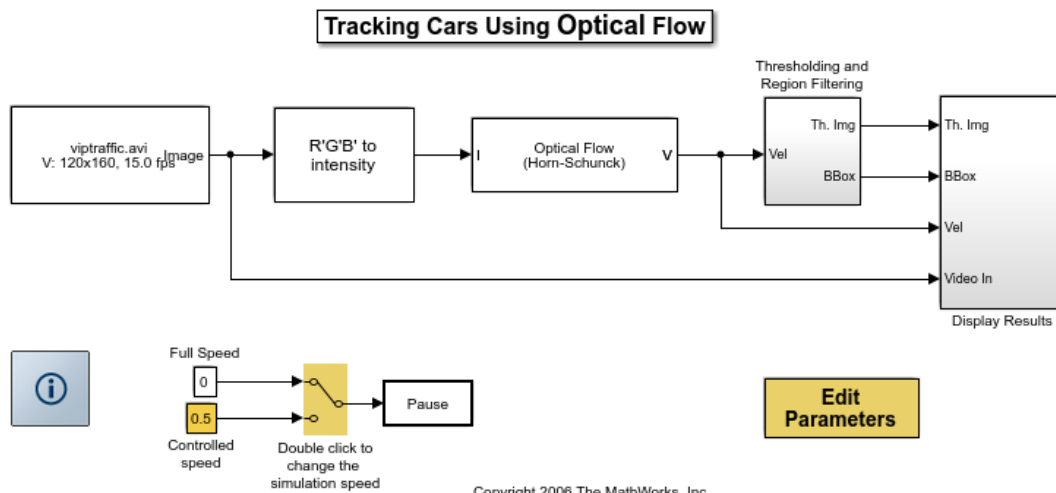


Figure 39. Simulink model centered around the Horn-Schunck Optical Flow Block [<https://www.mathworks.com/examples/simulink-computer-vision/mw/vision-ex14612482-tracking-cars-using-optical-flow#1>]

The last sensor that was investigated was the capacitive encoder made by CUI, shown in Figure 40. The current motors on the vehicle contain both encoders and hall sensors. This combination allows for elaborate spatial-vector commutation on the brushless DC motors. Given that one of the main objectives in the senior project is to reduce the cost of the platform, it may be necessary to select new motors that are sensor-less or just contain hall sensors. Without encoders, it is not possible to do spatial-vector commutation, and block commutation must be done instead for motor position control. Apart from allowing for precise position control of motors, encoders can also be used to determine the wheel speed of the vehicle. This particular encoder was chosen because it is small and can be adjusted to fit many different motor shaft sizes. The need for an encoder sensor really depends on what type of motor is selected. Although the complexity is increased, purchasing a sensor-less motor (no encoder or hall sensors built into it) and then purchasing encoders to mount onto the motor may be cheaper.



Figure 40. CUI Capacitive Encoder [<http://www.cui.com/product/motion/rotary-encoders/incremental/modular/amt11-v-kit>]

A Pugh Matrix (see Appendix F) was made to evaluate all of the sensors. Table 9 below shows the criteria that were chosen to evaluate the sensors.

Table 9. Criteria chosen for Sensor Selection

Criteria	Weight
Integrates well with current software/hardware	5
Sensors used in Industry (student gain valuable skill interfacing with sensor)	3
Cost	2
Reliable Data	4
Long term support for sensor	3
Compatible with MATLAB/Simulink	2
Easy for students to work with	1

It was very important that the sensor be compatible with our current system architecture which is why the criteria of integrating well with the current software/hardware has the largest weight. Because the future course will include topics on smart vehicles, we decided that it would be very beneficial to the students to work with sensors that are currently being used by automotive companies, that way they could develop skills that would prepare them to enter industry. Cost was another factor since about a dozen of each sensor would need to be purchased to outfit each vehicle.

After evaluating each sensor based on the criteria, it was determined that the indoor GPS sensor and the CUI encoder would be a great addition to the platform. The indoor GPS sensor would simplify the calculation of the slip angle. Additionally, it would allow for the possibilities of more student projects. For example, with the GPS sensor and a LiDAR sensor, the car could be programmed to go from one corner of the room to another corner while avoiding obstacles. Another project idea would be to have the car trace a path such as a circle using positional feedback from the indoor GPS sensor.

The CUI encoder would be crucial to the system if new sensor-less motors were selected. If motors with built in hall sensors were selected, then there would have to be some analysis and testing to determine how much improvement there is from using encoders. Although LiDAR and Stereo Cameras did not receive high results through the Pugh Matrix, they will still be considered. We currently have a LiDAR sensor in our possession, and so we plan on doing some testing to see how the LiDAR sensor integrates with the architecture. As for the Stereo Vision Camera, the best method for integration into the architecture seems to be through an NVIDIA Jetson. Given that in a couple years the architecture may switch to being powered by a Jetson, it may be a good idea to begin that transition now by using the Zed Stereo Camera board with the Jetson to get 3D data.

As for the Optical flow option, because we have access to the Raspberry Pi camera, we may spend some time working with the Simulink Optical Flow block to see if we are able to get any meaningful feedback from it. Something relating to optical flow could also have potential for a student project.

The next step for the sensor development is to finalize the decisions on what sensors should be added to the system and then purchase those sensors to get them integrated as soon as possible. A lot of the work on the control algorithms is dependent on having the feedback from the proper sensor. The control algorithms will be discussed in the next section.

4.3 Controller Design Development

This section will compare the performance of the different controllers discussed in section 2.5. Each controller will be evaluated using a set of criteria that is specific to the controller being designed. Finally, Pugh matrices will be used to rank each controller against a datum controller to determine which controller should be used for each core lab. The controller designs discussed here are the same that were discussed in section 2.5: acceleration control, Adaptive Cruise Control (ACC), Lane Departure Assist System (LDAS), and stability control.

4.3.1 Acceleration Control

As discussed in section 2.5.1, the options for the design of the acceleration controller are a simple PID controller, a hybrid PID-Fuzzy controller, a hybrid PID-Fuzzy controller with PSO, an MMC-PID controller, and an MMC-SMC controller. While the PID controller is simple to implement and robust, it doesn't respond well to non-linearities and is limited—all that one can do to adjust the behavior of a PID is to change the values of K_p , K_i , and K_d . The other methods that will be explored in selecting an acceleration control algorithm are all essentially modifications or add-ons to a PID controller in order to customize and improve its performance.

The hybrid Fuzzy controller is meant to improve the PID's response to nonlinearities in the system. In a system as complex as a vehicle, there are many nonlinearities: tires, suspension, and powertrain, to name a few. It is important, then for the acceleration control to be able to adapt to a nonlinear system. The reason for the hybrid Fuzzy and PID controller is that the Fuzzy controller alone doesn't possess the steady state precision that the PID has.

The Fuzzy controller with Particle Swarm Optimization is simply meant to be another means of customizing the controller in order to affect the system's behavior. With the PSO, the Fuzzy variables can be customized in order to improve the system's time response, percent overshoot, and error. Figure 41 below shows the differences in response to a step input for the simple PID, the Fuzzy PID, and the Fuzzy PID with PSO.

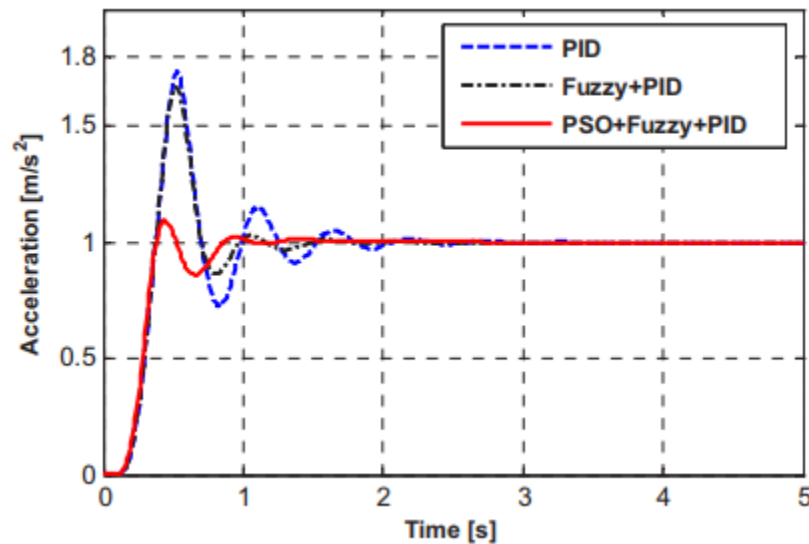


Figure 41. Response of PID controller, hybrid PID and Fuzzy controller, and hybrid PID and Fuzzy controller with PSO to a step input [Zhongpu and Dongbin]

The hybrid Fuzzy controller without the PSO improves the transient response of the simple PID controller—the oscillations of the system are more damped with the Fuzzy than with the simple PID. The PSO, however, dampens the PID response even further, resulting in a response that has very little overshoot.

Model matching controllers (MMC) were developed to have improved robustness and time response as compared to the conventional PID controllers. They can also be customized in selection of a controller for the feedback loop. One MMC design that was investigated for this application used a PID as its feedback controller, while another one used Sliding Mode Control (SMC). The SMC is essentially a modified PID that allows the system to better respond to nonlinearities. A simulation was performed that compared a simple PID controller, the MMC + PID, and MMC + SMC. In the simulation, a vehicle model was intended to follow a certain acceleration pattern. Figure 42 below shows how well each controller followed the input acceleration.

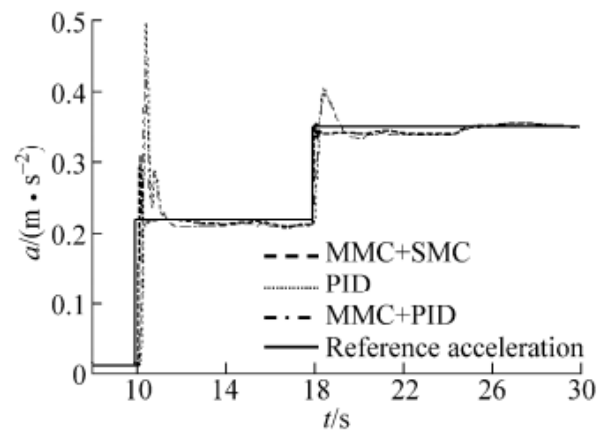


Figure 42. Response of PID controller, Model Matching Controller (MMC) with PID, and MMC with Sliding Mode Control (SMC) in acceleration simulation [Yuejian et al]

While all methods have similar time responses, the simple PID controller's response has the most overshoot. The MMC + PID has a slight amount of overshoot, but the MMC + SMC has almost none, and even appears to reach slightly below the reference acceleration. Another way of measuring the performance of these controllers is by looking at the error that was measured. A plot of the error during the simulation is shown in Figure 43 below.

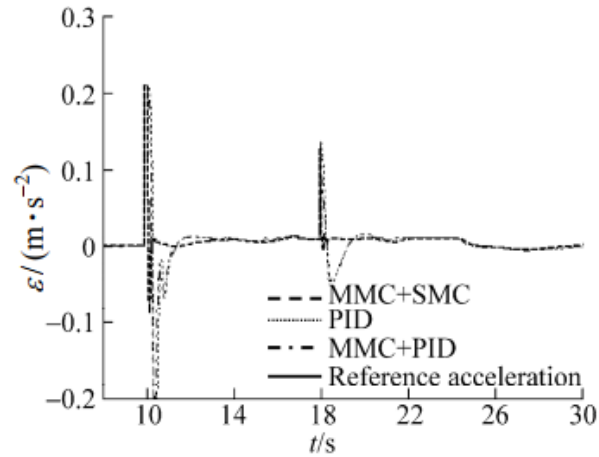


Figure 43. Error of PID controller, Model Matching Controller (MMC) with PID, and MMC with Sliding Mode Control (SMC) in acceleration simulation [Yuejian et al]

As in the acceleration response, the PID controller has the highest spikes of error, with the MMC + PID having the second highest spikes in error, and the MMC + SMC the lowest spikes of error. It appears, however, that all the control methods end up undershooting the final value.

A Pugh matrix was used to select which of these methods would be best to implement in the design of the acceleration controller. Each method was compared to a simple PID controller in regard to a number of criteria: smooth operation, ability to handle non-linearity, easy to implement, teachable to students, rapid response, low error, and low overshoot. The basic Pugh matrix was modified slightly in that these criteria were weighted in order to take into consideration the prioritization of the criteria. It was determined that criteria such as easy to implement and teachable to the students were more important than certain performance characteristics of the controller. The weights are shown in Table 10 below.

Table 10. Criteria and weights used in Pugh matrix for selection of acceleration controller

Criteria	Weight [%]
Easy to implement	30
Teachable	20
Smooth operation	20
Ability to handle non-linearity	10
Low error	10
Rapid response	5
Low overshoot	5

The weights were multiplied by the scores each method received when they were evaluated for a given criteria against the PID datum, which were either +1 or -1. The scores for each criterion were then summed. The Pugh matrix is shown in Appendix G.

The automatic score for the method that serves as the datum is zero. Since none of the other methods scored greater than zero, it follows that we should proceed with PID acceleration control.

4.3.2 Adaptive Cruise Control

The first and simplest method of ACC described in section 2.5.2 is PID control. This method, while simple to implement, doesn't have as desirable of a transient response as some of the other methods explored here. A simple step input was applied to the Simulink model described in section 2.5.2. The system's response is shown below in Figure 44 [Sivaji and Sailaja].

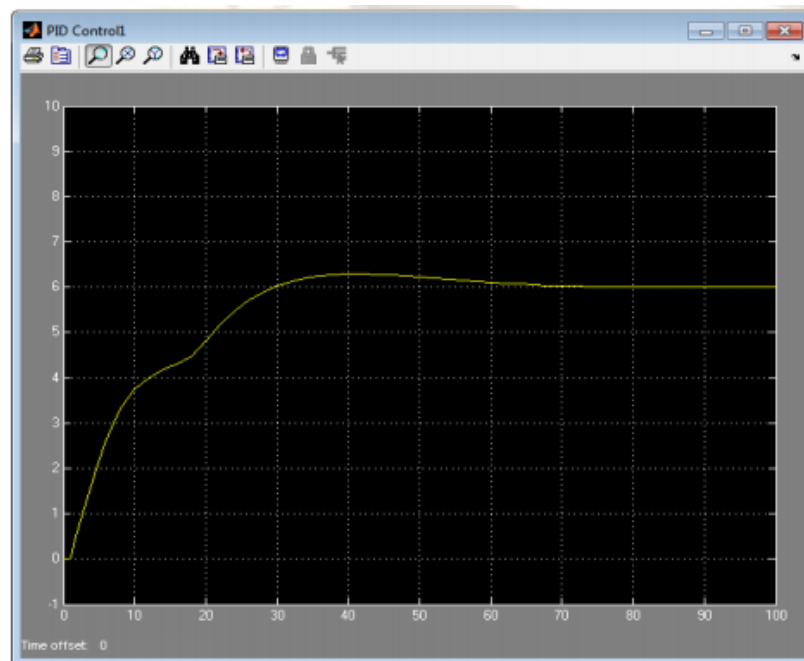


Figure 44. Response of Simulink model to a simple step input [Sivaji and Sailaja]

The PID controlled ACC system doesn't mimic the step input very precisely. The transient response is slow to reach the new desired value, and contains oscillations within it that may be undesirable for the vehicle performance. The controllers discussed in the following sections contain improvements upon the simple PID method.

The Model Predictive Controller discussed in section 2.5.2 was used in several different traffic simulations in order to determine its performance. The first scenario modeled a situation in which the ACC-equipped vehicle would be going along at cruising speed when a slower-moving vehicle merges in front of it. The ACC vehicle would then be forced to adjust its speed in order

to maintain a safe following distance. Figure 45 below shows the result of the simulation as measured in the following distance of the two vehicles.

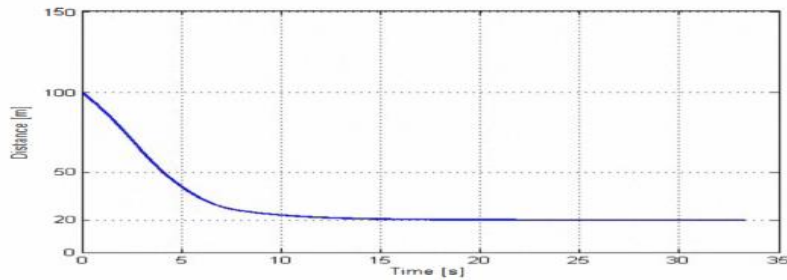


Figure 45. Response of simulated Model Predictive Control Adaptive Cruise Control to a slower-moving vehicle merging in front of it, as measured by the following distance [Kural and Guvenc]

The MPC adequately slows down the vehicle’s speed in order to maintain a safe following distance behind the leading vehicle, by adjusting the vehicle’s desired acceleration and sending the commands to its lower-level control, which executes them. The MPC’s acceleration matching control is shown below in Figure 46.

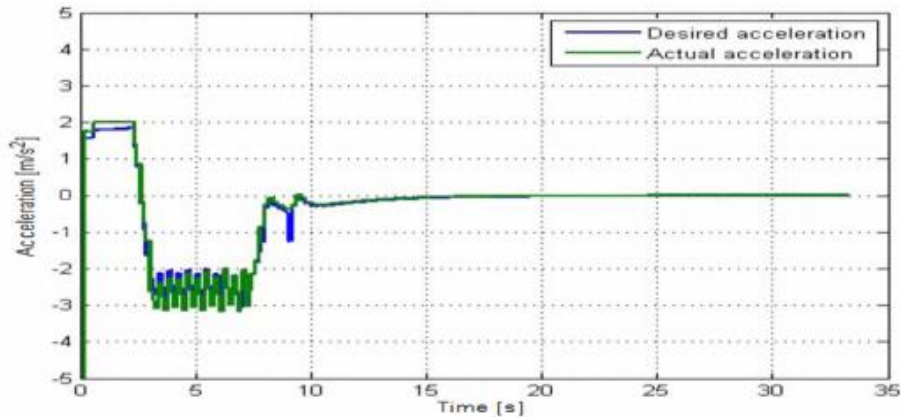


Figure 46. Acceleration tracking of Model Predictive Controller in traffic simulation [Kural and Guvenc]

The PI controller of the lower-level control loop of the MPC very precisely matches the vehicle acceleration with the desired acceleration output by the higher-level control loop [Kural and Guvenc].

The second scenario models a situation in which the ACC vehicle comes upon a vehicle cruising at a lower speed than the ACC vehicle. The ACC vehicle must then adjust its speed to the leading vehicle, even when the leading vehicle then accelerates again. Figure 47 below shows the speeds of the two vehicles in the first and second scenarios.

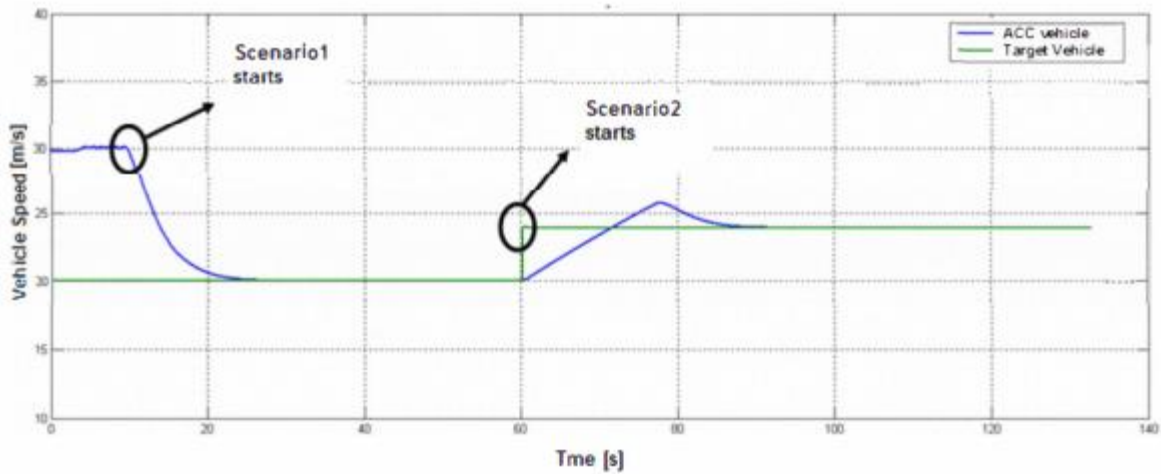


Figure 47. Velocity adjustment of Model Predictive Control Adaptive Cruise Control in a traffic simulation [Kural and Guvenc].

The ACC-equipped vehicle is capable of re-adjusting the cruising speed, as long as it is less than or equal to the set cruising speed, when the leading vehicle accelerates from a slower speed. The speed adjustment of the ACC vehicle also appears to be accomplished in a very smooth manner. Figure 48 below shows the achieved following distance versus the desired following distance for this scenario.

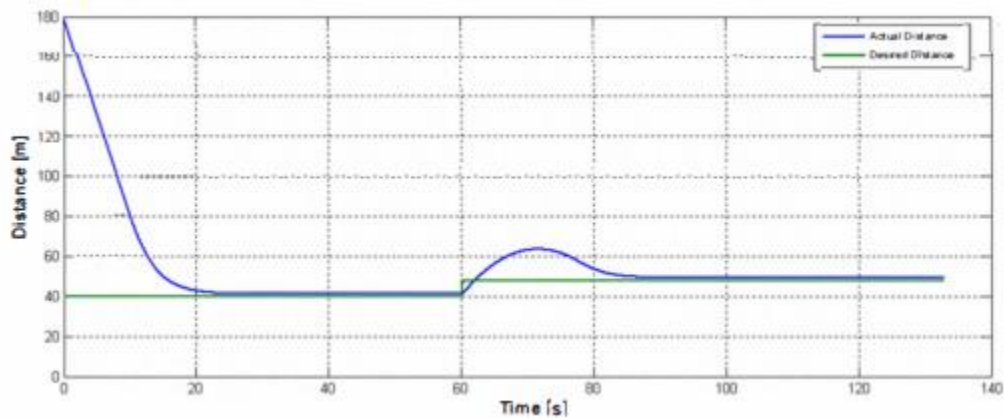


Figure 48. Following distance adjustment of Model Predictive Controlled Adaptive Cruise Control in a traffic simulation [Kural and Guvenc].

The MPC succeeds in never allowing the vehicle to go below the safe following distance in the simulation [Kural and Guvenc].

In the third scenario, the ACC-equipped vehicle is cruising behind a leading vehicle that is going the same speed. A third vehicle then merges between them. The ACC vehicle must then adjust its speed in order to maintain a safe following distance behind the third vehicle. In Figure 49 below, it can be seen that the ACC vehicle slows down when the third vehicle merges, and then accelerates back up to cruising speed once a safe distance has been established [Kural and Guvenc].

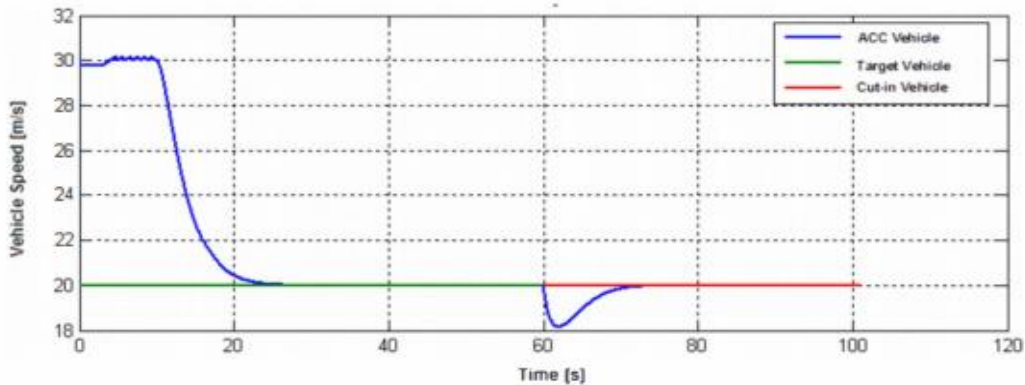


Figure 49. Vehicle speed adjustment of Model Predictive Controlled Adaptive Cruise Control in a traffic simulation [Kural and Guvenc].

The following distance between the ACC-equipped vehicle, the leading vehicle, and the third merging vehicle can be seen below in Figure 50.

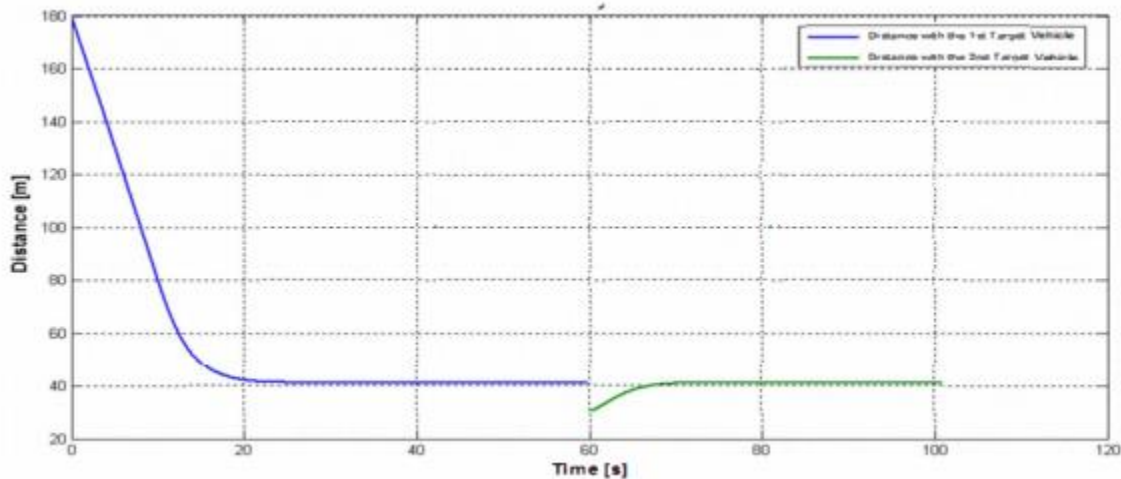


Figure 50. Following distance adjustment of Model Predictive Controlled Adaptive Cruise Control in a traffic simulation [Kural and Guvenc].

At the beginning of the merge, the following distance between the ACC-equipped vehicle and the merging vehicle is less than the desired safe distance. As the ACC vehicle adjusts its speed, however, the gap returns to the safe following distance [Kural and Guvenc].

The Model Predictive Controller has the advantages of working with nonlinear vehicle models, designing and executing smooth, accurate acceleration control, and maintaining a safe following distance at a minimum. Its performance as compared to other controllers will be studied later in the section.

The Adaptive Neuro-Fuzzy Predictive Controller (ANFPC) described in section 2.5.2 was evaluated in three different traffic scenarios. Its performance was compared to a Linear Quadratic Regulator (LQR) and a Constrained Linear Quadratic Regulator (CLQR). In the first scenario, the leading vehicle gradually decelerates. Figure 51 below shows the response of the three controllers, as measured in different parameters [Lin et al].

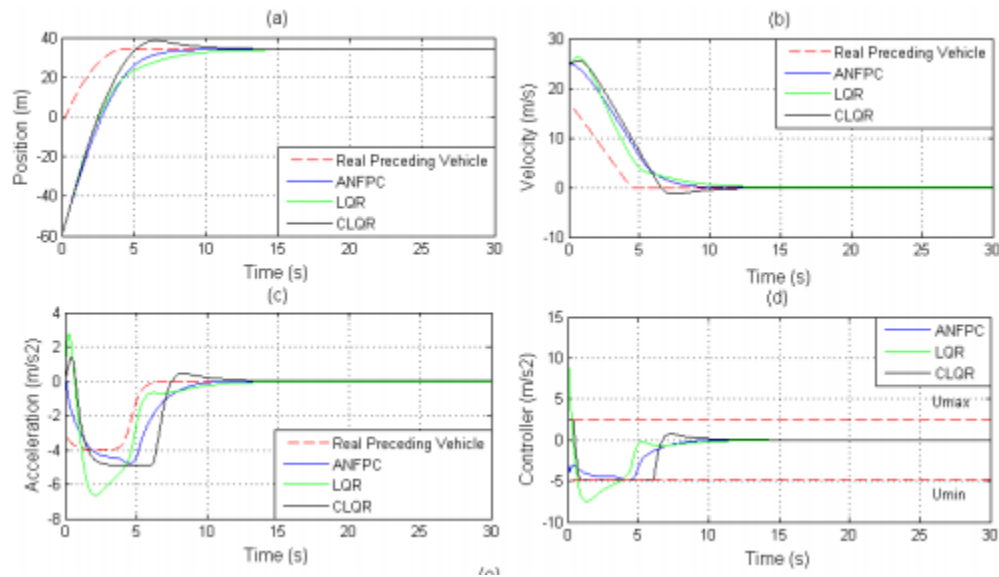


Figure 51. Response of Adaptive Neuro-Fuzzy Predictive Controller, Linear Quadratic Regulator, and Constrained Linear Quadratic Regulator to traffic scenario in which the leading vehicle gradually decelerates [Lin et al].

Figure 51 (a) shows the position of the three controllers over the time of the simulation. The ANFPC gradually reaches the new safe following distance. Figure 51 (b) shows the velocity of the leading vehicle and that predicted by the three controllers over the course of the simulation. The ANFPC displays better velocity control than the CLQR, and gradually brings the vehicle's velocity in line with the leading vehicle. The acceleration is tracked in Figure 51 (c), and shows that the ANFPC displays better acceleration control than the other two controllers. This is evidenced by the acceleration range in which the controller operated during the course of the simulation, shown in Figure 51 (d). The ANFPC is able to deliver the performance of the controller in a narrower range of acceleration and deceleration than the other two controllers, smoothing the ride and improving rider comfort [Lin et al].

In the second scenario in which the controllers were tested, the leading vehicle gradually accelerates. Figure 52 below shows the response of the same three controllers, with the same parameters measured.

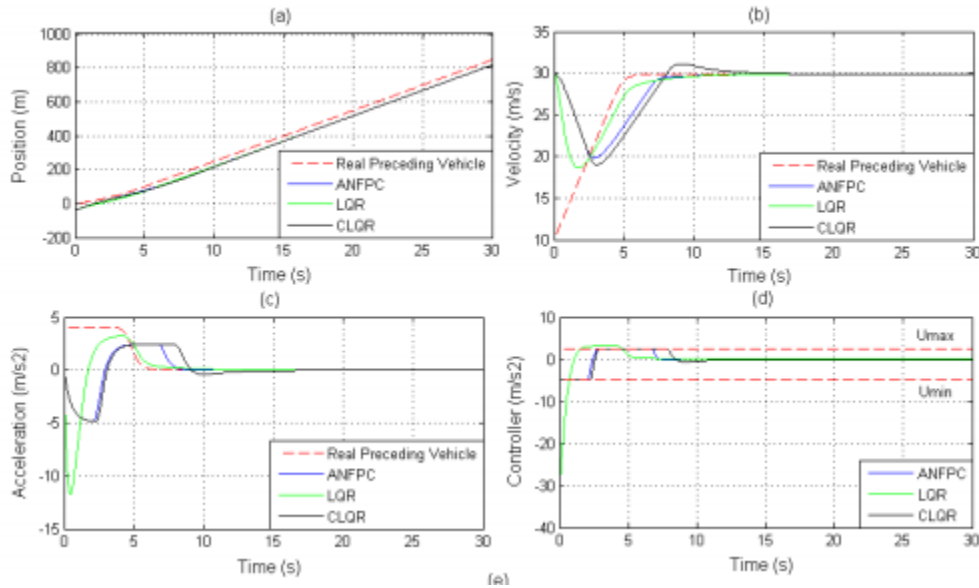


Figure 52. Response of Adaptive Neuro-Fuzzy Predictive Controller, Linear Quadratic Regulator, and Constrained Linear Quadratic Regulator to traffic scenario in which the leading vehicle gradually accelerates [Lin et al].

As shown in Figure 52 (a), all three controllers perform well at distance tracking the leading vehicle. In Figure 52 (b), it should be noted that the ANFPC performs better at speed control than the other two controllers. The ANFPC more closely matches the velocity of the leading vehicle, without excess adjustments in speed. The ANFPC and the LQR are also capable of staying within the acceleration limits given to ensure smooth behavior and rider comfort, as seen in Figures 52 (c) and (d) [Lin et al].

In the third scenario, the leading vehicle decelerates suddenly, rather than gradually as in the first scenario. Figure 53 below shows the responses of each controller in regard to position, velocity, and acceleration.

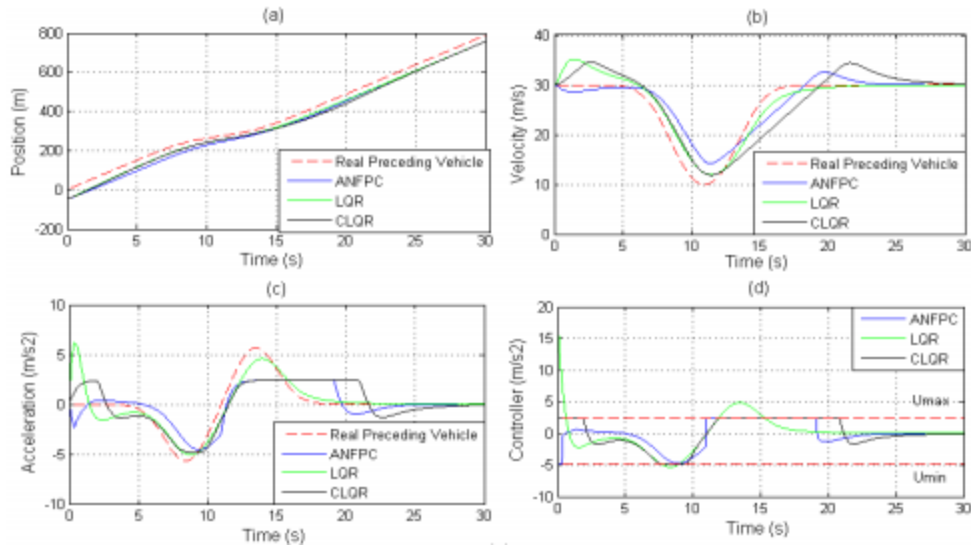


Figure 53. Response of Adaptive Neuro-Fuzzy Predictive Controller, Linear Quadratic Regulator, and Constrained Linear Quadratic Regulator to traffic scenario in which the leading vehicle suddenly decelerates [Lin et al].

In this scenario, while all three controllers perform adequate distance tracking, as seen in Figure 53 (a), the ANFPC's speed regulation lies in between the LQR and the CLQR, as seen in Figure 53 (b). the CPQR tracks the leading vehicle's speed the best, while the LQR and the CLQR are both a little bit slower to respond. However, the CLQR also exceeds the limits of acceleration, as seen in Figures 53 (c) and (d), while the LQR and ANFPC do not [Lin et al].

The ANFPC is able to rapidly adjust its following distance, speed, and acceleration because of the unique implementation of the T-S fuzzy predictive model that is able to model the leading vehicle's behavior [Lin et al]. Next the ANFPC will be evaluated against the MPC and the PID controllers to determine its appropriateness for this project.

A method similar to that used in section 4.3.1 was used to select which of the controllers is most appropriate for application in the intelligent vehicles course at Cal Poly. Each method was compared against the PID method using certain criteria such as ease of implementation, whether the method is easily teachable, ability to handle non-linearity, and success in distance, speed, and acceleration control. Each of these criteria were given weights in order to determine their relative importance. The weights are shown in Table 11 below.

Table 11 Criteria and weights used in Pugh matrix for selection of Adaptive Cruise Controller

Criteria	Weight [%]
Easy to Implement	15
Teachable	20
Able to handle non-linearity	10
Distance control	20
Speed control	15
Acceleration control	20

As in section 4.3.1, these weights were entered into a Pugh matrix that contained each of the methods. The matrix is shown in Appendix G.

Since neither the Model Predictive Controller nor the Adaptive Neuro Fuzzy Predictive Controller scored higher than the PID controller, it was concluded that a PID controller would be best suited for this project. The PID controller combines suitable performance with simplicity, whereas the MPC and ANFPC both have better performance, but are far more complex to implement and would be difficult to teach in an undergraduate course. In the next section, different methods of Lane Departure Assist Systems are explored for their viability of implementation in this course.

4.3.3 Lane Keeping Assist

The nested PID method described in section 2.5.3 was evaluated using simulations from the popular simulation program CarSim in order to evaluate the effectiveness of the controller. CarSim was used to model a driving scenario in which the vehicle experiences curvature at highway speeds. The controller presented in section 2.5.3 was compared to the controller embedded in the CarSim application, which is a Model Predictive Controller. The results of the simulation, measured in path-following error, steering angle, and yaw rate, are shown below in Figure 54 [Marino et al].

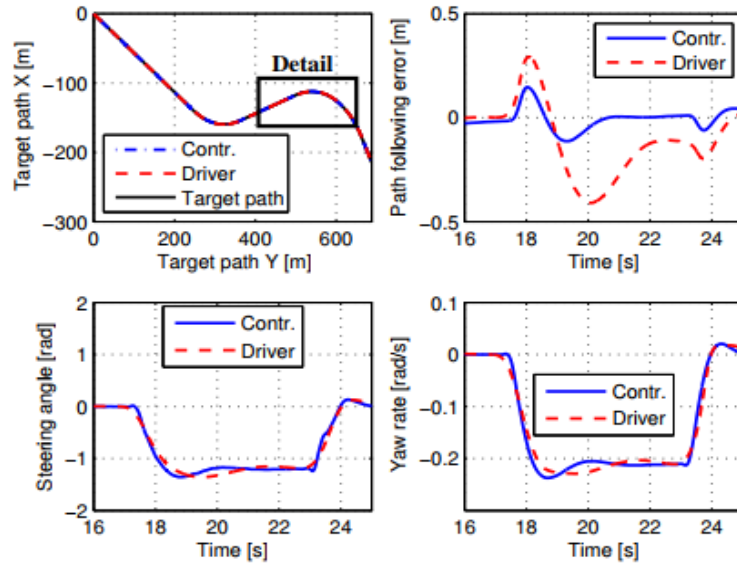


Figure 54. Results of CarSim simulation of nested PID controller LDAS and CarSim Model Predictive Controller LDAS [Marino et al].

The plot on the top left shows the target path that was set for the two controllers to follow. The detail area on the plot shows the point in the simulation in which the results of the other plots would be shown. The second plot on the top right shows the path-following error, in meters for both of the controllers. The nested PID controller has approximately 70% reduced path-following error compared to the MPC. The steering angle measurements and yaw rate measurements for the two controllers were similar, even though the nested PID greatly reduced the path-following error. This method will be compared with the other LDAS methods presented here in the following paragraphs [Marino et al].

The controlled invariance method described in section 2.5.3 was tested using a simulation in MATLAB. The simulation was meant to mimic a driver with a tendency to drift towards the right of the lane. The results from the simulation are shown in Figure 55 below.

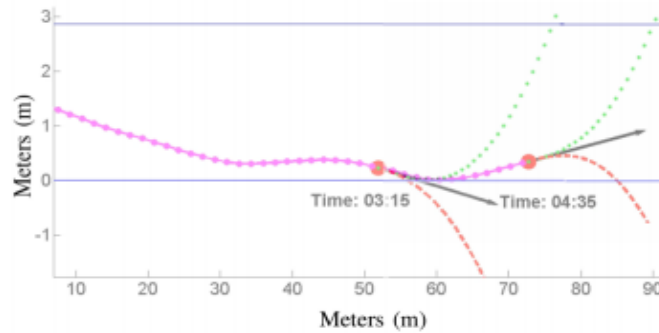
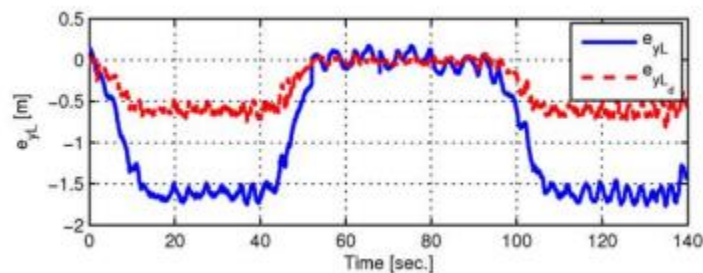


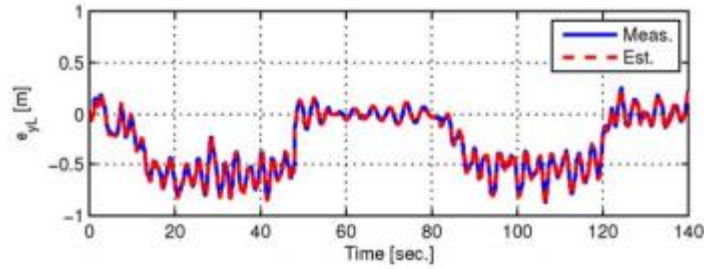
Figure 55. Results of MATLAB simulation to test controlled invariance LDAS method in which driver is simulated to drift to the right [Hoehener et al].

At time 03:15 in the simulation, the safety supervisor calculates that the driver's trajectory leads to an inevitable lane departure. At this point, the system intervenes and provides a new trajectory that not only avoids lane departure on the right, but also does not provide such a dramatic steering input that it would cause lane departure on the left. In later paragraphs, this system is compared to other LDAS methods to determine which is appropriate for the course [Hoehener et al].

The state feedback controller described in section 2.5.3 was implemented on a test vehicle that was then driven around a track. The test vehicle used was a Hyundai Tucson equipped with a vision system, an IMU, and wheel encoders to track wheel speed. The vehicle was driven around the curved test track at speeds up to 160 km/h to test how well the controller kept the vehicle in the lane. With the proportional state feedback controller, the error in the yaw rate was measured to be up to 1m. An integrator was later added to bring that error down to approximately zero. Figure 56 (a) and (b) demonstrate the improvements of the integrator in reducing the lateral offset error.



(a)



(b)

Figure 56. (a) Lateral offset error for a proportional state feedback controller. Figure 56 (b) Lateral offset error for a state feedback controller with an integrator [Son et al].

The addition of the integrator brings the difference between the desired lateral offset and the measured lateral offset to essentially zero.

The performance of the PID controller, controlled invariance method, and the state feedback controller with and without an integrator were compared in order to develop an LDAS controller for the project presented in this paper. As in section 4.3.2, weighted Pugh matrices were used to compare the performance, ease of instruction, and ease of implementation of each of the controllers. The specific criteria by which each controller was evaluated are: ease of implementation, ease of instruction, ability to handle non-linearity, lateral offset minimization, and smooth activation. The relative weights of each of these criteria are shown in Table 12 below.

Table 12. Criteria and weights used in Pugh matrix for selection of Lane Departure Assist System Controller

Criteria	Weight [%]
Easy to Implement	15
Teachable	20
Able to handle non-linearity	10
Lateral offset minimization	20
Smooth activation	15

These weights were entered into the weighted Pugh matrix. The PID method of LDAS was used as a datum against which the controlled invariance method and the state feedback method with and without the integrator were compared. The weighted Pugh matrix is shown in Appendix G.

All three controllers scored better in the criteria than the PID controller, which served as a datum. However, the state feedback controller with and without an integrator scored the same as compared to the PID controller. A second Pugh matrix was then developed to compare the state

feedback controller, with and without an integrator. The second Pugh matrix is shown in Appendix G.

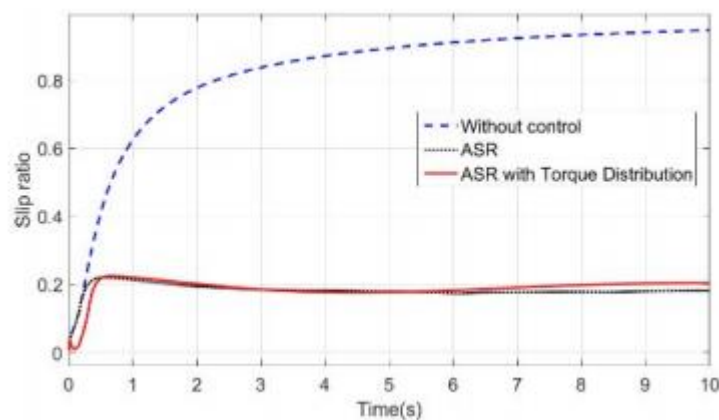
From the results of this second Pugh matrix, it is determined that a state feedback controller with integral control should be implemented in the Lane Departure Assistance System to best meet the needs of this course. The next section explores the different stability controllers presented in section 2.5.4 and evaluates them based upon the needs of this course.

4.3.4 Stability Control

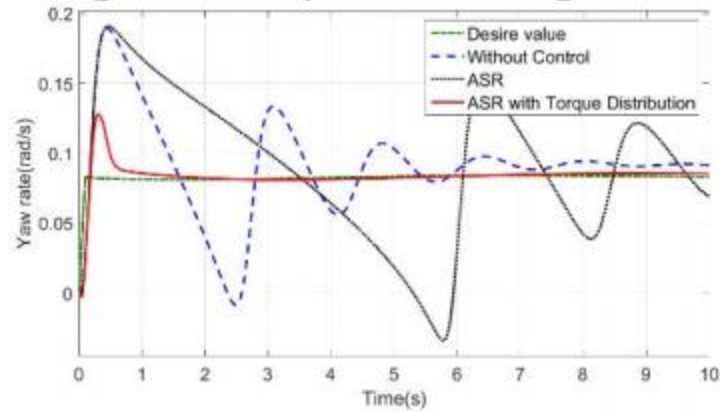
As discussed in section 2.6.4, there are many different types of stability controllers: Active Front Steering (AFS), Active Rear Steering (ARS), Four Wheel Active Steering (4WAS), Direct Yaw moment Control (DYC) through active braking, and DYC through torque distribution. The first design decision that had to be made was what kind of stability controller we wanted to design.

The vehicle platform that is to be designed in this project is currently powered by four electric motors that power each wheel. While that configuration may change throughout the course of the project (see sections 2.2 and 4.1 for the mechanical background and concept development), it would only alter in terms of the number of electric motors and whether there are two powering just the front wheels, two powering just the rear wheels, or one per axle. No matter the configuration, this setup is very well suited to DYC stability controller that modifies the torque output to each wheel. Additionally, it was in the best interests of the students in the course that the control algorithms that they are implementing be as simple as possible and accessible. As a result, it was decided that the stability controller would be a DYC torque distribution controller. The following paragraphs discuss the performance of each of the controllers presented in section 2.5.4. Then, each of the controllers are evaluated based upon how well they fit the needs of this project.

The fuzzy controller presented in section 2.5.4 was evaluated using a CarSim simulation in which the controllers were modeled in MATLAB Simulink. The first simulation that was performed was a simple constant velocity simulation with a low coefficient of friction between the tires and the road. The results of this simulation are shown in Figure 57 (a) and (b) below.



(a)



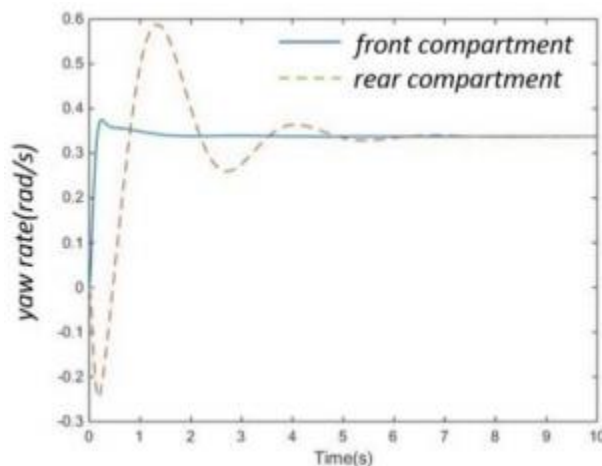
(b)

Figure 57. (a) Slip ratio throughout the course of the simulation (b) Yaw rate throughout the course of the simulation [Zhang et al].

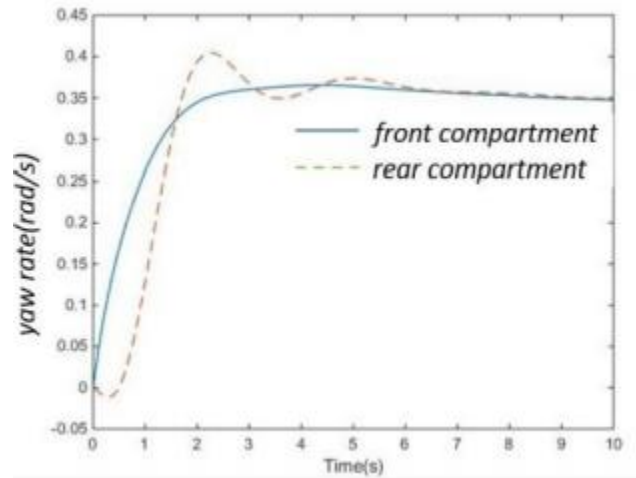
While the ASR is very effective at regulating the slip ratio, it is less effective at controlling the yaw rate of the vehicle. This is where the dynamic torque distribution controller comes in. The torque distribution controller can take in data on the vehicle's speed, slip angle, and steering angle, and using the rules laid out in section 2.5.4, determine the amount of torque that should be applied to each wheel [Zhang et al].

In the next section, a PID controller will be analyzed for its success in the application of vehicle stability.

The PID controller discussed in section 2.5.4 was tested with a vehicle simulation. The controller was applied to a three-axle vehicle with two driven axles and one steering axle. Different controller configurations were tested: solo PID on the front driven axle, solo PID on the rear driven axle, and double PID applied on both axles. The yaw rate was measured as the result of a step input in steering angle. The results for the solo-PID controllers on the front and rear axles are shown in Figure 58 (a) and (b) [Wang et al]



(a)



(b)

Figure 58. (a) Yaw rate response to a solo PID controller applied at the front driven axle (b) Yaw rate response to a solo PID controller applied at the rear driven axle [Wang et al].

While the solo-PID controllers at each of the driven axles did stabilize the vehicle, the stabilization took place faster at the axle with the controller. The response at the driven axle without the controller is much more sluggish. This phenomenon can be seen in Figure 58 (a), where the rear axle takes 5 seconds to stabilize, while the front axle with the controller is stabilized in a fraction of a second [Wang et al].

The last configuration that was tested was the double PID controller applied at both driven axles. The results are shown in Figure 59 below.

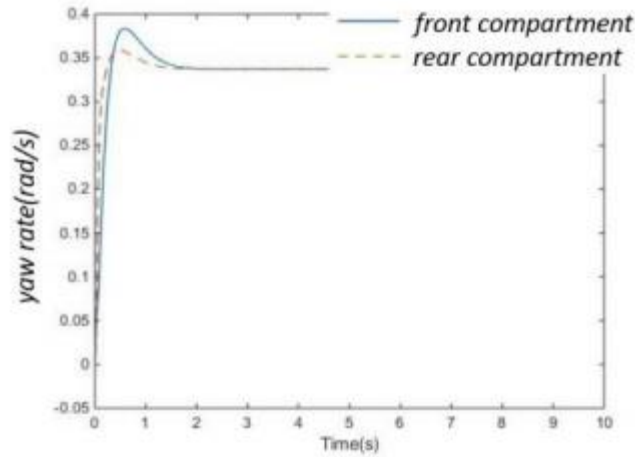


Figure 59. Yaw rate response to a double PID controller applied at both driven axles of a three-axle vehicle [Wang et al].

Here, both axles stabilize almost within a second. This is a marked improvement upon the solo-PID design. The double PID stability controller is a potential design for the stability controller for this project, and will be compared against the other methods presented in this section later on.

The state feedback controller presented in section 2.5.4 was tested in two different simulations. In both simulations it was tested at constant vehicle speed, with a step input of steering angle. What was varied between the two simulations was the coefficient of friction between the tires and the road. The first simulation simulated the vehicle driving on a dry asphalt road, and as such the coefficient of friction between the tires and the road was very high. In the second simulation, the controller was tested on a road with a low coefficient of friction, such that one might find on roads with snow and ice. The results for the dry road simulation are found in Figure 60 (a) – (d) [Esmailzadeh].

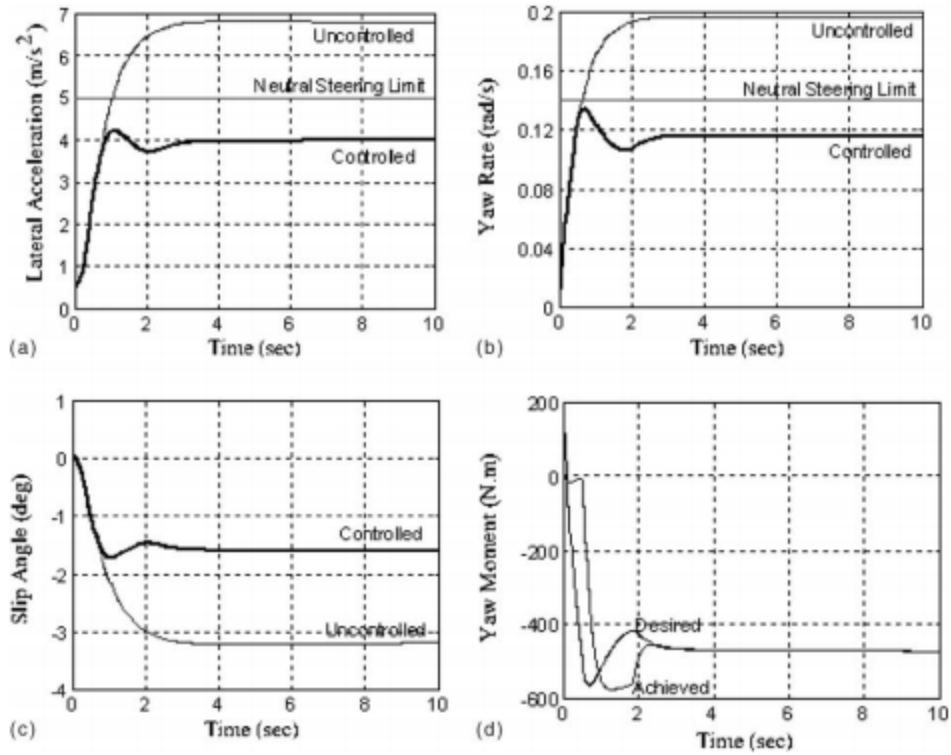


Figure 60. (a) Lateral acceleration response of vehicle with optimal controller at a constant speed on a dry road with a step input of a steering angle, as compared to an uncontrolled vehicle (b) Yaw rate response (c) Slip angle response (d) Yaw moment response [Esmailzadeh].

The optimal controller is very effective at reducing the lateral acceleration, yaw rate, and slip angle, as compared to the uncontrolled vehicle. Additionally, the controller was able to achieve almost perfect matchup between the actual and desired yaw moment within 3 seconds. Figure 61 (a) – (d) shows the results from the simulation on the snowy road [Esmailzadeh].

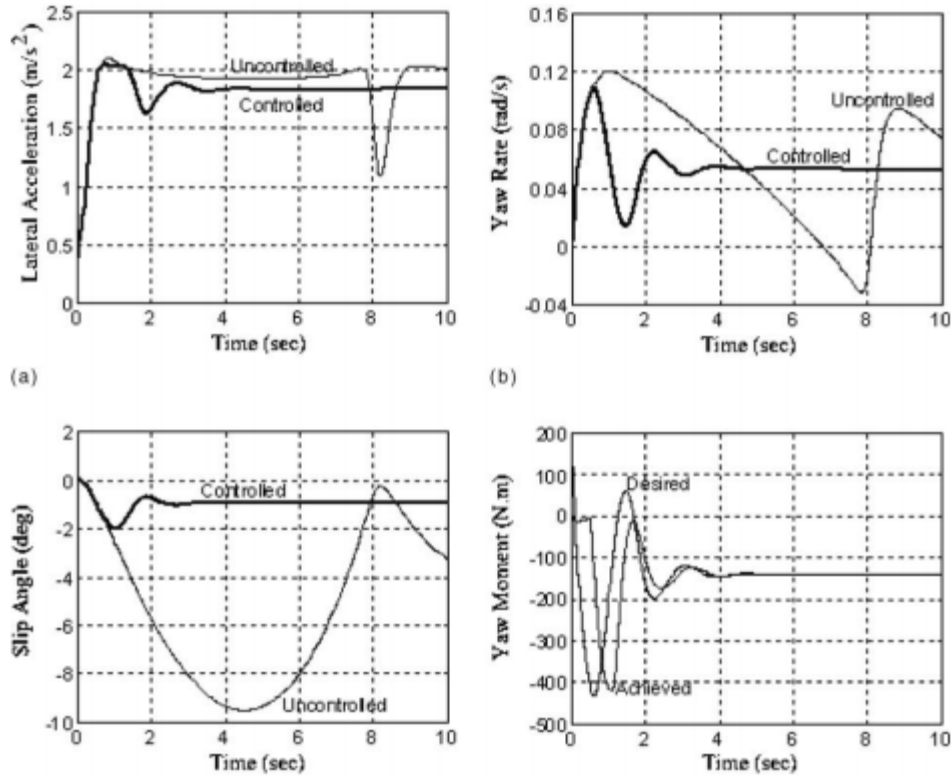


Figure 61. (a) Lateral acceleration response of vehicle with optimal controller at a constant speed on a snowy road with a step input of a steering angle, as compared to an uncontrolled vehicle (b) Yaw rate response (c) Slip angle response (d) Yaw moment response [Esmailzadeh].

In this simulation, the optimal controller was able to achieve a slight reduction in the lateral acceleration as compared to the uncontrolled vehicle. It was, however, able to achieve significant stabilization of the yaw rate, and significant reduction in the slip angle as compared to the uncontrolled vehicle. The controller also achieved matching between the achieved and desired yaw moment within four seconds.

The three controllers evaluated above: fuzzy, PID, and optimal state feedback, were evaluated for their appropriateness for this project according to a number of criteria: ease of implementation, ease of instruction, ability to handle nonlinearity, yaw moment minimization, yaw rate minimization, slip angle minimization, and smooth activation. The criteria were weighted according to the weights shown in Table 13 below.

Table 13: Criteria and weights used in Pugh matrix for selection of Stability Controller

Criteria	Weight [%]
Easy to Implement	20
Teachable	20
Able to handle non-linearity	10
Yaw moment minimization	10
Yaw rate minimization	10
Slip angle minimization	10
Smooth activation	10

These weighted criteria were used in the development of a Pugh matrix. The PID controller was used as the datum against which the Fuzzy and optimal state feedback controller were compared. The Pugh matrix is shown in Appendix G.

The results of the Pugh matrix show that the optimal state feedback controller would be most appropriate in this application. The optimal state feedback controller combines robust control with ease of implementation and a platform that would be challenging for the students, but not conceptually out of their reach.

In this section, we explored the different control methods that were presented in section 2.5 for each of the core labs of this course: acceleration control, Adaptive Cruise Control (ACC), Lane Departure Assist Systems (LDAS), and Stability Control. We determined that PID control would be best for the acceleration control and Adaptive Cruise Control functions, and state feedback control would be best for Lane Departure Assist and stability control.

The first task in implementing these controllers is to develop the vehicle model, as described in section 2.1. Once the vehicle model is developed, we can begin to implement it in the design of the controllers. One of the major unknowns at this stage in the development is how we will deal with the sensor data. The sensor data must be output in a form that is useful to the controller. One main struggle will be organizing the sensor data and performing unit conversions such that the data can be integrated into the controller functions.

Another hurdle that will have to be overcome is the design of the state feedback controllers. Only one of our group members has taken a class in state space controls, so the rest of us will have to teach ourselves a lot of the basics to be able to design the state feedback controllers.

4.4 Software/Firmware Concept Development The design process of the Teensy firmware and further research into Simulink software solutions are discussed below. An outline of the firmware ideation that occurred and initial task diagram creation is included along with an investigation of Mathworks software that may improve the performance of the system. Later in

the design process, the choice of the Teensy as a vessel for the firmware and its role in the system was reassessed. Discussion on this topic occurs in section 5.5.

4.4.1 Teensy Firmware Design

Software and firmware design is a unique area of this project, and does not lend itself to the decision matrix method of concept development. Instead, brainstorming sessions led to a list of potential tasks. An initial firmware architecture in the form of a task diagram was created to help initialize the firmware design. But firmware can be rapid prototyped quite easily and once a basic structure is created, additional tasks can be added when a need for them arises. It isn't even necessary to decide on the use of a Real Time Operating System (RTOS) yet as the design of tasks as finite state machines can work for both an RTOS and a cooperative multitasking scheme.

Specifics of the firmware design will likely arise as Simulink and control system design, user interface, and other specifics arise in their design processes. For instance, the need for a more computationally intensive task such as data filtering or other complex data analysis has not yet been established and if it is needed, it is unclear whether it would be implemented in Simulink or on the Teensy. The firmware could integrate this new task very easily (designing the task is still a challenge) and may even change from cooperative multitasking to RTOS to accommodate the additional computational load the code introduces, though this would take a bit more work to accomplish than simply adding a task.

The developed task diagram is shown in Figure 62 below. The tasks are split into two types: Dynamic and Static. This will come up naturally in the design as menu and LCD tasks will only need to run when the vehicle is stationary and sensor feedback and control loop communication tasks will only run when it is driving. Priority and timing associated with an RTOS are not assigned to each task but will be if an RTOS is deemed necessary. The task diagram was revisited later, in the context of moving away from the Teensy microcontroller (section 5.5). Many of the static tasks were combined into one called Mastermind, and the Dynamic tasks were set aside until further understanding of the Raspberry Pi is acquired. The task Mastermind was designed as a finite state machine that took over the roles the other tasks were to have. There will likely be more Dynamic tasks in the future but they may not be the same as described here to the firmware design decisions discussed.

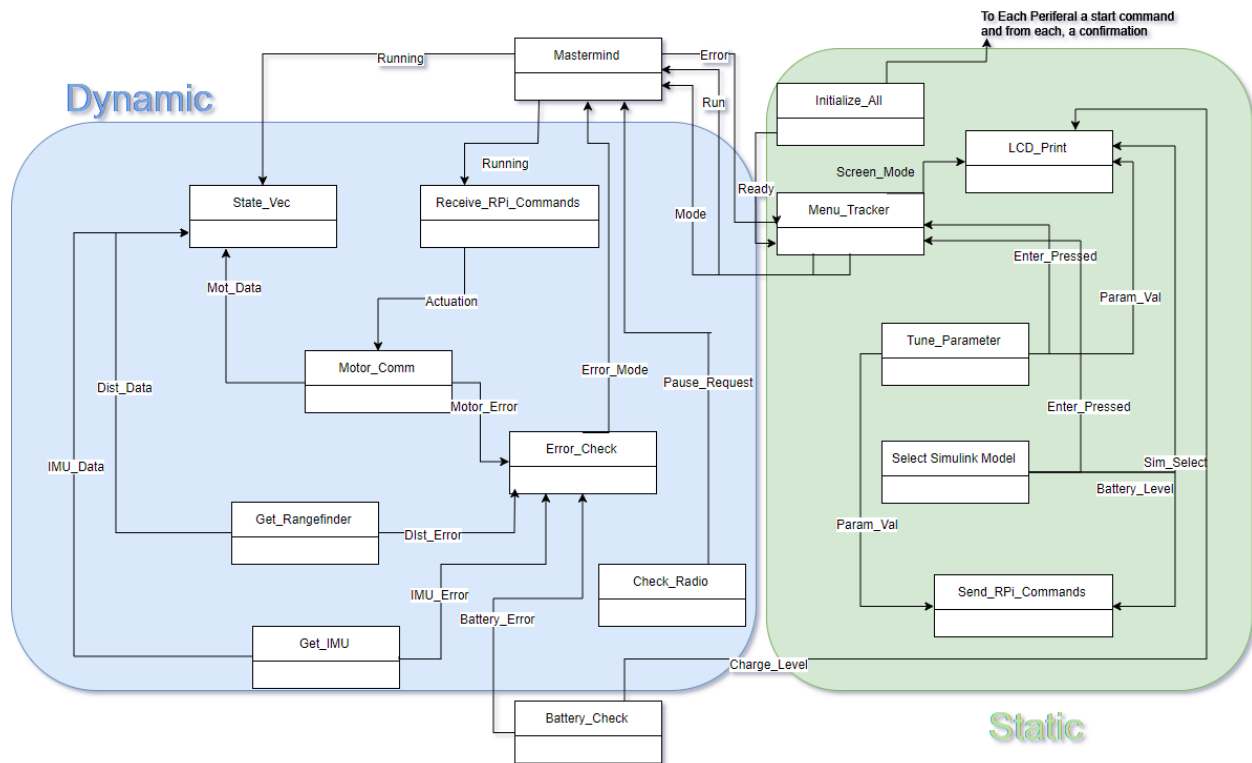


Figure 62. Initial Teensy firmware architecture concept.

It is necessary to provide a brief explanation of the intended purpose of each task in the diagram. Arrows show the data and flags shared between tasks, but further explanation is included below.

Dynamic Tasks

Mastermind:

This task will determine the operating mode of the vehicle, setting and clearing flags that signal certain tasks to go dormant under dynamic or static operating conditions. It is anticipated also to act on error states and transition the operational mode accordingly.

Get_Rangefinder and Get_IMU:

These tasks will not be the only sensor acquisition tasks, they will exist alongside a similar task associated with each sensor. These will communicate with each sensor, gathering readings when asked and storing them in the sensor data array at the proper address.

State_Vec:

Currently the firmware will send the most recent measurement from a particular sensor to the Raspberry Pi when it asks for it. This is done using an interrupt service routine that responds to the receipt of a Serial Peripheral Interface (SPI) request message from the Pi. An interrupt service routine is a block of code that runs after dedicated interrupt hardware reacts to a stimulus such as the toggling of a pin in response to a timed signal. It will halt all other operations and run the code associated with the service routine before resuming the previous state of the program. There is potential here for data points that should correspond to the same timestamp being asynchronous if the request messages are sufficiently separated in time. This task will attempt to

accomplish the same task - communicating the entire state feedback vector - while maintaining the desired synchronization such that the vector is representative of the state of the system for every Simulink time step.

Motor_Comm:

This task will update the motor controllers with Simulink actuations or setpoints communicating them over CAN. It will likely need to have low latency and respond to a message as quickly as possible.

Receive_RPi_Commands:

Motor commands from the Raspberry Pi are currently part of the same interrupt service routine as the sensor feedback data. This will likely be changed and have the received commands be more closely linked to the *controller* outputs of the RPi. The task is designed separately of the *State_Vec* task as a result. It is in the interest of readability, and modularity to keep these separate as the controller someone designs for the car (particularly researchers) may need to behave differently than the sensor feedback. It also fits in well to the way a traditional controls block diagram is organized.

Error_Check:

Error check receives messages from the different sensor, motor and battery tasks, interprets them, and sends an error mode message to Mastermind that is configured in such a way that Mastermind knows how to respond to it. It is grouped this way so that all error handling can be done in one block of code, improving readability by helping designers or technicians locate every type of error message that has been defined.

Battery_Check:

It is not yet certain how exactly this task will monitor and regulate the battery level and flow of power. It may require dedicated hardware such as a printed circuit board with current and voltage monitoring elements. Ideally this task would be able to display a charge level, detect current spikes, battery temperature, and be able to shut off power if need be. It will communicate with the LCD print task described below during static operating conditions.

Check_Radio:

This task is a relatively simple one; it will wait for trigger and steer messages from the radio controller. Certain radio inputs will be able to change the operational mode of the vehicle or, if manual mode is running, communicate with the motors and servo.

Static Tasks

Intitalize_All

This task will initialize all the peripherals (motor controllers, radio receiver, servo, etc.) and wait for confirmation before setting a ready flag. This task will likely only run once but may get incorporated into a reboot task if one is necessary. The task will have to know the startup sequences and code associated with each peripheral as well as set up all the communication bus code.

Menu_Tracker

The Menu that the user will navigate when using the user interface must be written as a task. This task is really the most likely to require the use of the finite state machine structure as there are several distinct “screenshots” that need to be printed to the LCD for each window of the menu. It will keep track of the user numbered input buffers and set flags that tell the LCD_Print task what to print. Additionally, the input buttons will mean different things for each of these windows and the proper menu task must be activated for the inputs to make sense. The tasks Select_Simulink_Model and Tune_Param will activate based on flags set by this task though these task may be absorbed into the Menu_Tracker task if their code ends up being too simple.

LCD_Print

This task manages what the LCD displays based on commands from Menu_Tracker. It will know what LCD device it is interfacing with (the Crystalfontz product has been selected currently) and understand its command set. It will have no “intelligence” other than printing what Menu_Tracker tells it to.

Send_RPi_Commands

The user will have the capability to access Raspberry Pi processes such as selecting from uploaded Simulink models while navigating the menu. This task sends the appropriate messages over the proper protocol to make the inputs heard by the Pi. It will be triggered by messages from

Tune_Parameter and Select_Simulink_Model

It is not yet clear whether these tasks will be absorbed by the Menu_Tracker. Their current intended purpose is to convert into Raspberry Pi compatible data, which is input by the user when their menu window is open.

4.4.2 Simulink Software Considerations

In section 2.3.2, the problem of Simulink timing requirements was introduced. Further research was done on this topic and a potential solution involving the Mathworks software Embedded Coder was investigated. Further control system design may prove the need for a real-time operating system to be created on the Raspberry Pi and if so, not only will a program like this be necessary, the Simulink designs that are compiled from it need to be easily and automatically meshed with that system. The focus of this section will be to explain some relevant features of Embedded Coder - features that may warrant a switch between the current Raspberry Pi to another single-board computer that is compatible with the software.

A major part of designing well informed Teensy Firmware is knowledge of how the Simulink models it will interact with compile and run on the single-board computer. This knowledge is accessible through Embedded Coder in the form of code generation reports. These are reports that contain the C code generated from a Simulink model. They allow the designer to study how the code will run on embedded hardware. They also let you trace the generated code to blocks and signals in your model.

These reports would allow insight into how certain blocks, such as those used to communicate to the Teensy, behave. For instance, in the current Simulink model, there is a Stateflow block that requests sensor information and sends commands over SPI to the Teensy. It is currently unknown whether all of these requests happen sequentially or whether they completely execute before the next control loop timestep. An Embedded Coder code generation report could tell us

whether this communication is interrupted at any point in its execution. In addition to these, the program can create a dictionary of important parameters and data definitions, essentially auto-generating documentation that helps interpret the generated code.

Another feature that produces good compatibility with the Teensy is the ability to alter data types of signals and parameters. Abstract data types can be created, and native Simulink types can be altered. This ability would be useful for maintaining homogeneity of information between the two systems and allows Teensy code to be written that has better access to functions the Simulink model generates. For instance, timestamped data types may prove useful on the Teensy for error checking purposes: Simulink could send timestamped data that the Teensy can check against its own internal clock to see if a request was missed.

Embedded coder allows for merging with an RTOS through features specifically designed for automatic integration with one. These features would allow the control loop to meet timing and latency requirements set up by a custom RTOS written on the single-board computer it is running on greatly improving real-time behavior for computationally intensive control algorithms. Embedded coder provides an “Embedded Real-Time Target” allowing the user to select a specific real time-operating system to run on. It will create automatically the necessary compatibilities to merge with it. This mode feature does not have to be used however, as there are also single-tasking, multitasking, and asynchronous multi-rate modes. If the algorithms created run fine in these modes, RTOS integration is not necessary.

Future work and testing will need to be done before Embedded coder can be considered essential, but knowledge of its existence and its capabilities will help guide design decisions related to control algorithms needed for the desired autonomous vehicle functions.

4.5 User Interface Concept Development

It was decided that the “ease of use” requirement could best be met by designing a user interface with menu system that allows students and researchers access to important vehicle functions. Currently, once a Simulink model is deployed onto the Raspberry Pi, the user must navigate through a complicated command line procedure without the help of a graphical user interface in order to run the deployed model. Designing a user interface for the vehicle would eliminate the need to manually enter commands to the Raspberry Pi via laptop or computer each time a model is deployed. The user interface could be programmed to automatically send commands that select or switch between uploaded models or alter model parameters.

This solution will allow for a seamless transition between the design and testing of autonomous vehicle algorithms as well as serve as a general-purpose display for errors or other messages. Several solutions arose in brainstorming that would satisfy the need, but the most feasible and likely solutions took the form of an LCD interface. Several products available for purchase as well as some custom-designed options and combinations were considered, and a final concept design was decided upon using the Pugh matrix method.

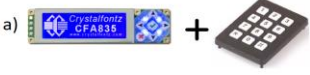






4.5.1 User Interface Hardware Ideation

Ideation was done to come up with some potential user interface hardware. Methods of physical interaction (buttons, knobs, etc.), ways to display the necessary information, and combinations of both that could potentially satisfy the needs of the interface were conceived. After an initial Pugh matrix was done, a few lead concepts consisting of both commercial solutions and custom ideas (Table 15) were selected for use in a final Pugh matrix (Appendix H). These were evaluated against criteria that reflect the project needs (Table 14). All options were identified as compatible with communication ports available on the Teensy before being entered into the matrix.

Table 14. Criteria used in Pugh matrix for selection of hardware for user interface

Criteria
Ease of programming menu/printing/interacting with Teensy
User friendly
Time to integrate/manufacture
Power consumption
Amount that can be displayed at once
How nice does it look and feel
Reliable
Cost effective
Ability to change numerical values

Table 15. Hardware options compared in user interface Pugh matrix

Description	Image
a) Crystalfontz LCD product with number keypad	
b) Crystalfontz LCD product with rotary encoder	
c) "Screenkeys" product and Graphic LCD screen	
d) "Screenkeys" product and character LCD screen	
e) Graphic LCD with directional keys and rotary encoder	
f) Graphic LCD with directional keys and number keypad	
g) Character LCD with directional and number keypad	

The Crystalfontz LCD product has directional arrows, as well as “enter” and “cancel” buttons that are fully decoded out of the box and the company provides a command set for drawing lines, symbols, characters, etc. Additionally, they have two mounting options. This product ranges in price from \$60 - \$100 for a single device.

Screenkeys is a product that consists of multiple button-LCD hybrids i.e. there is an LCD display on each button. These could be arranged as pictured and be configured to display many different things such as a set of directional arrows with “enter” and “back” buttons or an entire number keypad. There is a Windows based graphical user interface for testing and development of a user interface. We were unable to get pricing information for this product as the company did not reply to us, so the pricing cost-effectiveness score was set to zero in the Pugh matrix.

A character LCD is a display that has fixed rows where characters can be displayed and can display less than a graphic LCD is able to for the same screen size. Character LCDs are in the range of \$10-\$30 and graphic LCDs are in the range of \$15 - \$100 for less than 3” diagonal screen size. These would be combined with some combination of a rotary encoder, directional arrows, and number keypad.

The rotary encoder by Adafruit is \$5 and is used as a rotational input to adjust numerical values. A number keypad is around \$20 and is used to punch the numerical values, a different way of entering them than the rotary encoder. The directional arrow button modules are priced at \$20 each and are for navigating the menu.

4.5.2 User Interface Hardware Pugh Matrix Result

After two rounds of Pugh matrix decision making, the Crystalfontz LCD with number keypad (option a) was decided upon. The two runners-up were the Crystalfontz LCD with rotary encoder and Screenkeys product with graphic LCD screen. The number keypad scored higher for its ability to alter numbered values than the rotary encoder and uncertainty in the pricing of the Screenkeys had the Crystalfontz LCD win out. This solution will be easier to integrate than the other options, has pre-programmed buttons and an instruction set that will ease our menu design process. It has variable brightness options for power saving and will be more reliable than the other custom options we came up with because much of the functionality is designed commercially and only requires the addition of one module (the keypad) whose function is simple to implement.

5. Final Design

The overall design presented in this section is a culmination of the separate design decisions made in the mechanical, sensors, software/firmware, and controls sections of this document. As discussed in section 4.1, the mechanical improvements were originally to add a protective shell to the car to protect the electronics, add a “double decker” fixture in order to provide mounting points for the sensors, and to investigate the ability to change the manufacturing processes—specifically of the shaft couplers. Through a reduction in scope, the only remaining mechanical element of the design process became the selection of cost effective motors and encoders. It was decided to move to external encoders, as well as include motor drivers that will be developed by Charlie Revfem. These will significantly reduce the cost of the platform.

As discussed in section 5.3, the only sensor that was added to the vehicle is an indoor GPS sensor. These, in addition to the Inertial Measurement Unit and Time of Flight sensor that are already on the vehicle will be able to supply enough feedback for the four core labs for the course: acceleration control, Adaptive Cruise Control (ACC), Lane Departure Assist Systems (LDAS), and stability control.

The controller designs presented in section 4.3 have all been modified slightly—the changes made are presented in section 5.4. The final version of the firmware is presented below in section 5.5.

5.1 Motor Selection

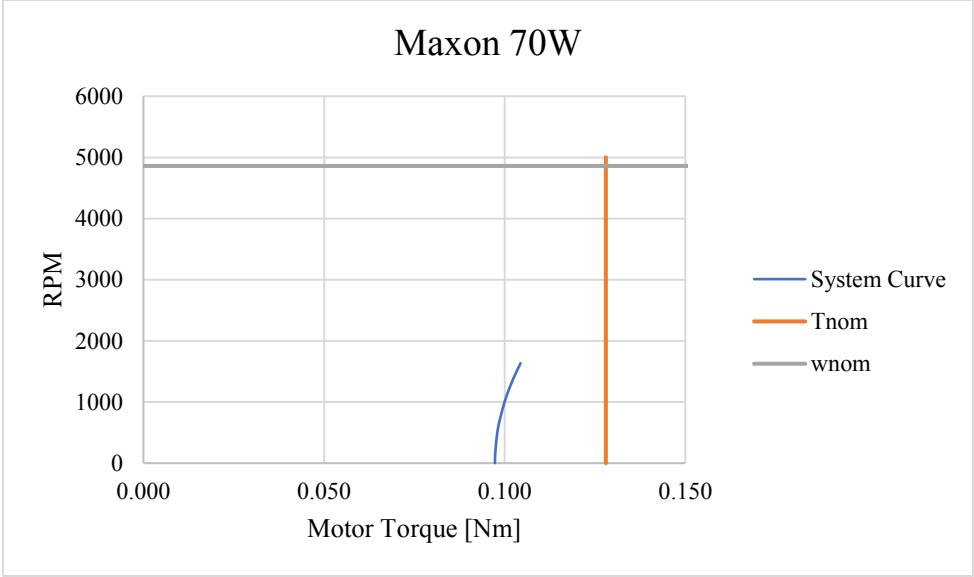
Analysis was performed to determine whether more cost-effective motors could be used on the vehicle, without sacrificing too much in the name of performance. The goal was to bring the per vehicle cost down. A broad selection of brushless DC motors were analyzed for their performance characteristics and physical compatibility with the existing vehicle. This selection included Maxon motors like the current motors on the vehicle, as well as motors from such brands as Nanotec, Turnigy, and Multistar. Lower-cost brands such as Turnigy and Multistar offer motors that are used in hobbyist projects such as drones and helicopters. These motors have high maximum torques, which would improve the acceleration of our vehicle, but because they are inexpensive, do not have a lot of documentation on their performance characteristics. Their physical dimensions would have also required extensive redesign to implement onto our vehicle. Because we didn't want to purchase motors and redesign parts, only to find that the motors didn't fit with our application, we decided to go with what we knew and continue using Maxon motors.

In order to try to bring the cost of the high-quality Maxon motors down, we considered replacing the current 70W motors with 50W or 30W versions. Another way we could bring the cost down was by getting the motor without an internal encoder, an add-on that adds \$180 to the cost of each motor. We could instead purchase an encoder separately and attach it to the output shafts of the motors.

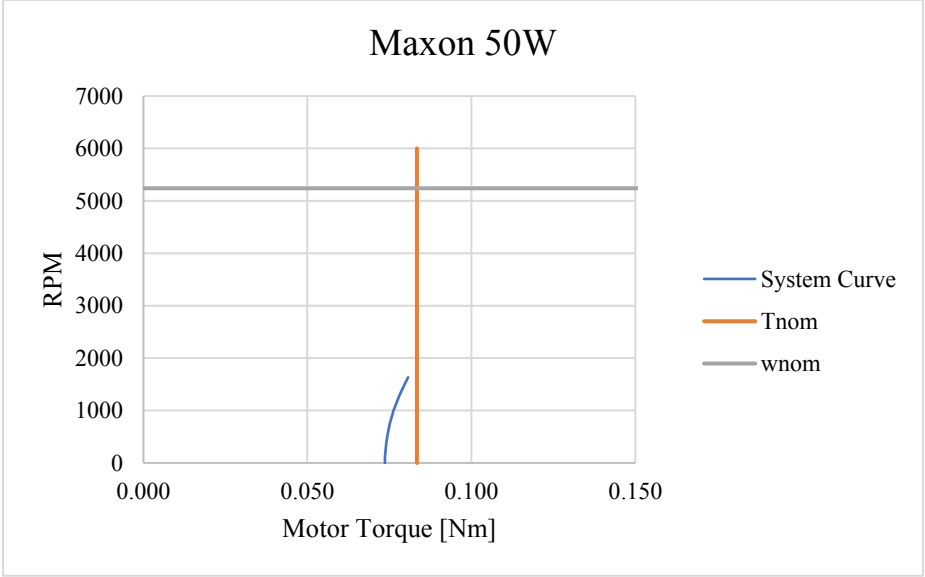
We ended up considering six options for the motors: the 70W, 50W, and 30W Maxon motors, each with internal or external encoders. To compare the performance of the 70W, 50W, and 30W motors, we developed a system curve. The system curve represented the demand on the motors if the car were to accelerate at its maximum acceleration from a standstill up to its top speed on a level surface. This operating condition was selected because it is a theoretical “worst case” for the motors, and we wanted to make sure that the motors were able to handle even the most extreme case that they might face once implemented. For each motor, we mapped the system curve onto a plot with the continuous region of the motor—the torque and speed values at which the motor could run continuously. By making sure that the demands of the vehicle in this extreme case lie within the continuous region of the motors, we could thus ensure that all normal operation of the vehicle would lie within this continuous region. It is important for us to stay within the continuous performance region of the motors in order to preserve the life of the

motors and to keep them from overheating during use. The Excel sheet used for this analysis is shown in Appendix I.

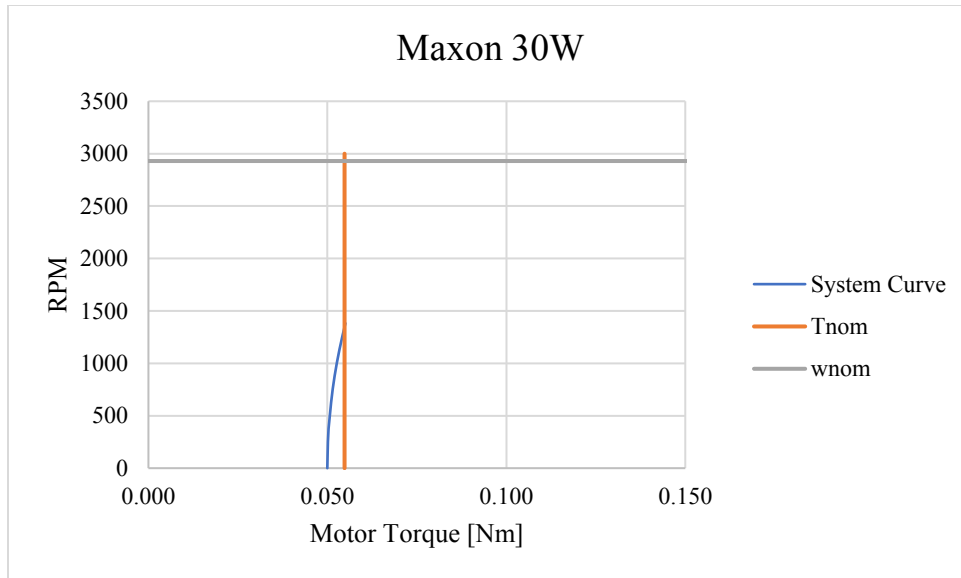
For each of the 70W, 50W, and 30W motors, the system curve was mapped onto the continuous region of the motor. The maximum acceleration and top speed were adjusted for each motor until the system curve was within the continuous region. The results are shown in Figure 63 below.



(a)



(b)



(c)

Figure 63. Possible system curves of the vehicle undergoing maximum acceleration up to top speed for each motor. (a) 70W Maxon motor (b) 50W Maxon motor (c) 30W Maxon motor

To put the performance characteristics in more perspective, an additional performance study was conducted. The top speed that each motor would allow while staying in its continuous region was scaled up, in order to be compared to the top speed of a normal road car. Additionally, the time it would take the car to reach the top speed of the motor is also spelled out. Finally, the prices of the motors and their add-ons are compared in the last two columns. The results of the performance study are shown in Table 16 below.

Table 16. Summary of Performance Parameters for Different Motors

Option	Parameters						
	Max. Acceleration [m/s ²]	Top speed [m/s]	Top Speed [comparative mph of a real car]	Time to top speed [s]	Price Motor [\$]	Price Add-Ons [\$]	Total Price [\$]
70W + internal	1.4	6.5	101.78	4.64	\$160.50	\$180.75	\$341.25
70W + external	1.4	6.5	101.78	4.64	\$160.50	\$37.40	\$197.90
50W + internal	1	6.5	101.78	6.50	\$147.13	\$180.75	\$327.88
50W + external	1	6.5	101.78	6.50	\$147.13	\$37.40	\$184.53
30W + internal	0.6	5.5	86.12	9.17	\$100.38	\$180.75	\$281.13
30W + external	0.6	5.5	86.12	9.17	\$100.38	\$37.40	\$137.78

The first option, the Maxon 70W motor with an internal encoder, is the motor that is currently on the vehicle. Its high level of performance and the high-tech internal encoder both contribute to its price of \$341.25 per motor. We would like to lower that price without sacrificing too much in the name of performance.

It is important for the vehicle to model the performance of a real vehicle. Students of the course will be implementing stability control algorithms, and for them to be able to test their designs, it must be possible for the vehicle to become unstable. If the vehicle is not capable of reaching high enough speeds, testing for instability will be difficult to accomplish.

As seen in Table 16, the performance and price of the 70W and the 50W options are very similar. It was for this reason that we decided not to consider the 50W motor. After considering the performance implications, it was decided that it would be best to keep the 70W Maxon motor. To reduce cost, future vehicles will be implemented with the 70W motor without the internal encoder. We will not replace our current motors with new 70W motors without the internal encoder, but we will purchase the external encoder for testing purposes.

5.2 Vehicle Model

As explained in Section 2.1, the vehicle model that was developed is based upon the Bicycle model, a simplification of the dynamics of a car in which it is assumed to be two dimensional and certain driving resistances are neglected. This vehicle model will be used to simulate the vehicle's response to certain inputs and physical situations, lending students insight into the dynamics of a car. It will also be used as a tool for controller design, where it will act as a simulation of the plant and allow students to monitor how certain dynamic variables such as lateral acceleration and yaw rate respond to their controllers.

The inputs to the model are the steering angle input and the voltage and current inputs to the motor. The model outputs steering angle at the wheel, motor speed in RPM, vehicle speed, vehicle lateral and longitudinal accelerations, and yaw rate and yaw angle. The model uses the equations of the bicycle model introduced in section 2.1. The motor is modeled using a linear DC motor model introduced in section 2.1.

In order for the model to predict the behavior of our car, some parameters on our vehicle needed to be measured. These included the location of the center of gravity, the steering behavior, the cornering stiffness, and the inertia about the center axis. The longitudinal location of the center of gravity was found using four scales at each of the wheels of the vehicle. The weight distribution between the front and rear axles can be used to find the longitudinal location of the center of gravity using

$$l_f = L \left(\frac{R_r}{W} \right),$$

where l_f is the distance from the front axle to the center of gravity, L is the wheelbase of the vehicle, R_r is the reaction force at the rear axle, and W is the total weight of the vehicle. To find

the height of the center of gravity, the front axle of the car was raised by a known height. From this, the angle of incline is determined. The weight on the front axle is then measured—the weight will have changed from when the car was not on an incline. Techniques from statics are then used to find the height of the center of gravity, using

$$h = \left(l_f - L \left(\frac{R_{fnew}}{W} \right) \right) \cot(\theta) r ,$$

where R_{fnew} is the new weight on the front axle, θ is the angle of incline, and r is the radius of the wheels.

The steering behavior was found by inputting a range of steering gains and measuring the output steering angle. The steering command inputs were over the range of -400 to 400. These values represent position inputs to the steering servo, that were then converted into steering angle inputs by the servo. The steering angle output at the wheels was measured using a protractor. The input and output steering angles were plotted for both of the front wheels. The plot is shown in Figure 64 below.

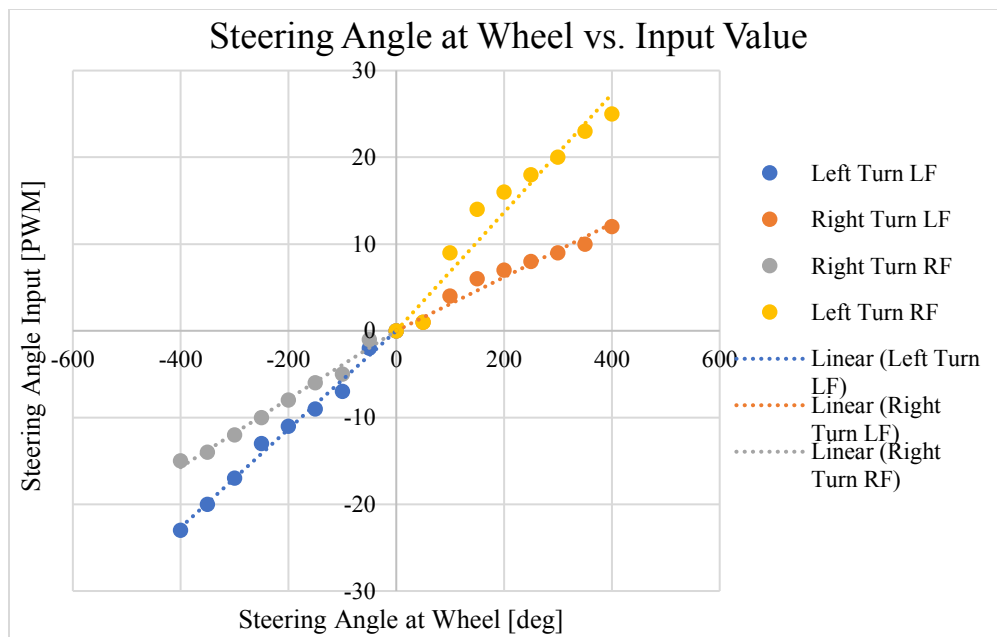


Figure 64. Steering behavior of the car over a wide range of input values for the left and right front wheels.

The linear curve fits will be used to predict the output steering behavior of the car based on an input steering gain.

One thing we noticed when testing for the steering behavior is that there is quite a bit of play in the steering. The way the steering mechanism is designed allows the wheels to move approximately three degrees in either direction even when the steering servo is locked in a position. The amount of play could potentially affect our control of the vehicle. If it does, we

either have the option of attempting to model the play in our control or attempt to eliminate it by fixing the mechanical components of the steering. We also through this test confirmed that we have Ackermann steering. With Ackermann steering, both steered wheels rotate about the same point, so that while cornering they are not parallel with one another. This is so that the vehicle does not slip while cornering when the outside wheel travels further than the inside wheel. What this means in terms of steering response is that the slopes of each of the four lines will be different, and the steering will exhibit nonlinear behavior. Linearizing this for design purposes will cause discrepancy between prediction and observation.

The vehicle cornering stiffness proved to be very difficult to model. Cornering stiffness is the ratio of the degree of slip in a corner to the amount of lateral force transferred. In the International System (SI) Units, it is measured in Newtons per degree. In order to accurately obtain the cornering stiffness, a test would have to be performed with a customized dynamometer that could measure lateral force transfer at different degrees of slip. Because such a test rig was unavailable to us, we had to estimate the cornering stiffness. We used an estimation method that utilized the dimensions of the tire and some properties of the tire's construction. Even using this method, some parameters such as the compression modulus of the tire and the belt thickness had to be estimated due to a lack of empirical data for RC car tires. We were, however, able to get an estimated value of the cornering stiffness and will continue to refine it once we begin testing the vehicle [Hewson].

The inertia about the center axis of the vehicle was measured from the SolidWorks model. This method was selected because it will be simple to update the value as changes are made to the vehicle.

The vehicle model is shown in Appendix J.

5.3 Sensor Integration

In section 4.3, multiple sensors were presented and discussed as potential options to be integrated onto the vehicle. A final selection was made based on criteria including: cost, relevance in industry, and ease of integration. The list of sensors to purchase and integrate was narrowed down to the LiDAR, the Indoor GPS, and the Bluetooth module. These sensors now join the time of flight rangefinder, IMU, ultrasonic sensor, and camera, as the combination of sensors available for usage on the car. The IMU, ultrasonic sensor, camera, and time of flight rangefinder are all fully integrated on the vehicle. The LiDAR, Indoor GPS, and Bluetooth, however, are still in the process of being implemented. What follows is a discussion of the preparation that has been made to interface with the sensors along with the testing that is planned once the entire sensor interface has been established. Although the firmware may be transitioned to being run on the Raspberry Pi, a lot of the planning was made assuming the Teensy would be used. Fortunately, with regards to what is described in this section, not much would change if the firmware is ported over to the Raspberry Pi.

In Table 17 below, the data, usage, communication protocol, and implementation status associated with each sensor is identified. The status is labeled as either: tested, planning, or implemented. "Implemented" refers to the sensor being currently functional on the system;

“Tested” refers to the interface and data output of the sensor having been tested; and “Planning” refers to the sensor interface currently being prepared.

Table 17: Description and status of the sensors.

Sensor	Data Output	Usage	Comm. Protocol	Status
LIDAR	Distances to points 270° around car (r, θ) [m]	Obstacle Avoidance, Obstacle Map Generation	Serial UART or USB	Tested; Planning
Time of Flight-Range Finder	Distance to Closest Object (high resolution) [m]	Obstacle Detection	I2C	Implemented
IMU	Linear Acceleration [m/s ²], Absolute Angular position [rad], Rate of Angular Rotation [rad/s ²]. (all 3-axis)	Acceleration, Estimate Position, Yaw Rate, Heading Angle	I2C	Implemented
Indoor GPS + IMU	Position (x,y,z) [m] & IMU (same as above)	Absolute Position and Velocity (from GPS) & IMU (same as above)	Serial UART or USB or SPI	Planning
Ultra-Sonic Sensor	Distance to Closest Object (low resolution) [m]	Obstacle Detection	Analog	Implemented
Camera	Location of Lanes, distance to lanes (x,y) [m]	Lane Line Detection, Velocity of Vehicle*	MIPI CSI-2	Implemented

The first task in preparing sensor integration is managing the communication protocols. The communication protocols that are going to be required for the new sensors are: serial UART and SPI. The Teensy currently has both of those communications implemented for other sensors. Figure 65 below shows the current ports being used on the Teensy (marked in blue) and the ports that will be needed for the sensor additions (marked in red).

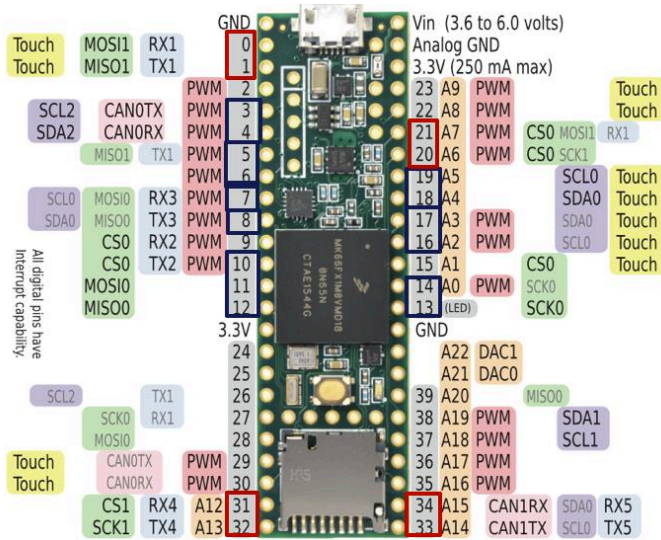


Figure 65. The Teensy 3.6 Pinout showing the currently utilized pins (marked blue) and the ports that will be needed for the new sensors (marked red). [<https://www.pjrc.com/teensy/pinout.html>]

5.3.2 LiDAR Sensor

Similar to the Bluetooth module, the motivation of implementing a LiDAR sensor on the car is to provide students with more potential functionality for their term project. The car currently only has the ultrasonic sensor and the time of flight rangefinder. Both of these give single readings of distance. These two sensors have their benefits and tradeoffs. The ultrasonic sensor has a larger detection zone than the time of flight sensor, so if there is an obstacle somewhere in the front right corner of the car, it is possible that the ultrasonic sensor will pick it up. The time of flight sensor, however, has a much smaller detection zone. It only senses obstacles that are in the line of sight of the sensor. Although the ultrasonic sensor can sense an obstacle around it, it cannot pinpoint the exact location of the obstacle. For example, if there was an obstacle that was in front and to the right of the car, the ultrasonic sensor would detect it, but it would not know whether the obstacle was to the right, straight ahead, or to the left of the sensor. Because the ultrasonic sensor has a small detection zone, if it detects an object it knows that that object must be right in front of the sensor. The LiDAR is the most robust of the three sensors. It outputs distances as a function of the angular position, with a resolution of $.36^\circ$. Because of the high resolution and relatively high range (4m), the LiDAR outputs a large amount of data that can be difficult to process and store.

Although the communication interface (serial) is very straight forward to get started, storing and processing all the data that the LiDAR produces will be a challenge. We have not yet interfaced the LiDAR sensor to the Teensy, but we have tested the LiDAR sensor using a different microcontroller. We were successfully able to get readings and store the data. The data was stored in a matrix grid. Instead of storing absolute readings (Ex: 2100mm) in the matrix, only 1's and 0's were stored. Each element in the matrix, depending on what row and column it was on, corresponded to a location in the 2D ground plane. Because the location of the obstacle is encoded through its placement in the matrix, the element then was simply a 1 representing an obstacle at that location or a 0 representing no obstacle at that location. For example, if the LiDAR was in a 20 foot by 20 foot room and we wanted a grid resolution of 0.5 foot by 0.5 foot, there would be 1600 elements in the matrix (40 rows by 40 columns). If the matrix was fully populated and we wanted to determine whether there was an obstacle at point $(x=10,y=15)$, we would check the element value at row $x=20$ and column $y=30$. If the vehicle knows its location in the grid matrix, it then becomes simple to perform obstacle avoidance and navigation. The concept just described is shown below in Figure 67. The left image shows the collection of obstacles that the LiDAR was placed in front of. The right image is the matrix grid generated from the scene on the left. The 1's (obstacle locations) on the matrix grid align with the obstacle locations on the image. Note that there are 0's behind the 1's on the grid even though there is a wall behind the obstacles in the image. Because the LiDAR cannot see behind the obstacles, it assumes that the area behind the obstacles is open. If the vehicle were to travel around the obstacles, it would then detect the wall and the matrix grid would be updated to reflect the new known obstacles.

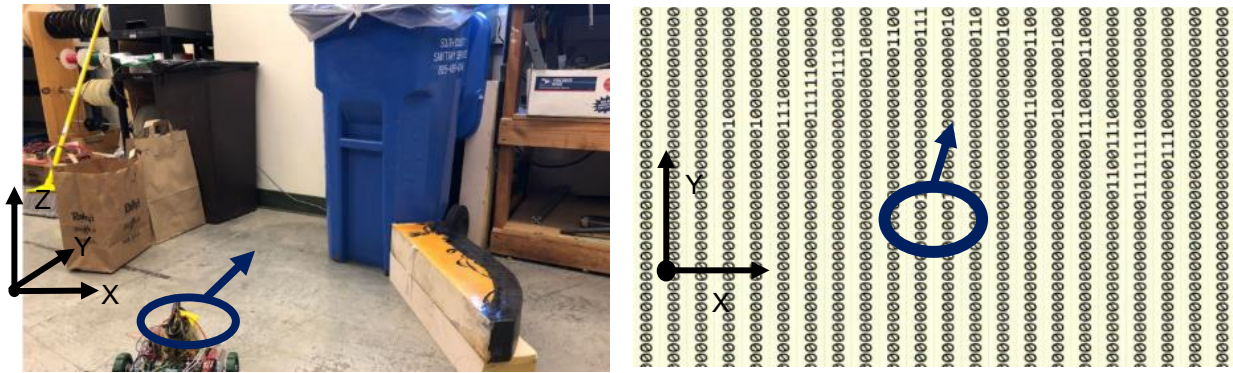


Figure 67: Testing done on the LiDAR that generated a “map” of obstacles

The next steps for the LiDAR is to integrate it into the current system. As mentioned before, it is unknown how the large quantity of data being transmitted per Serial message will affect the Teensy’s performance, in the sense of taking processing time that would have otherwise been used for other necessary tasks. Once data is being retrieved from the LiDAR, we will look into the best way to make the data available to the students to use on the Simulink model. After that we would like to have work on an algorithm demonstrating the car avoiding some obstacles by utilizing the grid matrix.

The last sensor that needs to be integrated is the external encoder. The external encoder, shown below in Figure 68, outputs a quadrature signal that is used to determine how much the motor shaft has rotated.



Figure 68: The external encoder manufactured by CUI. The mounts will help with fixing the encoders to the form of the car.

[<http://www.cui.com/product/motion/rotary-encoders/incremental/modular/amt11-v-kit>]

The challenge with integrating this sensor is not the communication or data interpretation, but the physical integration– figuring out how to mount the encoder. These sensors will not communicate directly to the Raspberry Pi or the Teensy, but will instead be read by the new motor drivers being designed by Charlie Refvem. Since the motor housing is 3D printed, we are planning on putting some threaded inserts in the motor housing and then fastening the encoders to the motor housing using the mounts on the encoder. Figure 69 below shows the idea of mounting the encoders on the motor housing.

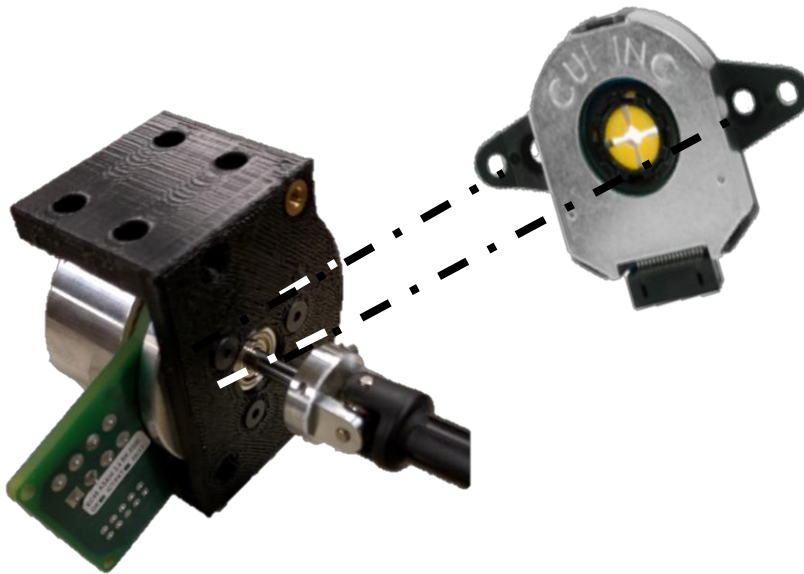


Figure 69: How encoder may be fixed to vehicle.

5.4 Control Algorithm Structures

In this section, we will present three control algorithm “skeletons” for algorithms that will be implemented on the vehicle. These skeletons include the input and output signals and the sensors that the signals will come from, as well as any kind of manipulation that will be necessary to generate those input and output signals. These three control algorithms will be completed in the coming few weeks, to align with a project in our AERO 553 Advanced Control Theory Class.

5.4.1 Adaptive Cruise Control

The Adaptive Cruise Controller (ACC) was developed for the vehicle using equations found in chapter 6 of Vehicle Dynamics and Control [Rajamani]. This is a change from the adaptive cruise control algorithms discussed in section 2.6.2 as we found Rajamani to be a more reliable source. For the Adaptive Cruise Controller, only the design of the controller was completed, the “manufacture” and testing of the implemented controller fell out of scope due to time constraints. In this section is presented the theoretical design and simulation of and Adaptive Cruise Control algorithm for eventual implementation on the SSIV.

The main design constraint we focused on for the controller was that it had to maintain string stability. According to Rajamani, “the string stability of a string of ACC vehicles refers to a property in which spacing errors are guaranteed not to amplify as they propagate towards the tail of the string.” A string of vehicles all running the same ACC algorithm is shown in Figure 70.

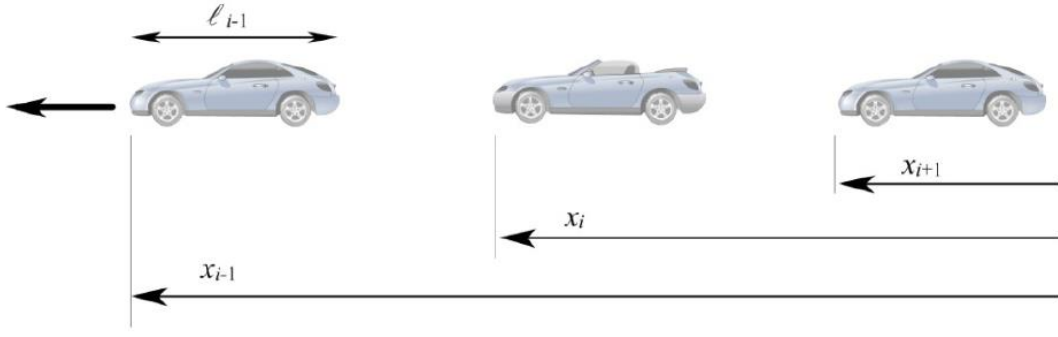


Figure 70. String of ACC vehicles and coordinate system [Rajamani].

The inter-vehicle spacing, ε_i , and its error signal (Spacing Error, δ_i) are defined below [Rajamani].

$$\varepsilon_i = x_i - x_{i-1} + \ell_{i-1}$$

$$\delta_i = \varepsilon_i + h\dot{x}_i$$

The constant headway, h , in units of seconds is defined as the time it would take for the vehicle to travel the desired inter-vehicle spacing for a given velocity of the ACC vehicle. When there is zero spacing error, the inter-vehicle spacing is minus the ACC vehicle's speed multiplied by the constant headway time. A constant headway policy is used because, unlike a constant spacing policy, it allows for string stability. The control law presented by Rajamani is shown below where h is the headway in seconds and λ is chosen by the designer in such a way that meets the string stability requirements at the very minimum.

$$\ddot{x}_{i_des} = -\frac{1}{h}(\dot{\varepsilon}_i + \lambda \delta_i)$$

The controller outputs a desired acceleration that is accepted by a first order approximation of the car's dynamics. The equation below shows that the car is assumed to have a first order lag in acceleration. The time constant was assumed to be 0.5 seconds in this paper but the real time constant for our system can be measured in an open loop torque step response – a system identification test.

$$\ddot{x}_i = \frac{1}{\tau s + 1} \ddot{x}_{i_des}$$

A Simulink model, seen in Figure 71, was constructed that closed the loop on velocity of the i^{th} vehicle in the string, given a velocity input from the preceding vehicle.

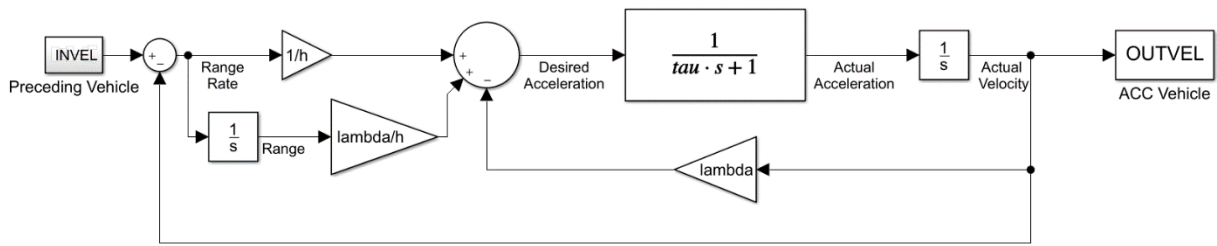


Figure 71. Simulink model for ACC controller simulation.

There are two conditions that are necessary and sufficient to ensure string stability. The transfer function shown below relates the spacing error of the i^{th} vehicle to that of the vehicle preceding it. The first condition is that the transfer function must always have a magnitude less than 1 to ensure that the energy in the spacing error diminishes toward the tail of the string. The second is that its impulse response must not change sign [Rajamani].

$$\frac{\delta_i}{\delta_{i-1}} = \frac{s + \lambda}{h\tau s^3 + hs^2 + (1 + \lambda h)s + \lambda}$$

To meet condition 1, the headway h must be greater than or equal to twice the time constant of the plant. A headway of 2 seconds was assumed desirable for this simulation, but a smaller headway may be used if improved traffic flow is necessary. A λ of 0.135 was found through trial and error to satisfy condition 2 and produce the desired dynamic behavior of the system, i.e. zero steady state spacing and velocity error.

The impulse response for the chosen λ is shown below in Figure 72. The sign of the spacing error did not change in response to a unit impulse input.

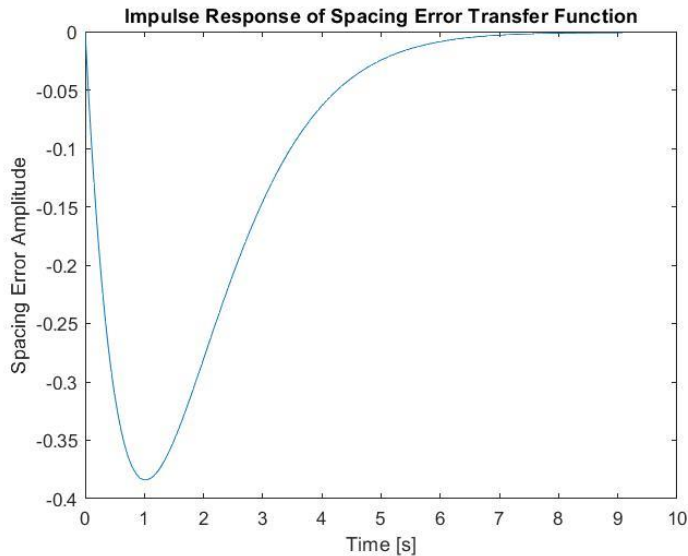


Figure 72. Spacing error impulse response of i^{th} vehicle in the string.

The simulation of Figure 73 mimics a preceding vehicle moving at a constant velocity for 35 seconds, then accelerating at 0.5 g's for 5 seconds and then moving again at a constant rate. Initially, the ACC vehicle is going over twice as fast as the preceding vehicle but slows down to its speed within 30 seconds. In this initial situation, the ACC vehicle actually sped up to reach the desired inter-vehicle spacing of 14 meters. This is not a desirable characteristic and gives one motivation for a transitional controller that knows when to activate the spacing error control, maintain cruise control speed, or brake as hard as possible to avoid a crash [Rajamani page 157 through 163]. Without a transitional controller in this situation, the car would calculate that the range to the preceding vehicle is very large and would receive a control actuation dominated by this term. So, despite the preceding vehicle moving much slower, it speeds up to close the gap (potentially in violation of the speed limit). A transitional controller would maintain the cruise control speed it initially had and transition to the headway controller when the range was sufficiently small.

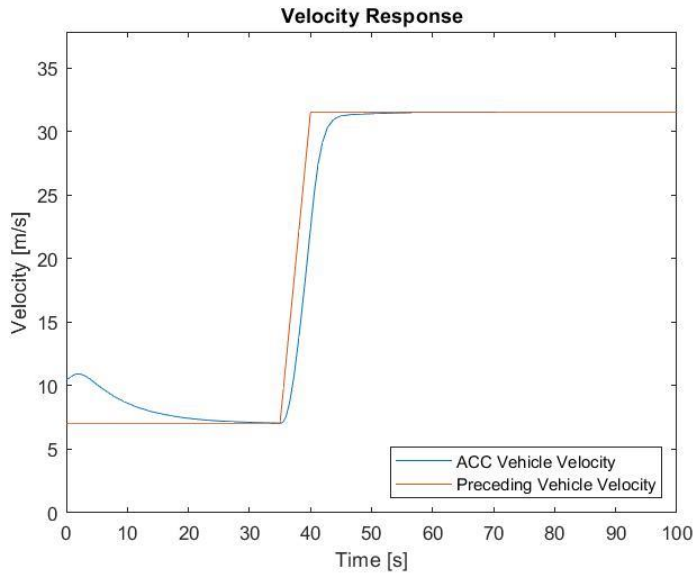


Figure 73. Velocity of i^{th} car in response to a preceding vehicle velocity profile.

As the preceding vehicle accelerated, the ACC vehicle followed suit with a steady state velocity error until the preceding vehicle slowed to a new steady state speed. When this happened, the ACC vehicle matched its speed once again. For both situations, the car reached close to the desired inter-vehicle spacing, as seen in Figure 74.

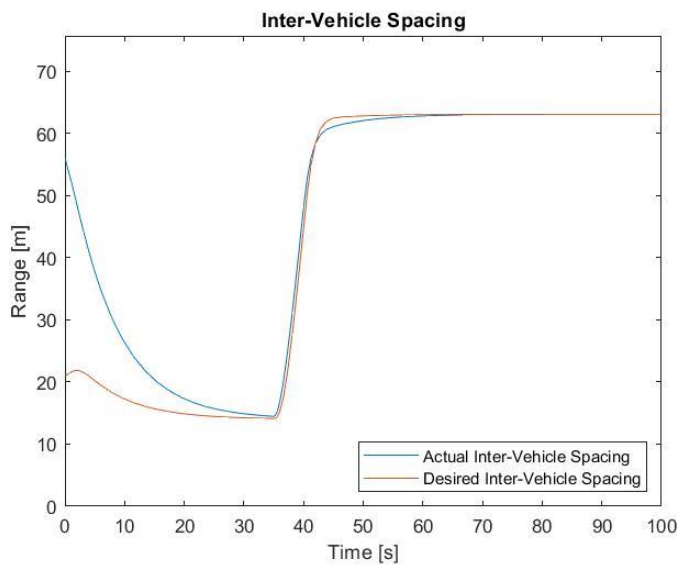


Figure 74. Actual inter-vehicle spacing (the range to the preceding vehicle) and desired inter-vehicle spacing of the ACC vehicle in response to the preceding vehicle velocity profile of Figure 73.

Choosing λ was somewhat arbitrary as there were many selections that could have produced reasonable behavior with varying levels of oscillation and timing characteristics. We did find that certain magnitude extremes provided undesirable behavior, as seen below in Figure 75.

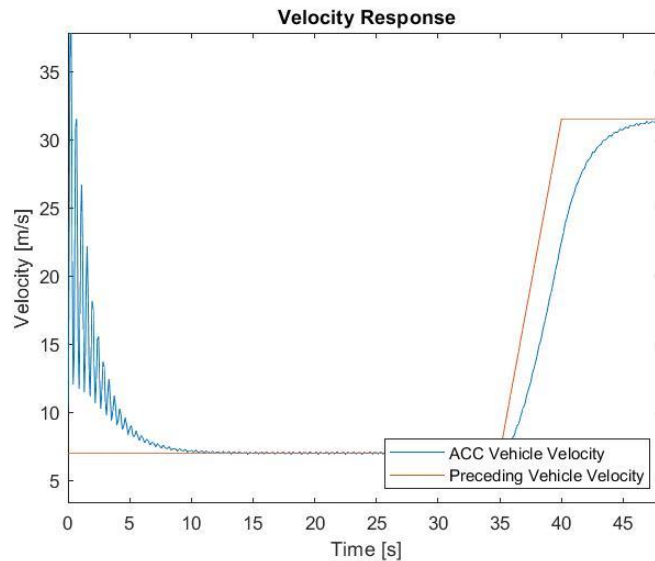


Figure 75. Velocity response of ACC vehicle under same conditions as Figure 73 but with $\lambda = 100$.

Figure 76 shows that a much smaller λ produces a steady state spacing error. This is effectively zeroing out certain important terms in the control system. Both λ 's met the string stability requirement but were not good choices. This points to future work that can be done to select a more optimal λ and will be discussed in section 9.2.

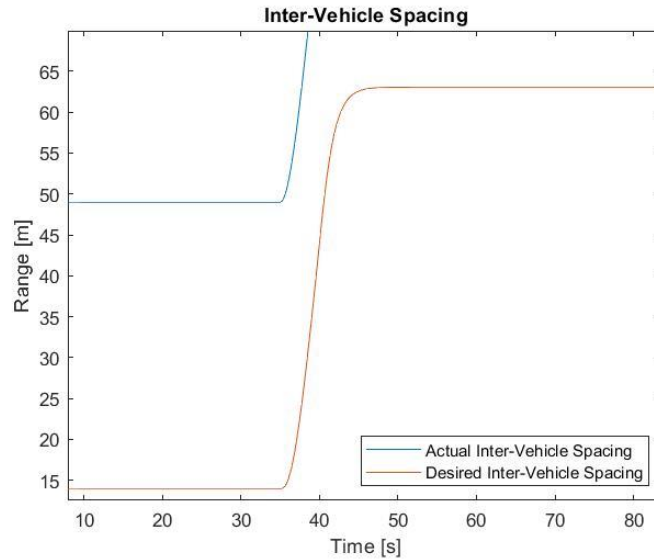


Figure 76. Inter-vehicle spacing vs desired inter-vehicle spacing of ACC vehicle under same conditions as Figure 73 but with a λ of 0.000001.

5.4.2 Lane Keeping

For the lane keeping controller, there are two main components: the actual controller and the lane detection block. The lane detection block takes in a stream of images from the raspberry pi camera. The next step is then to crop the image to remove the top and other areas where lane lines are not expected to be found. The current method detects the lane lines at a look-ahead distances, L . The combined detection of the lane segments at each look ahead distance is shown in Figure 77.

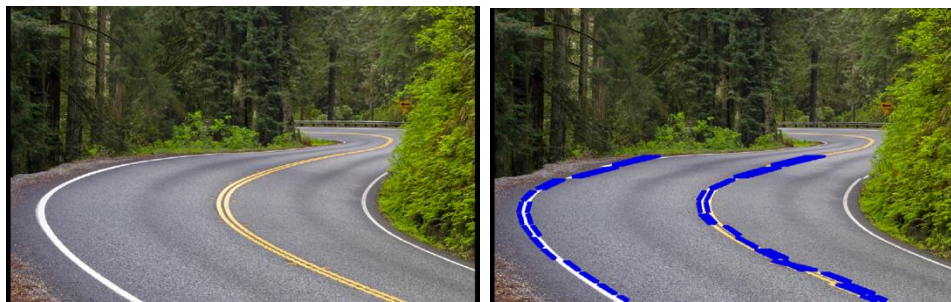


Figure 77: Example image of lane where left image shows the input image and the right image shows the series of lane lines detected that together form the curved lane line

The next step in the lane detection algorithm is to just limit the search of lane lines to regions of interest (ROI) by estimating the location of the lane line in row r_{n+1} based on where the lane line was in row r_n and the direction of the lane line. This will significantly reduce the amount of computation being done by the algorithm.

Another task that needs to be done in the lane detection algorithm is transforming each row and column of pixels to a x and y point on the ground plane. This transformation makes the assumption that the ground is flat, but it is the only way determine relative distances in the image without a stereo camera. The transformation will be accomplished by calibrating the camera. For example, an image can be taken of a paper with a grid of fixed square sizes. If we know that size of the grid, we can determine the relative distances of the pixels based on which grid square the pixels fall into. One thing to note– the transformation of pixel row and column to Euclidean space is not linear. For example when one looks at a rail road track continuing on into the horizon of sight, it appears as if the left track and right track converge, but they do not.

The location and direction of the lane line is used to generate a desired yaw rate. Specifically, the offset length from the car to the lane line and the angular difference between the lane curve tangent and the vehicle heading angle are used to determine a desired yaw rate. Based on the difference between the measured yaw rate and the desired yaw rate an error signal is produced. The error signal is fed through a PID controller and gets converted into an input steering angle to the plant. The lane keeping controller Simulink diagram is show in Figure 78 below.

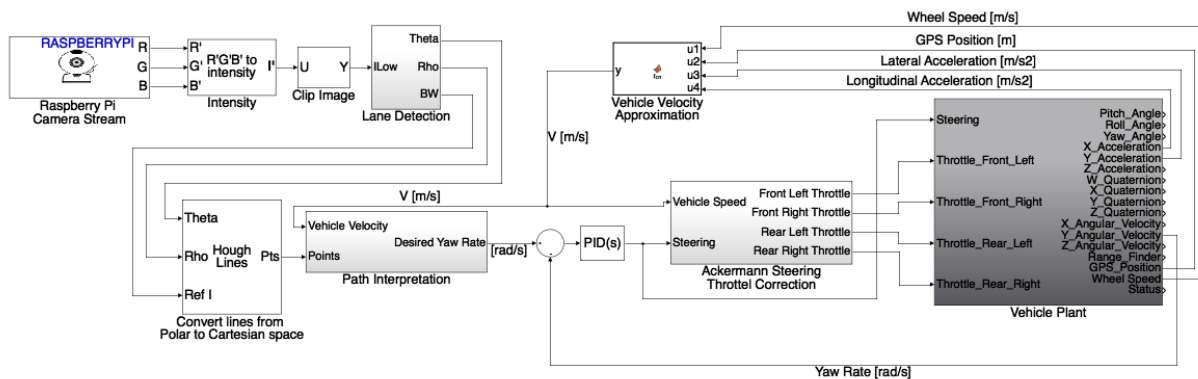


Figure 78: Preliminary block diagram of lane keeping algorithm.

The design of the block diagram was drawn from the work of Taylor [J. Taylor et al.]. Taylor uses full state feedback and the paper also describes an observer that was used to determine the curvature of the road, which manifests itself as a disturbance in the model. Additionally, this paper describes a few other controller design alternatives including a lead-lag controller and a linearized input output controller. We believe the work in the paper will be a good foundation to draw from in the development of our lane keeping algorithm because the approach described aligns well with the topics that will be discussed in the course.

The next steps in the controller design is to have an optimized lane detection algorithm that can produce a desired yaw rate. Also, more research will be done into understanding the observer design so that it can be implemented on our controller as well.

5.4.3 Stability Control

The stability controller went through a number of design changes from the original idea for the controller, due to unforeseen complications of the design. This section will enumerate each of the design changes, the reasons for them, and the process that we went through to end up at the final design.

As described in section 2.6.4, the original design for the stability controller was approximately “half” of the kind of stability controller that you would find in a car that you buy today. A car has a stability controller and a traction control: the stability control regulates the yaw moment of the car, and the traction control regulates the slip of the tires. Due to limitations with what we could do with our tire modeling, we decided to only attempt to implement yaw moment regulation.

Yaw moment regulation can be accomplished in a couple of different ways: by modifying the power delivered to the wheels, and by providing a steering input. Because the SSIV has one motor per wheel, it is capable of providing different amounts of power to each wheel, and therefore is very well set up for the former method. This method of varying the power delivered to each wheel, called “torque vectoring,” was the method in mind when the controller was designed as described in section 2.6.4.

After attempting to mate the controller design with what we were able to get from different sensors (mostly the Inertial Measurement Unit, IMU), we realized that it would take a master’s degree in controls and lots of experience in vehicle dynamics to implement torque vectoring. Among other problems, some of the problems we incurred were:

- Lack of a 4-wheel vehicle model
- Lack of in-depth (non-linear) tire modeling
- Wheel speed data not available
- Needed nested controller design

To the last point, we realized that torque vectoring would require a whole other level of control. Multiple nested loops would have to ensure that the torque commands for the stability control didn’t cause the vehicle to drastically change speed. Essentially, we would need cruise control to be a part of the stability controller.

For the reasons listed above, we decided to try to implement yaw moment regulation using the other method: steering control. This method is called “steer-by-wire” because the controller commands a steering angle.

When designing a controller for a real system, one has to make sure that the states that you are trying to control, i.e. position, speed, acceleration, etc. are available via your sensor feedback. One cannot control position if there is no way to measure it. So when designing the steer-by-wire yaw controller, we had to design around the states that we could measure. Many steer-by-wire systems found in academic literature use the vehicle’s steering angle as one of the states. We have no way of measuring that, so we couldn’t attempt to control it. We finally settled on a control scheme presented in *Vehicle Dynamics and Control* by Rajesh Rajamani. Rajamani lays out a control scheme that uses the basic bicycle model, laid out in section 2.3.1, and lateral vehicle dynamics to arrive at a linear state space vehicle model. The states of the model are error

states: error in position and error in orientation. The model also includes as states the derivatives of each of these—the derivative of the positional error being lateral velocity error and the derivative of orientation error being yaw rate error. By setting the model states to error states, Rajamani turns a set point control problem, regulating to a desired, non-zero set point, into a regulation problem, where the errors are regulated to zero. The derivation of this state space model can be found in pages 27 – 37 of *Vehicle Dynamics and Control*.

The rest of the control strategy is a simple state feedback controller with an LQR gain. A simplified block diagram is shown below in Figure 79.

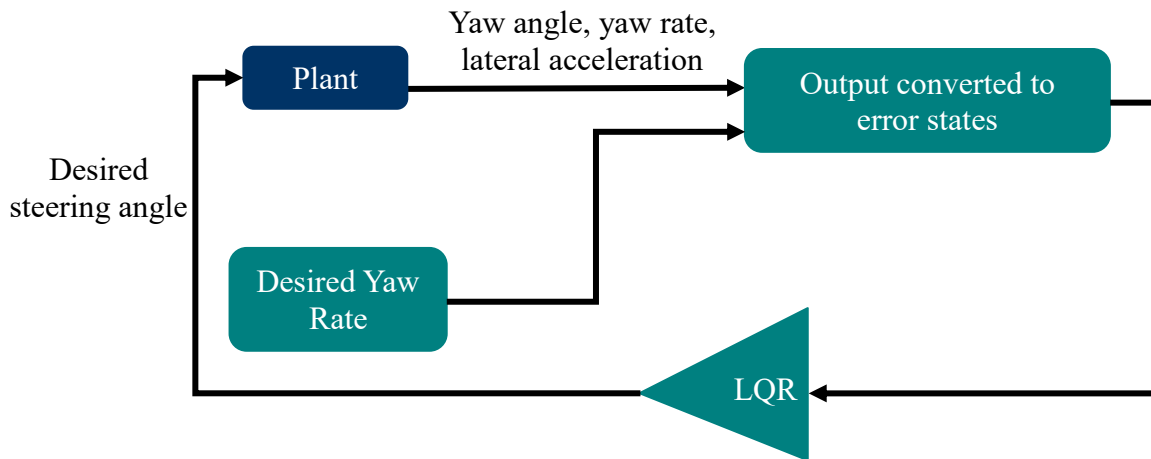


Figure 79: Simplified block diagram of stability controller.

The LQR gain was calculated as described in section 2.6.4. Section 6.3.2 describes more in-depth the implementation of this controller, including the modifications that had to be made to it to fit the sensor data that was available. Section 7.3.2 describes the testing procedures and results. The final controller design used in simulation is shown in Appendix K.

5.5 Firmware Design

Several firmware design decisions had been made between the Preliminary Design Review and the Critical Design Review. We made decisions regarding the computer hardware and software architecture of the vehicle and created outlines for some key firmware elements. Sections 5.5.1 and 5.5.2 describe the path we were going to take to accomplish the software goals of the project. Since the Critical Design Review, however, changes to the firmware design were made. The need for these changes was made apparent once we began getting into the details of the firmware “manufacture” where certain design requirements and time constraints were made apparent. Sections 5.5.3 on describe the design that is currently implemented on the SSIV. These sections will reframe the big picture software and hardware design elements and provide justification for the manufacturing and testing that took place to create the final product.

5.5.1 Teensy to Raspberry Pi Firmware Changeover

We have been considering changing from a combination of the Raspberry Pi and Teensy microcontroller to a Raspberry Pi only design. Based on conversations with Charlie Refvem who

worked closely with the uLaren team, and based on their Final Design Review [Miley, Phillips, and Grant] the uLaren team chose to use the Teensy 3.6 for several reasons. One was that it can communicate to the Maxon motor drivers using CAN protocol, while the Raspberry Pi cannot. Another is that they wanted the potential to interact with analog hardware. Lastly, they wanted an Arduino to allow for easy programming of the firmware by those with little programming experience. Our situation is different now; we are moving away from the Maxon motor drivers and the CAN protocol, we will only need to interact with digital sensors that the Raspberry Pi can interface with, and we want to move away from the Arduino environment so that firmware can be programmed in a variety of languages (including Python which is comparable in difficulty to Arduino). Also, we think the Teensy is adding unneeded complexity to the system for what it is accomplishing. Using just the Raspberry Pi would simplify the system, allow for flexibility in firmware design, and according to Mathworks support staff [Stabile], very likely serve our purposes.

There are still some uncertainties with our ability to switch over the entire firmware. We will have to learn how to navigate the Raspbian Operating System using the code we write, since we do not want the user to have to interact with Raspbian at all. We will also likely have to learn how to communicate between files running in parallel. If we want the firmware to communicate sensor data to a Simulink executable, we would have to learn a communication protocol that can facilitate data transfer between different files on the same computer. It may turn out that switching to the Pi entirely proves to be very difficult and take away from our ability to develop the desired vehicle capabilities.

We have decided to compromise and keep the Teensy for now. The Teensy firmware currently functions well enough to test our controller designs and sensors, which we be able to start doing immediately. At the same time, we will implement a Raspberry Pi official touch screen running Graphical User Interface (GUI) software, see Figure 80. We will design Firmware that creates a menu system and allows the user to select and run uploaded Simulink files. This will give us the opportunity to better understand the difficulties associated with the switch and allow us to gain experience in interacting with the Raspberry Pi without having to commit fully to using it alone.



Figure 80: Raspberry Pi official touch screen [Raspberry Pi].

5.5.2 Firmware Architecture Design

The firmware architecture we will be adopting on the Raspberry Pi is a task and state machine scheme. There will be a main loop called “main” that runs every task in the firmware each time through the loop. Each task can only be in one of a finite number of states at any given time--this is what it means for a task to be a finite state machine. The state that a task goes to each time through main is dictated by the state variable. For example, if the state variable is 5 when the loop arrives at that task, it will enter state 5. Transitions between states of a task are decided in a state transition diagram. Transitions are encoded as arrows, and states as blocks. Over each arrow is written the condition necessary for the transition and what is done as a result. The state transition diagram of our current task, Mastermind, is shown Appendix L. For the user interface task shown in the figure, the state machine framework works well for establishing the different menu screens and the inputs the user has available to them.

From within this framework, dynamic tasks can be written to complete a process incrementally, taking a small piece of the processing time available with each pass through main. A task meant to acquire the state feedback vector for the stability control lab might have a different state for each sensor needed, gathering them one at a time as quickly as it can but not bogging down other tasks by attempting to retrieve the vector all at once. While this scheme does not guarantee that a process happens precisely when it is desired like a RTOS would, it is appropriate for soft real-time applications. In our case, the high-level control of the vehicle dynamics will likely require a 100Hz control frequency maximum.

5.5.3 Firmware Design Updated

The Teensy to Raspberry Pi firmware changeover of section 5.5.1 described a complete removal of the Teensy 3.6 from our system. The final system incorporates it heavily, in fact the Teensy

played a vital role in our ability to run the control systems we created. This return to the Teensy took place for a few reasons:

1. Switching away from the Maxon motor controllers became unfeasible within our timeframe. To test control algorithms before the end of the year, CAN communication to these controllers was an absolute necessity, and the Pi does not have this capability.
2. A microcontroller unit like the Teensy was found necessary because the system must actuate a servo motor with a precise pulse-width modulated (PWM) signal, and it must read PWM signals from the receiver (the associated transmitter, or radio is handheld and allows for manual steering and throttle commands to be sent to the vehicle). The Teensy 3.6 is well suited to measure real time signals while the Raspberry Pi is not.
3. The Teensy adds an additional abstraction layer to the data acquisition making it easier to read from each sensor as desired.

We decided to change the role that the Teensy played in the system quite a bit. Rather than having high level functions such as dictating some operational modes of the vehicle as it had at the beginning of the year, it now acts as sort of sensor/peripheral hub. The Teensy firmware is responsible for continuously gathering sensor data, actuating peripherals, executing commands sent from the Raspberry Pi, and providing a dead-man failsafe. The Serial Peripheral Interface (SPI) communication protocol that the Pi and the Teensy use to communicate took, unexpectedly, a majority of the firmware manufacturing time for the final product.

The Raspberry Pi's functionality has also shifted. We have still moved much of the high-level operations over to the Raspberry Pi but in different ways than previously described. We are providing a screen-based user interface as before, but we have moved away from the Raspberry Pi official touch screen to a simpler 2x16 character LCD screen from Adafruit. This UI is facilitated by Raspberry Pi firmware but is not required for data collection. The data acquisition on the Pi is done solely by the executing Simulink file. It does this using SPI blocks available in the Raspberry Pi Simulink Support package. Simulink subsystems were designed that allow easy linking of a control system to its required sensors and actuators on the vehicle. The "manufacture" of these subsystems is discussed in section 6.

Figure 81 shows the new firmware architecture scheme and shows the relationship between the user's computer, the Raspberry Pi, the Teensy, and the sensors and actuators as discussed above. The next portion of the report will discuss this architecture in more detail.

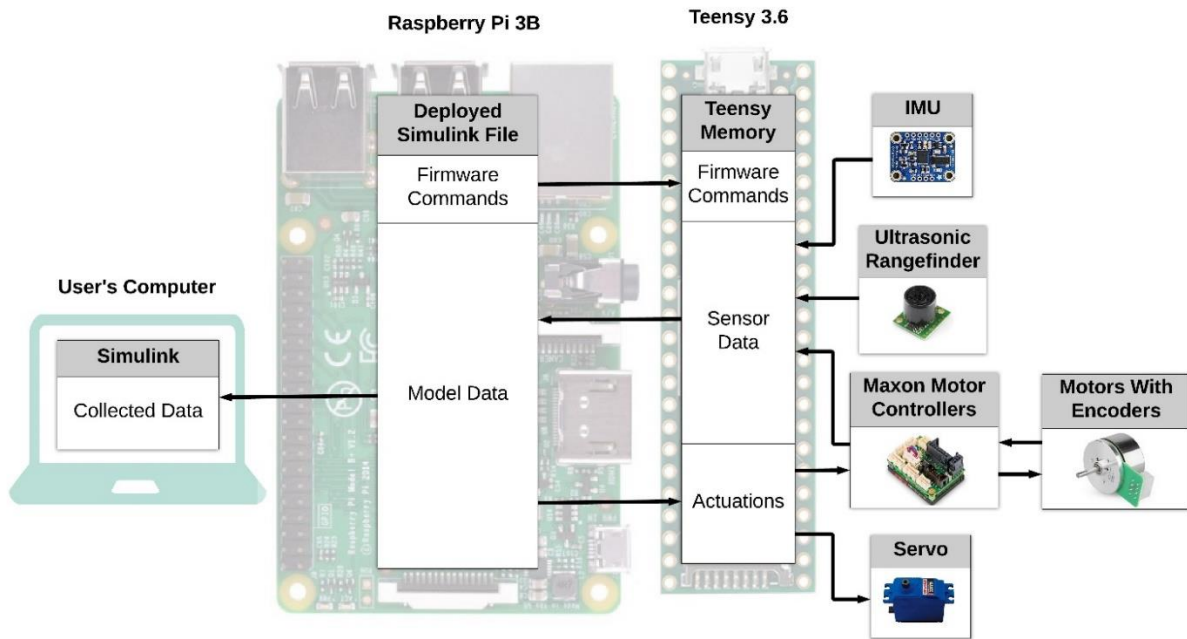


Figure 81. Firmware architecture diagram of final design.

User's Computer ↔ Raspberry Pi

The user's computer on the far left is where control systems are designed. The SPI communication subsystems are provided as Simulink files for use in the Simulink GUI and give the control systems access to the necessary peripherals. The Raspberry Pi runs the deployed Simulink file in either external or deployed mode:

- A model running in *External Mode* through the Raspberry Pi Simulink Support Package connects through Wi-Fi to the user's computer. In External mode, data can be streamed to scopes in the Simulink GUI for immediate viewing during or after testing. Parameters can also be adjusted while the model is running.
- A model running in *Deployed Mode* has no connection to the user's computer. This method is facilitated mainly by the LCD screen and Raspberry Pi firmware.

For both, the user must, at least once, connect Simulink to the Raspberry Pi through Wi-Fi by entering the IP address, username, and password of the Pi. In external mode, the user will design their control algorithms around the communication subsystem provided, attaching scopes or "to workspace" blocks to any signals they are interested in collecting data on. To run the Simulink model, external mode is selected at the top of the Simulink GUI (along with a few other necessary setting configurations) and the green play button is pressed to run the model. The code will compile and upload itself to the Raspberry Pi and while operational, the user will be able to view data and change parameters. For deployed mode, rather than press the green play button, the "deploy to hardware" button is pressed, and the executable file will be placed on the Pi in a

designated folder. From there, the model can be run through the LCD screen UI. Data in this mode must be stored in a .mat file and retrieved later for post processing and plotting in MATLAB.

Raspberry Pi ←→ Teensy ←→ Sensors/Actuators

From the Raspberry Pi's perspective, the Teensy is a block of memory that can be accessed through SPI read or write messages. 128 bytes of the Teensy's Random Access Memory (RAM) is allocated to storage of important information. The information is split into intuitive fields called registers. The Teensy gives the Raspberry Pi access to functions and peripherals through these registers. The registers currently fall under three categories and are labeled in Figure 81 under the label "Teensy Memory". These categories are:

1. Firmware Commands
2. Sensor Data
3. Actuations

The *firmware commands* can be thought of as function calls available to the SPI master device, in our case the Raspberry Pi. The Pi writes a non-zero value to the command register and the associate code on the Teensy will run. For example, writing a non-zero value to the `init_motor_controllers` register will prompt the motor controllers to initialize.

The *sensor data* registers are filled through whatever communication protocol is associated with the individual hardware peripherals. For instance, the Inertial Measurement Unit communicates with the Teensy through I²C. The Teensy will use I²C communication protocol to gather the latest IMU values and store them in the appropriate registers. If the Pi needs to know the most recent z direction yaw rate, it reads two bytes of data (the yaw rate is a 16-bit integer) from the registers at the `gyro_z` address.

The *actuations* are accessed by the Teensy to update the motors and servo setpoints. For example, when running, it will continuously retrieve the motor actuation values from the associated registers and update the Maxon motor controllers' velocity setpoint through CAN communication protocol. The Raspberry Pi updates these actuation registers through SPI write messages.

The "manufacture" of the Teensy firmware, focusing mainly on the SPI communication code, can be found in section 6.

5.6 Bill of Materials and Budget

Because most of this project is a software design project, there was not a lot in the way of purchasing parts. As discussed in section 5.3, the only sensor that was implemented was the indoor GPS sensor. This, along with an LCD screen for the user interface, were all of the parts that were purchased. Table 18 below shows the total cost of all proposed purchased parts for this portion of the project.

Table 18. Proposed Bill of Materials

	Description	Item Value [\$]	Quantity	Amount [\$]
Income	Project funds	2000	1	2000
			Total Income	2000
Expenses	Indoor GPS Sensor	429	1	429
	LCD Screen	19	1	19
			Total Expenses	448
			Extra	1552

With a project budget of \$2000, our proposed total purchases are well within our budget. Table 19 shows the total cost of making the SSIV as it now stands from scratch.

Table 19. Total Bill of Materials to Make the SSIV from scratch

Total BOM				
Category	Item	Quantity	Price [\$]	Total Cost [\$]
Traxxas Slash	Traxxas Slash 4x4 4WD	1	499.95	499.95
Processors	Raspberry Pi 3B	2	39.95	79.90
	Teensy 3.6 Microcontroller	1	37.50	37.50
	4GB SD Card	2	9.95	19.90
Circuit Board and Parts	Custom Circuit Board	1	37.50	37.50
	Accessories	1	45.47	45.47
Battery	Battery	1	38.70	38.70
Hardware	Screw Terminals	1	7.01	7.01
	Female Header	1	0.68	0.68
	Fasteners	1	30.00	30.00
Raw Materials	Aluminum Round Stock	1	10.57	10.57
	Aluminum Bar Stock	1	32.46	32.46
	Aluminum Sheet Stock	1	14.58	14.58
	Nylon Sheet Stock	1	41.60	41.60
Motors and Motor Accessories	Maxxon EC45 70W Motors	4	160.50	642.00
	Maxxon Motor Controllers	4	540.00	2160.00
Sensors	Keyfob remote control button	1	6.95	6.95
	Keyfob remote receiver	1	4.95	4.95
	9 DOF Absolute Orientation IMU	1	34.95	34.95
	Maxbotix Ultrasonic Rangefinder	1	33.95	33.95
	Raspberry Pi Cam	1	29.95	29.95
	Pi Cam Cable	1	1.95	1.95
	Indoor GPS Sensor	1	429.00	429.00
	CUI Encoder	4	35.00	140.00
Total				4379.52

It is the hope that future work can be done to implement Charlie Revfem’s motor controllers, as well as some less expensive motors, in order to bring the total cost of the vehicle down. The Design Hazard Checklist for the final design is shown in Appendix M.

6 Manufacturing

Since this is mostly a software design project, the manufacturing process described in the sections below goes through the process we took to go from software design to implementation. It also includes sensor implementation and how we addressed challenges that came up along the way. Section 6.1 details the manufacturing of the firmware, section 6.2 details how the sensors

were integrated onto the vehicle, and section 6.3 details how the lane keeping, adaptive cruise control, and stability controllers were implemented.

6.1 Firmware Manufacturing

The main component of this is the state transition diagram of the firmware task we designed, shown in Appendix L. The diagram, along with a layout of anticipated functions, will be used to construct the actual code of the firmware for the Raspberry Pi user interface.

6.1.1 Raspberry Pi Firmware Tasks and Functions

Because of the firmware compromise we made the only task we designed is Mastermind, the task most closely related to the user interface. This along with some anticipated software functions are described below. As mentioned earlier, Dynamic task design will be left for a later time.

6.1.2 Mastermind Task

This task is responsible for managing the menu system and dictating which operations the Firmware is performing, i.e. setting the dynamic mode of the vehicle, prompting the initialization of Simulink executables, or requesting that experimental data be exported through USB. The states of this task are “Screenshots” of the menu system (set up by the Graphical User Interface software functions). Flags set in response to user inputs tell other tasks to run their functions. Mastermind also reacts to system errors halting vehicle operation and alerting the user. Many of the states in this task are not anticipated to run in parallel with other tasks even when more dynamic tasks have been designed as they only operate during the menu interaction. Below are the details necessary to construct the Mastermind task code.

General

1. Every time a state transition occurs, “Prompt” is set to 1 so that the next state of the task knows to run the GUI functions needed to produce the prompt the user sees on the screen. Once the prompt is printed, the variable is set to zero and the next time through the main loop, the current state will know not to run those functions again.
2. A state may be transitioned to when the user pressed a button on the touch screen. The GUI will have been configured such that if this happens, a function is called that changes the state variable and allows this switch to occur. The way that this works will likely depend on the GUI we end up using.
3. Every non-dynamic menu state has a back-button option associated with it that will set the state variable for the previous state.

State 0 – Initialize

1. This state always transitions to state 1 immediately after running the initialization code for the Graphical User Interface and clearing Mastermind’s variables.

State 1 – Main Menu

1. Displays the options “Export Data”, “Plot Data”, “Reboot”, and “Select Mode”.

2. If “Reboot” is pressed the Reboot function is ran and the Raspberry Pi will restart.
3. If “Export Data” is pressed, the state variable is set for state 4.
4. If “Plot Data” is pressed, the state variable is set for state 5.

State 2 – Select Data File to Export

1. User is prompted to select the file whose data they want to load onto an SD card.
2. When file selected, sets the state variable for state 3.

State 3 – Exporting

1. Waiting screen while export occurs.
2. Can be cancelled by pressing back button.

State 4 – Select Data to Plot

1. User is prompted to select the file whose data they want to plot to the touch screen.
2. When file selected, sets the state variable for state 5.

State 5 – Display Plotted Data

1. Functions are run to plot the data to the screen.

State 6 – Select Operational Mode

1. Displays the options “Manual Mode”, “Simulink Mode” and “Custom Mode”.
2. “Manual Mode” sets the state variable for state 7.
3. “Simulink Mode” sets the state variable for state 8.
4. “Custom Mode” sets the state variable for state 9.

State 7 – Run Manual Mode Prompt

1. Asks the user if they want to run manual mode, “Run?” is displayed
2. If selected, sets the state variable for state 12.

State 8 – Simulink File Selection

1. User is prompted to select the Simulink file they would like to execute
2. When file selected, sets the state variable for state 10.

State 9 – Custom File Selection

1. User is prompted to select the Custom file they would like to execute. A custom file is one that they have written themselves and uploaded to the Raspberry Pi. Could be any executable code written in their preferred language.
2. When file selected, sets the state variable for state 11.

State 10 – Throttle to Run Prompt

1. Prompts the user to hold down the throttle on the radio before executing the Simulink file.

2. When throttle squeezed, sets the state variable for state 12.

State 11 – Run Custom Mode Prompt

1. Prompts the user to hold down the throttle on the radio before executing the Simulink file.
2. When throttle squeezed, sets the state variable for state 12.

State 12 – Initialize Peripherals

1. Runs functions to initialize the sensors and actuator associated with each different operational mode.
2. When the correct peripherals are on, sets the state variable for state 13, 14 or 15 depending on the mode.
3. If there was an error with one of the peripherals booting up, sets the state variable for state 16.

States 13, 14, and 15 – Dynamic Modes (Manual, Simulink, and Custom)

1. These states will turn on any tasks associated with their respective dynamic operating mode.
2. Run any startup functions necessary for the mode.
3. React to dynamic errors by halting the vehicle and setting the state variable for state 16.
4. When “Cancel” Pressed on any of these, set the state variable for state 6.
5. When transitioning to static operating conditions, this state will run reset procedure, setting flags to let the other tasks know it is time to stop (not certain yet what these procedures might be).

State 16 – Display Operational Error

1. Displays the error message associated with an error variable (there will be several possible sources of error and the error variable tells which message to print)
2. Prompts the user to acknowledge that an error has occurred
3. Once error acknowledged button is pressed, sets the state variable for state 6.

6.1.3 Functions

Data Plotter

This function has not yet been designed but will be once we have gained a better understanding of how the Raspberry Pi and its operating system will interact with Simulink files and the data they contain. It is anticipated that data plotter will extract and process data from the file its given and plot it directly to the Raspberry Pi touch screen.

Export Data through USB (Universal Serial Bus)

This function will transfer data from the selected file through a USB cable so that the students can open it in MATLAB on their computers. There will be Unix commands and file navigation associated with this function.

Initialize Peripherals

Initialize Peripherals is a function that accepts a desired peripheral, such as the IMU, as an input and runs its initialization code. This will be used by the Initialize Peripherals state in mastermind. This function will return a peripheral error if the peripheral is not starting up properly. This code could be a part of that state but making it into a function will help prevent the firmware from getting too cluttered and unreadable.

Reboot

This function will reboot the Raspberry Pi using Unix commands, the same as the restart button on a computer. It will be used by Mastermind in the Main Menu state.

6.1.4 Firmware Manufacturing Updated

Because of the new firmware design outlined in section 5.5.3, the firmware manufacturing that finally took place is very different than described above in sections 6.1.1 through 6.1.3. The Serial Peripheral Interface code on the Teensy became the focus of the firmware build. An SPI task that handles all the commands, data acquisition, and actuation available to the Raspberry Pi was written in the Arduino IDE's loop function. Additionally, two interrupt service routines (ISR's), *spi0_isr* and *spi_transfer_complete_isr*, that facilitate SPI message transfers between the Raspberry Pi (Master) and the Teensy (Slave) were written to provide a robust and organized means of communicating data. This section will go over these implemented firmware changes.

SPI Task

The SPI task was organized in a loop format much like the original code. A big difference, however, is that nothing is executed without first receiving a command from Master, and none of its code requires knowledge of the state of the Pi. So rather than the Teensy having both a "nominal mode" and a "Simulink mode", it simply loops through and manages the register map so that Master can get to everything it needs access to.

As described in section 5.5.3, there are three types of registers the task will interact with: commands, sensor data, and actuations. The loop will begin by executing certain functions if the state of the command registers prompts them to. It will execute each function only once and clear the command register so that the command can be executed again. Then the task will gather each sensor value and place it in the appropriate register. Currently each sensor value is being updated as fast as the loop executes but later, this code can be set to execute only as often as allowed by each individual sensor. Finally, the task will update the setpoints of the motor controllers and the servo using the actuation registers. If the dead switch is active, actuations will only go through if the radio trigger is being pressed. After this, the code loops back up to the command code and the process repeats. Figure 82 illustrates this process.

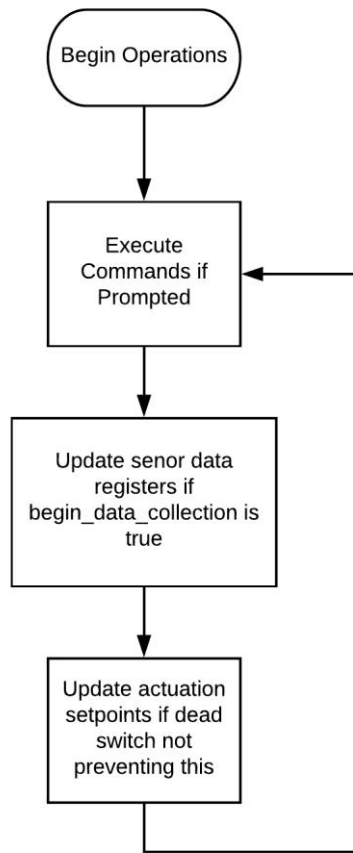


Figure 82. SPI task flowchart.

This implementation allows the Teensy to be isolated from any high level operational logic. It doesn't have to know whether a Simulink model is running on the Raspberry Pi, it doesn't need to know the frequency of the control loop, and it doesn't need to know what peripherals are turned on or not – it just does what it is asked to do. This allows all the operational procedure code to take place on the Raspberry Pi, it can be centralized there. This entire task is to interact with the register map data that the SPI communication interrupt service routines access during reads and writes from Master.

SPI Communication – SPI Message and Register Map Structure

An SPI message in this code consists of several one-byte frames. The Master can send as many frames per message as it wants and there are two types of messages it can initiate with the Slave, a read message and a write message. These are mutually exclusive, and which one is occurring is determined by the value of the MSB of the first byte received in a message, the Read/Write (R/W) bit. The master sets this bit to 1 for reads and 0 for writes. This byte also contains in the remaining 7 bits the register address (a value between 0 and 127). This value indexes a byte in the register map that is to be read from or written to with SPI data. The actual transfer of data between master and slave is done in the SPI hardware modules of the Teensy called shift registers. The shift registers both transfer frames bidirectionally, meaning that for Master to read

a byte, it must send a byte. The details of the SPI read and write processes are explained further in Figure 83 and Figure 84.

Descriptions of the currently available registers and their associated addresses and data types can be found in Appendix N. The data in the registers may take up 1 or more register addresses depending on the length of the data type. We are currently only using 40 bytes of the available 128. Certain data may span several register addresses depending on the length of the data. Each 16-bit data type takes up 2 registers. The synchronization between data and registers is managed by the register map C struct. The struct defines the name and data type of each element to be in the register map. A C union type binds the register map struct to an array of 128 bytes. This way the exact same data can be accessed like this:

```
registers.bytes[1]
```

or like this:

```
registers.reg_map.begin_data_collection
```

This mapping of data to registers was important in allowing the SPI communication protocol scheme to work in the proper manner while simultaneously allowing the Teensy to treat the registers as grouped elements of differing data types. Altering the register map can be accomplished by adding a data type and name to the register struct and printing the register map to the serial port.

Below is an example of an SPI address frame, the first byte of an SPI message:

```
0b10000001 - This as the first frame of a message indicates that a read message is to follow and it should start at index 1 of the registers, i.e. registers.bytes[1].
```

This index currently points to the memory location of the "begin data collection" command flag, i.e. registers.reg_map.begin_data_collection. The next frame the master sends will fill this memory location with data. Any subsequent frames will fill further registers with data, i.e. registers.bytes[2], registers.bytes[3] and so on. The following bytes of a message are the data to be transferred.

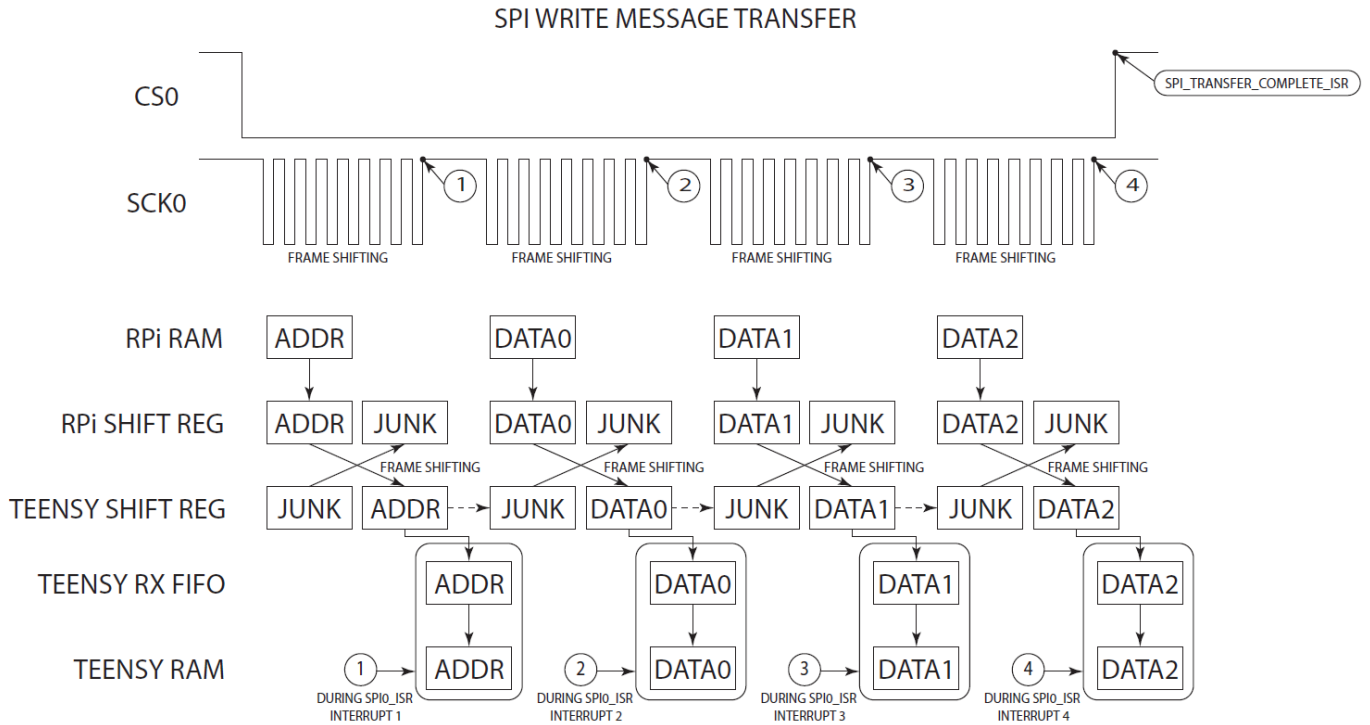


Figure 83. SPI write message diagram. Boxes refer to SPI data and they line up horizontally with a description of their memory location. CS0 refers to the chip select line that Master brings low to initiate a message. SCK0 refers to the clock line that controls the rate that the bits shift between Master and Slave. “SHIFT REG” refers to the shift registers built into the SPI modules whose data is exchanged simultaneously between Master and Slave during a shifting operation. The “RX FIFO” or “Receive First In First Out” is the built-in buffer that shift register data gets copied into immediately after a frame shift. This register is the only way to access incoming frames. “Junk” refers to garbage data that is ignored by the device that receives it.

Figure 83 demonstrates a three-byte SPI write message transfer. The Raspberry Pi sends the address byte followed by three bytes of data to be written to the Teensy register map starting at that address. Each frame shift between shift registers occurs during eight toggles of the clock line and is followed by a decoding of the transferred information. After each frame shift, *spi0_isr* gets executed. It grabs the Master to Slave data from the RX FIFO register and stores it in the appropriate spot in the register map (Teensy RAM). The exact process *spi0_isr* undergoes to decode an incoming frame is found in the *spi0_isr* flowchart of Figure 85.

Figure 84 demonstrates a two-byte SPI read message transfer. For read messages on the Teensy, incoming data other than the address byte is junk. The address is used to retrieve register data to be transmitted. The outgoing data is placed in the TX FIFO buffer, and then into the shift register, before being shifted out to the Raspberry Pi. There is a lag between when a byte is placed into the TX FIFO and when that byte is moved into the shift register to be sent out. The result is that Master must send three bytes instead of two to receive a single byte useful read data. More information about the TX and RX FIFO registers and other SPI module specifics can be found in the MK66FX1M0 chip reference manual, chapter 57 [Freescale Semiconductor, Inc.].

For both read and write, the end result of an SPI message is the data stored in either Teensy or Raspberry Pi RAM when the CS0 line is brought high and the transfer terminated.

SPI Communication – The interrupt Service Routines

The interrupt service routines, *spi0_isr* and *spi_transfer_complete_isr* that perform the decoding of the SPI frames will be explained in this section.

This interrupt service routine in Figure 85 decodes each frame of an SPI message upon the conclusion of a frame transfer. The flowchart begins at this occurrence. If *first_interrupt_flag* is true, the code will clear *first_interrupt_flag* to prevent code specific to the address frame from running twice. It will then copy the entire Teensy register map into a buffer (to prevent corruption of vital data). This buffer is what the ISR will use to access the registers for the duration of an SPI message. It will proceed to extract the register address and R/W bit and place them in buffers for use in subsequent interrupts. If the message is a read message, it will place the first byte requested from the register map buffer into TX FIFO through a write to the SPIO_PUSHR_SLAVE register. If the message is a write, nothing is done with the register map buffer. Subsequent interrupts will not do the preparatory steps, rather they only place write data into or extract read data from the register map buffer. After every interrupt, *except* after that of the first frame of a write message, the address buffer is incremented so that it points to the next byte in the register map. Also, the Receive FIFO Drain Flag is cleared to allow for a new interrupt to be triggered by the SPI module.

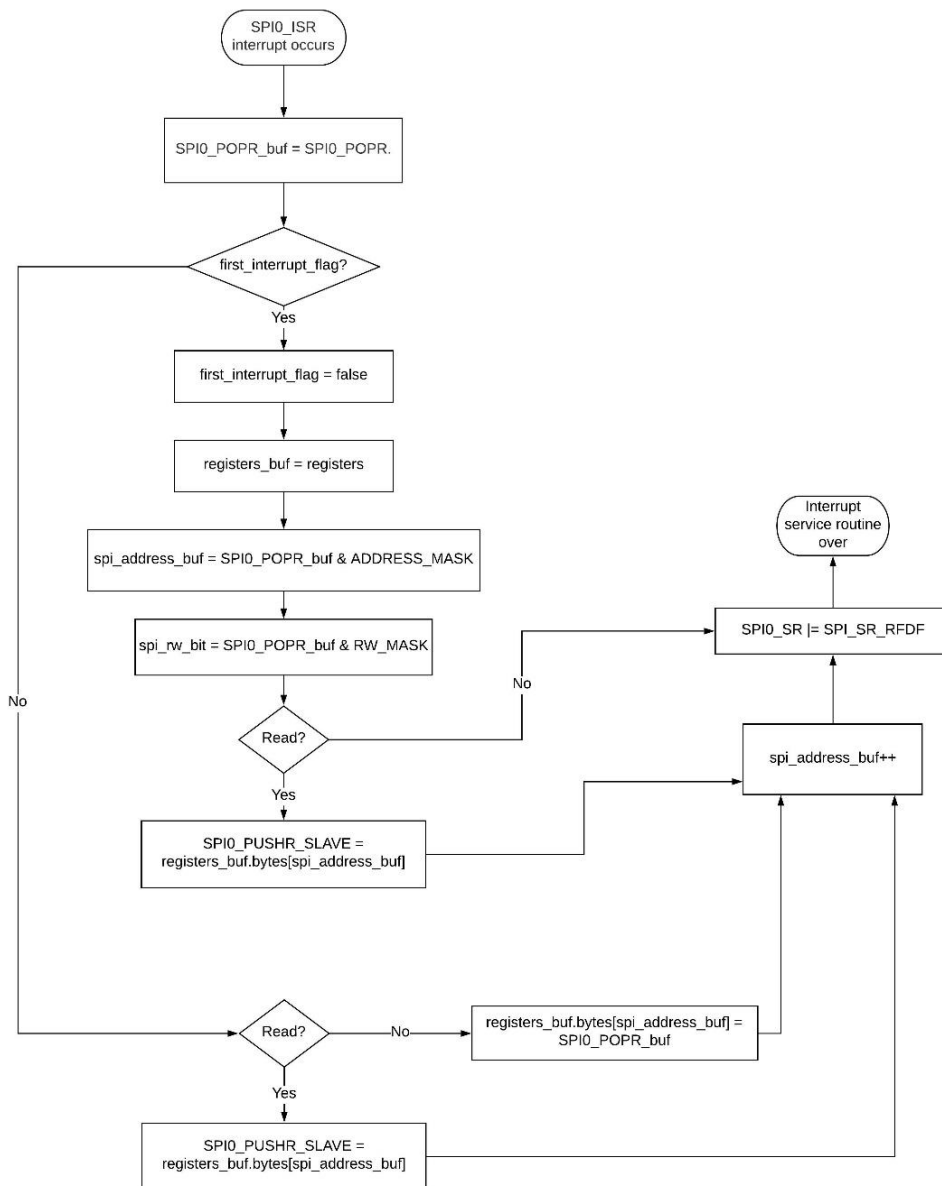


Figure 85. *spi0_isr* interrupt service routine flowchart. This is the flowchart was implemented in the Teensy firmware to decode incoming SPI frames on the Teensy. “SPI_POPR” is the register used to access the RX FIFO register. “SPI0_PUSHR_SLAVE” is the register used to access the TX FIFO register. “Registers” is the register map union – the 128 bytes of Teensy RAM. “ADDRESS_MASK” and “RW_MASK” are bit masks used to extract the address and R/W bit from the first frame of a message. “SPI_SR_RFDF” is the bit mask for the Receive FIFO Drain Flag in the SPI status register of the SPI module.

The conclusion of an SPI message prompts the chip select line (CS0 pin) to be brought high and the *spi_transfer_complete_isr* interrupt to occur. This interrupt is much simpler – it resets *first_interrupt_flag*, and places *register_buf* back into registers. This interrupt allows for the initial preparatory code in *spi0_isr* know to run upon receipt of the first frame of a message.

6.2 Simulink Communication Subsystems Manufacturing

The Simulink SPI communication subsystems used to test control our control algorithms were designed using SPI communication blocks from the Raspberry Pi Simulink support package. They work in conjunction with the Teensy SPI code described above in section 6.1.4.

Figure 86 shows the communication box that serves as the plant for an implemented control system. Currently the only sensor implemented is the IMU, each Euler angle, acceleration, and gyro reading come from this sensor. The *radio_throttle* and *radio_steering* variables come from the receiver and provide the radio input to the system. The throttle signals are profile velocity mode setpoints to each of the Maxon motor controllers. The *servo_out* signal is a steering command. Within this subsystem is an SPI communication block for each signal in the system currently. The Raspberry Pi will send a read or write message for each signal in this subsystem every timestep of the control loop. Data types in this subsystem are of vital importance as these blocks are interacting with 8-bit registers on the Teensy while simulation data is by default of the double data type. Data type conversions, byte concatenations, and bit masks are found throughout this subsystem.

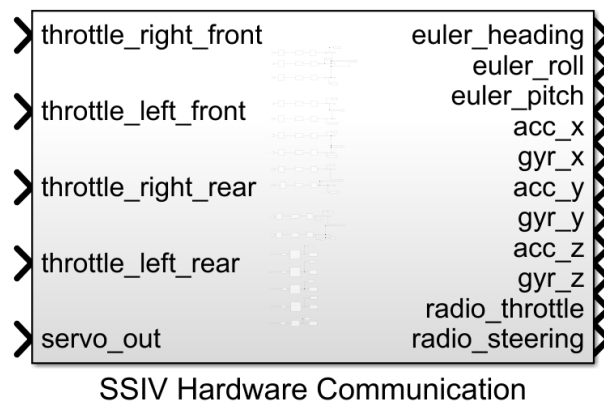


Figure 86. The communication “black box” Simulink subsystem with radio, IMU, steering, and throttle access.

Figure 87 shows the two read message SPI Master Transfer blocks associated with the *radio_throttle* and *radio_steering* signals. The Teensy register addresses of the data, 26 and 28 are the constant blocks on the far left. The address byte becomes the first element of a vector of unsigned 8-bit integers that are inputs to the SPI Master Transfer block as the outgoing message. This outgoing message is under the subsystem titled “read message (Master → Slave)”. This message is exposed in Figure 88. The returned message is processed in the subsystem titled “return message (Slave → Master)”. This one is exposed in Figure 89.

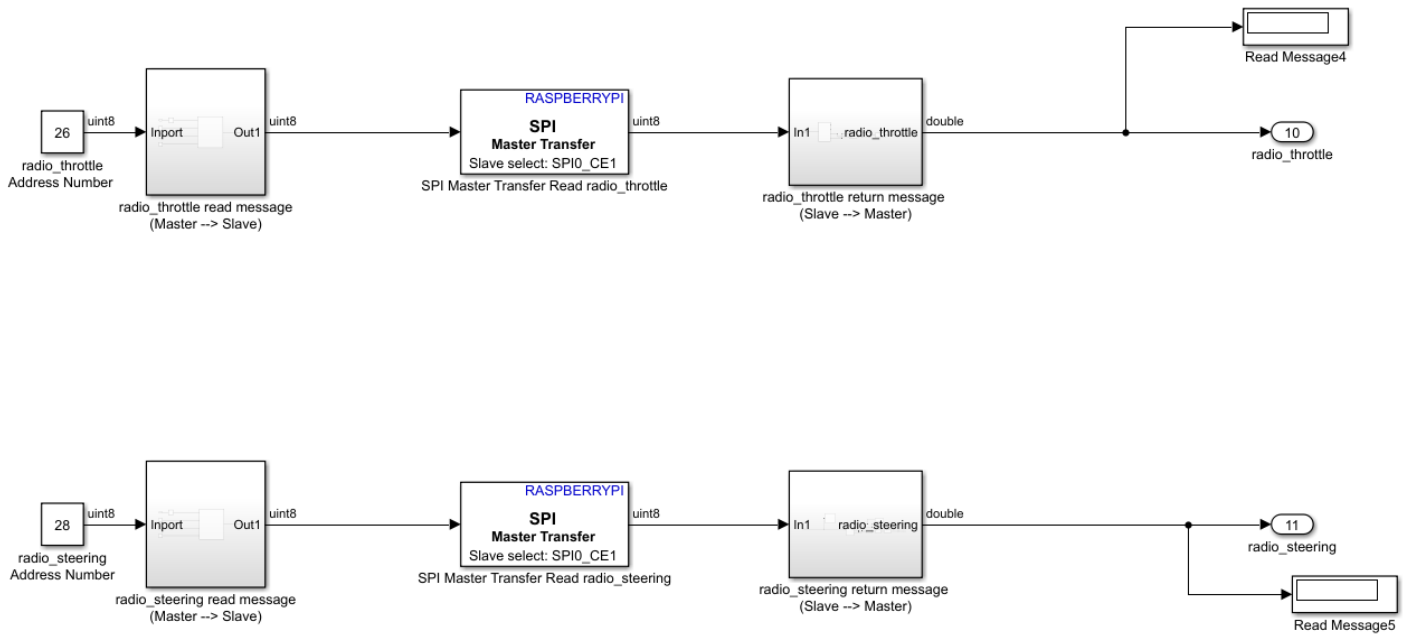


Figure 87. Radio SPI Master Transfer blocks configured to send read messages to the Teensy. These along with every other read and write message the Raspberry Pi will send are inside the Communication box of Figure 86.

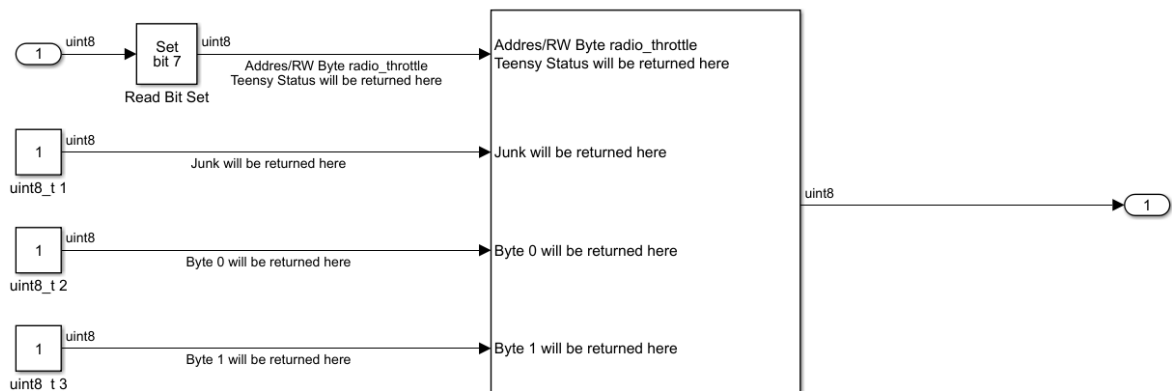


Figure 88. Read message (Master → Slave) for the *radio_throttle* signal. All read message subsystems have identical structure but may be reading more than two bytes of data.

The subsystem in Figure 88 constructs the four-byte message needed to request two bytes of read data from the Teensy register map. The first byte is the address and bit 7 of the address byte is being set to signal to *spi0_isr* on the Teensy that the incoming SPI message is a read message. This message is sending junk data that will be exchanged for the *radio_throttle* data.

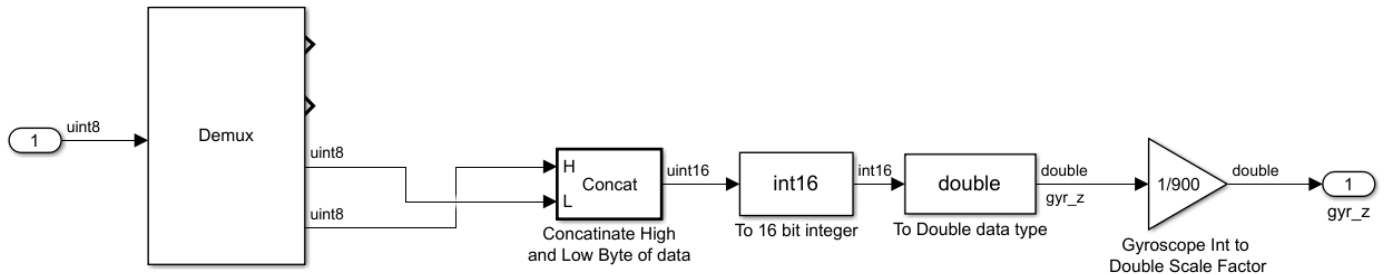


Figure 89. Return message (Slave → Master) for the *gyr_z* signal. All return message subsystems have identical structure but may be reading more than two bytes of data.

The subsystem in Figure 89 takes the data bytes of the return message, concatenates them, casts the result into a 16-bit integer (to recover the signage of the signal) and then casts that into a double data type for compatibility with the simulation data. Finally, a scale factor is applied to make the units correct – the 16-bit value was pre-scaled on the BNO055 to provide the proper resolution. When this data is brought into Simulink, dividing by 900 produces a signal in rotations per second that has floating point precision. The input to this block is the raw read bytes from the SPI Master Transfer block and the output is the *gyr_z* signal, a properly scaled floating-point value.

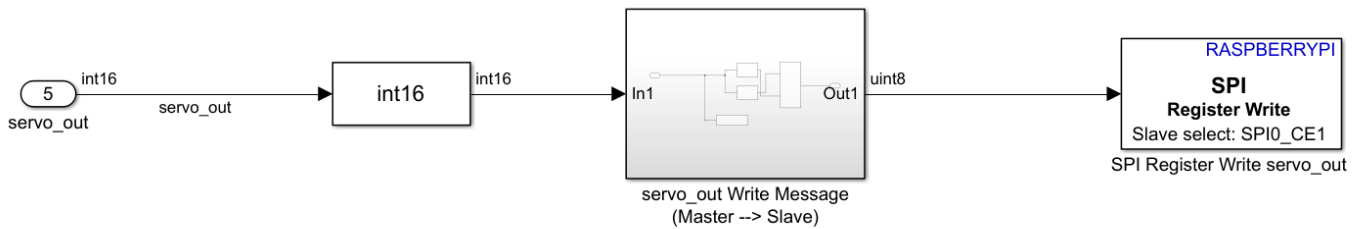


Figure 90. SPI Register Write block configured to write the *servo_out* actuation to the Teensy register map. These along with every other read and write message the Raspberry Pi will send are inside the Communication box of Figure 87.

Figure 90 shows an SPI Register Write block set up to send a write message to the *servo_out* register on the Teensy. This block is a bit different than the SPI Master Transfer block as the address is in the block’s internal settings (Figure 91). The *servo_out* Simulink data starts out as a double for use in the model calculations but is converted here to a 16-bit integer before it is converted into a write message that can be sent over SPI. This write message is constructed in the subsystem titled “Write message (Master → Slave)” and is exposed in Figure 88.

The write message of Figure 92 slices the 16-bit integer actuation value into a high byte and low byte so that it is compatible with the SPI communication protocol. The low byte precedes the high byte in the message due to the endianness of the Teensy.

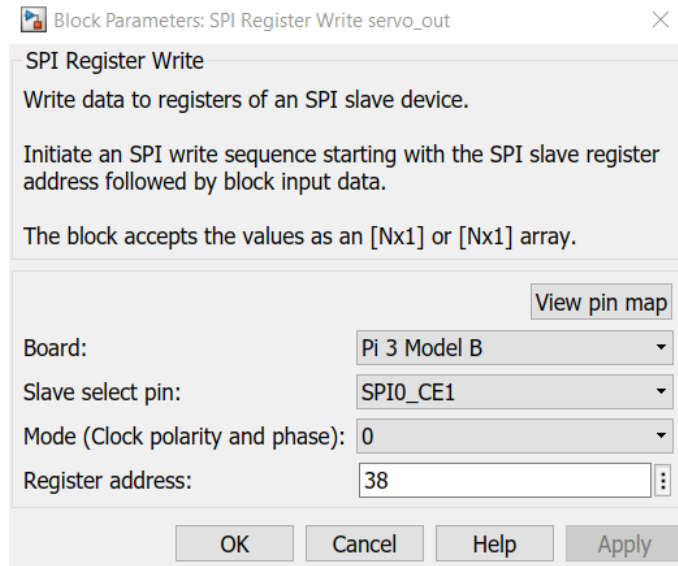


Figure 91. SPI Register Write block internal settings. Here, the address for the *servo_out* register (38) is set.

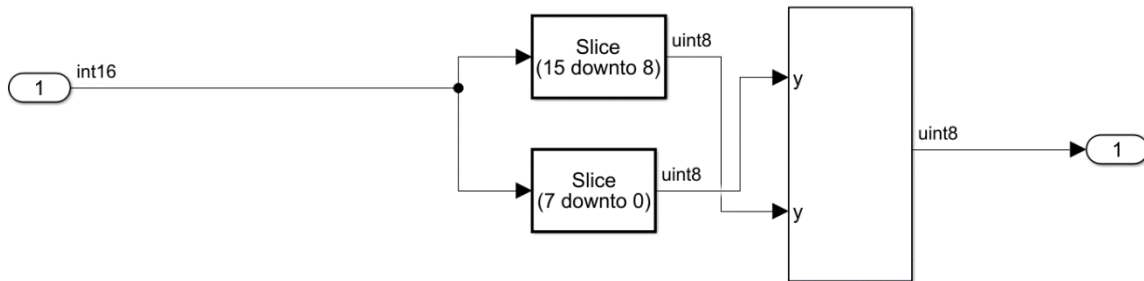


Figure 92. Write message (Master → Slave) subsystem. All write message subsystems have similar structure but may be writing a different number of bytes

6.3 Sensor Manufacturing

6.3.1 Vehicle to Vehicle Communication

Initially the plan was to implement a vehicle to vehicle communication architecture by using Bluetooth sensors. Bluetooth is a wireless communication protocol that has a master device and one or more slave devices. It was decided, however that this protocol would not facilitate vehicle to vehicle communication properly. It does not make sense to have one vehicle be a master and one vehicle be a slave.

In place of the Bluetooth sensors, Matlab’s built in IP communication box was integrated into the Simulink model in order to provide the opportunity for students to implement vehicle to vehicle communication in their projects. This is shown in Figure 93 below.

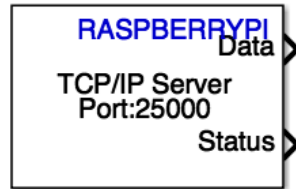


Figure 93. This RPi TCP/IP blocks allow users to send and receive data from one raspberry pi to another.

The current internet configuration on the car requires connection to SecureMustangWireless. A byproduct of this is that the IP address of the raspberry pi on the car is not set. This makes it difficult to incorporate the block into the “Communication Black Box”. The RPi IP block requires an IP address of the target raspberry pi to be entered. This can be difficult for the students to do if the IP addresses are not static. This issue is shown in Figure 94 below.

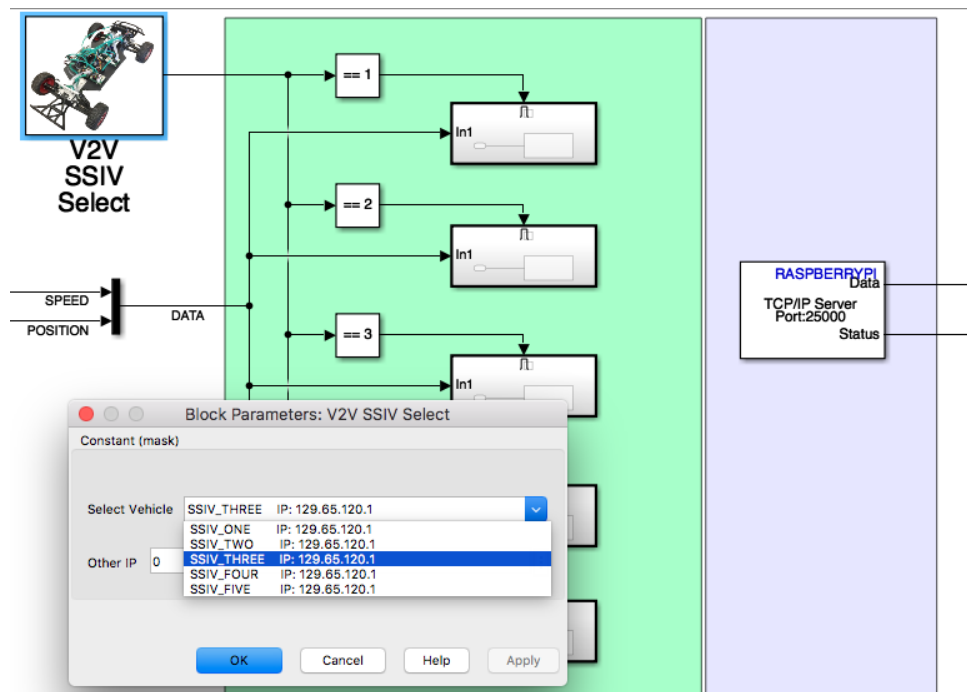


Figure 94: As shown in this image, ideally, if all of the IP addresses of all of the cars are static, it can be easy for students to select which vehicle they want to communicate with since the blocks can be pre-configured with the correct IP addresses.

6.3.2 LiDAR

One of the main objectives for the senior project was to integrate a LiDAR sensor and a GPS sensors so that students would have the opportunity to get experience with these sensors, both of which are central sensors used by intelligent vehicles. The LiDAR was a more difficult sensor to integrate because of the immense amount of data it produces. The LiDAR has a field of view of 240 degrees with a resolution of .35 degrees which equates to 682 data points. Each data point is received as a 16-bit value, thus a message of all the data is roughly 1364 bytes. It was decided that the LiDAR would be connected to the Raspberry Pi as it would be a lot of data to store on the Teensy and send via SPI.

In order to accommodate the LiDAR sensor into the Raspberry Pi, a python program was written to handle the data transfer to the sensor and to detect and report on any errors. Additionally, a device driver s-function was created in Simulink to retrieve LiDAR data that was stored in a file. The LiDAR architecture is summarized in Figure 95 below.

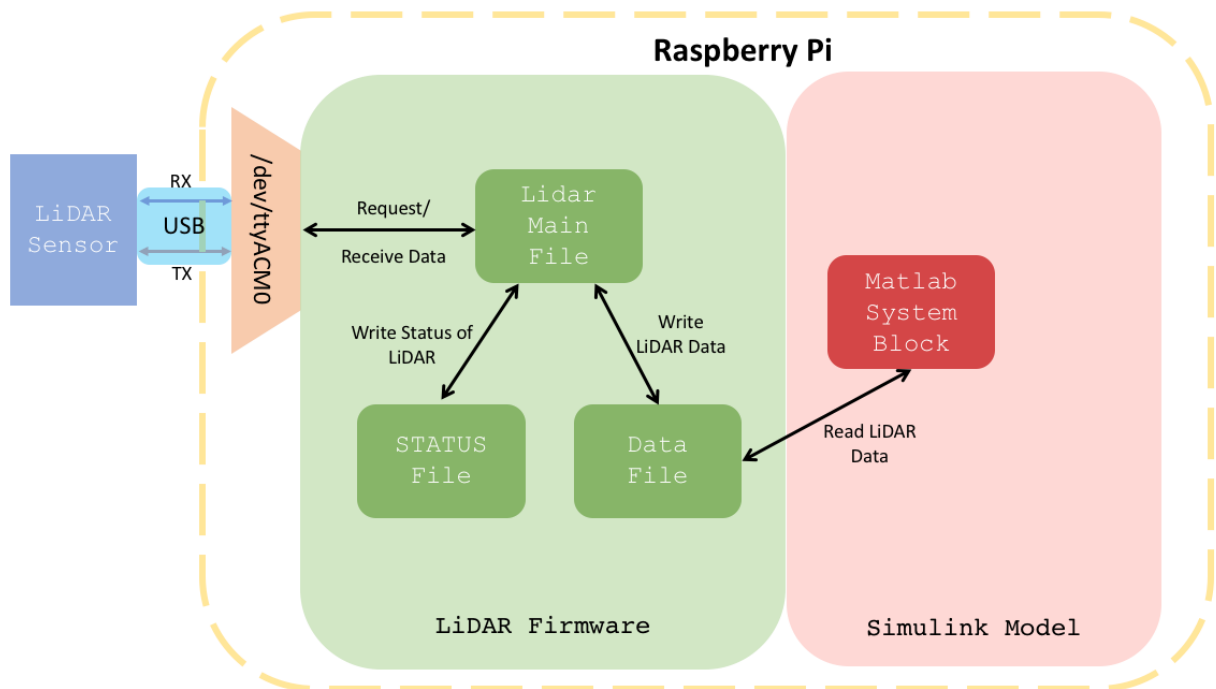


Figure 95. The graphic shows the architecture of the LiDAR data transfer, including the firmware program and the Simulink program.

The Raspberry Pi connects to the LiDAR via USB virtual serial. The LiDAR serial device appears on port /dev/ttyACM0 or /dev/ttyACM1. The numerical index is assigned based on the order in which the device connected. This is similar to how a first flash drive connected will be connected to drive E and then a subsequent flash drive will be connected to drive F. Given that this can be an issue (the port address changing) if for example another USB serial device is connected, the LiDAR program tries to connect with /dev/ttyACM0 (the default port) and then

attempts to connect to /dev/ttyACM1. Upon connecting successfully to one of the ports it sends a test message to see if indeed the LiDAR is on the other side (and not some other sensor device). If the correct response is received, then the program can confirm that the LiDAR is connected and it can proceed to requesting data. In the case that either the LiDAR is not connected to any of the /dev/ttyACM* ports, then an error message is saved into the “lidar_status” file and -1 values are output into the “lidar_data” file in place of actual data. The “lidar_status” file can be read by the LCD screen to report any of the messages stored there, for example, “LiDAR device not detected.”

There is a block in Simulink called the “Matlab System Block” where a device driver s-function can be defined and implemented. A device driver is code written in order to give the ability for Simulink to interact with the hardware on which it’s running (in the case of this project- the hardware is the Raspberry Pi). For example, we used an SPI Raspberry Pi block provided by Simulink to use the SPI on the Raspberry Pi. It is possible to create one’s “own” SPI block by using a Matlab System device driver to implement the same functionality. Other applications include setting an LED high or low and writing to a text file. A device driver s-function was used in order to be able to read and write from text files stored on the Raspberry Pi. The device driver blocks are shown in figure 96. File reading and writing was chosen as the method to transfer data from the python firmware running on the Raspberry Pi to the Simulink Model running on the Raspberry Pi. Other options considered were using pipes to send data between the executable files and transferring data over IP through loopback.

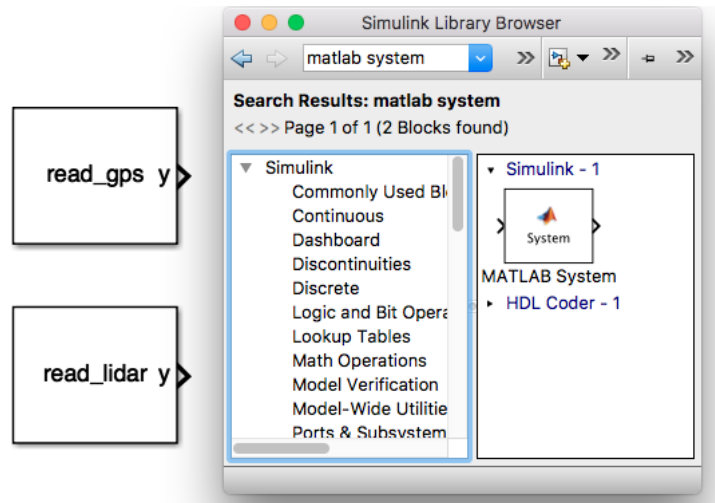


Figure 96. The above image shows the two Matlab System blocks created to retrieve data from the LiDAR and the GPS.

The Matlab system was created by writing a standard C function to read and write to a file. This C function is then wrapped by a Matlab function by specifying input and output parameters (datatype, size, etc.). It can then be included as a block in Simulink. The only requirement is that the configuration files be included in the path of the Simulink model. This file structure is shown in Figure 97 below.

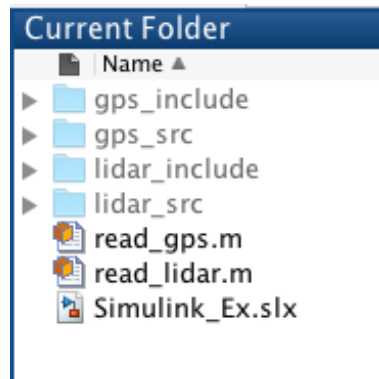


Figure 97. The image shows the dependencies (include/src folders and .m files) required to use the LiDAR and GPS matlab system blocks included in the same path as the Simulink model being run.

6.3.3 GPS

The GPS sensors were integrated into the system architecture in much the same way the LiDAR sensor was, that is, through serial communication to the Raspberry Pi and then subsequent data transfer to Simulink through a Matlab System block. One key difference is that the serial communication is achieved through GPIO serial as oppose usb virtual serial. As a result the communication occurs through the /dev/ttyAMA* port. The GPS system architecture is shown in Figure 98 below.

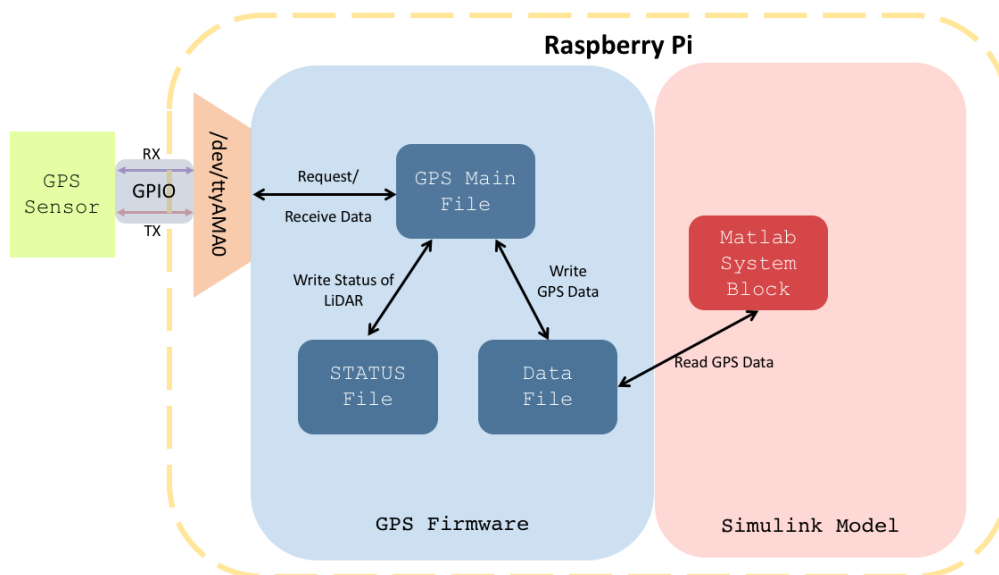


Figure 98. The graphic shows the architecture of the GPS data transfer, including the firmware program and the Simulink program.

Another difference from the LiDAR firmware architecture is that the GPS “Main” file calls another file provided by the GPS manufacturer- Marvel Mind Robotics. This file helps to parse the data and initialize the sensor.

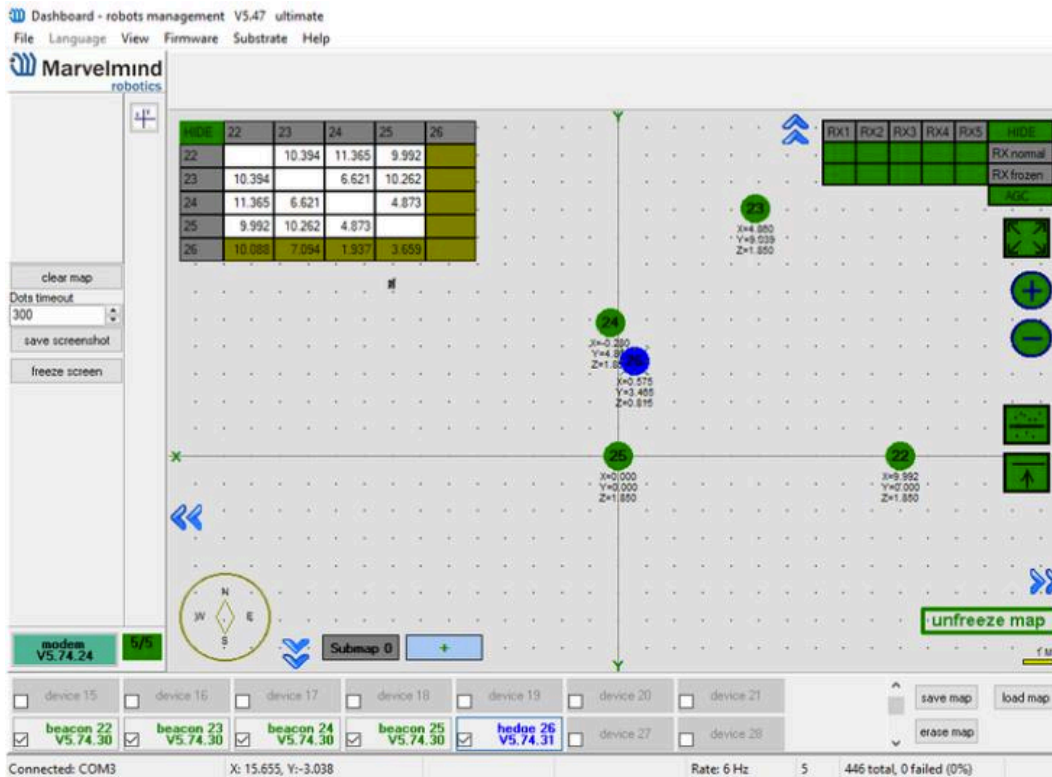


Figure 99. The image shows the GPS beacon locations through the use of “Dashboard” a program provided by Marvel Mind Robotics to monitor the beacons on the computer.

Before the GPS beacons could be used they needed to be flashed with the latest firmware. This was done by connecting them to the computer and opening up the software provided by the manufacturer. The software can be seen in figure 99. Once flashed the sensors were configured. This involved setting the communication protocol, the baud rate, the device address, and selecting the beacon to be used as the hedgehog. The hedgehog is the beacon whose location is being monitored.

Once the beacons were configured, the stationary beacons were placed at different corners of the room and the hedgehog was connected to the Raspberry Pi through serial. It is also necessary to connect the gps modem to power (usb outlet or computer). The modem has no battery and so needs to be powered to be activated. The GPS sensors will not “wake up” unless the gps modem is powered on.

While the LiDAR outputs up to 682 data values to Simulink, currently, the GPS sensor only outputs 3- x, y, and z position. The sensor can also output IMU readings from a built in IMU, but this is not necessary given that we already have an IMU.

6.4 Controller Manufacturing

6.4.1 Lane Keeping Controller

A lane keeping controller was implemented and tested on the vehicle. Before the lane keeping controller was implemented it was simulated using a model of the vehicle. The simulation was drawn from the work done by Taylor et al in the research paper, “A Comparative Study of Vision-Based Lateral Control Strategies for Autonomous Highway Driving.” [Taylor]

The article describes a full state feedback controller design. The controller goal was to regulate the offset distance, y_l , at the look ahead distance to 0, and thereby maintain the vehicle at the center of the lane line. The article suggested pole locations along with system parameters such as the look ahead distance. The general structure of the lane keeping controller is shown in Figure 100 below.

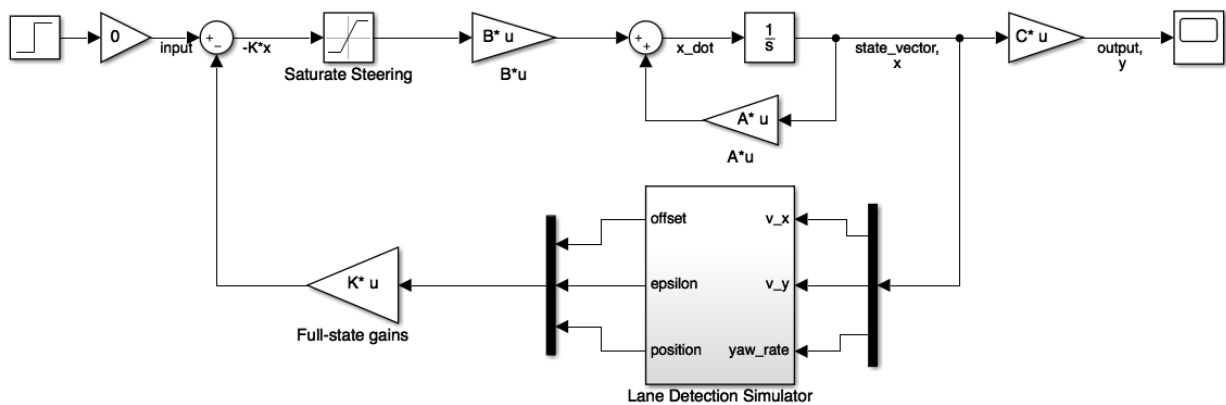


Figure 100. The model above was used for a simulation to demonstrate the performance of the lane keeping controller.

The simulation was made by generating lane lines and inputting the location of the lane lines into the controller to replicate the process of computer vision detecting lane lines. One of the simulations was performed with the vehicle starting initially at one of the lane lines. The goal of this simulation was to see whether the car would reach a steady state position along the center line. The results are shown below in Figure 101.

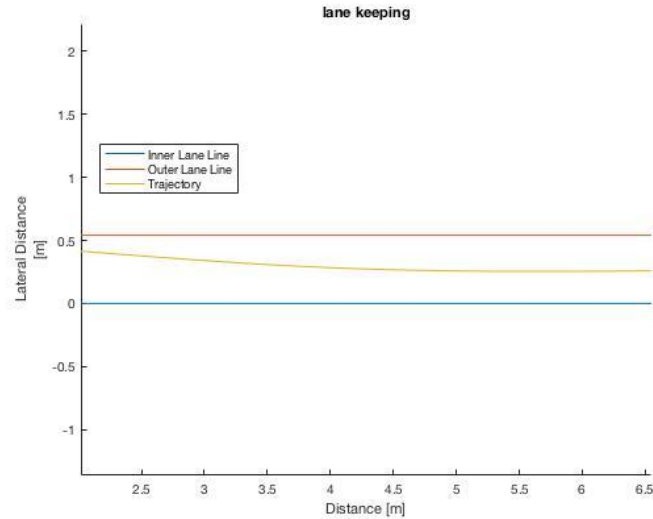


Figure 101. The plot of the trajectory of the car shows it approaching a steady state location in the centerline of the lane.

An animation was also generated to see the car's motion throughout time. Some snippets from the animation are shown below in Figure 102.

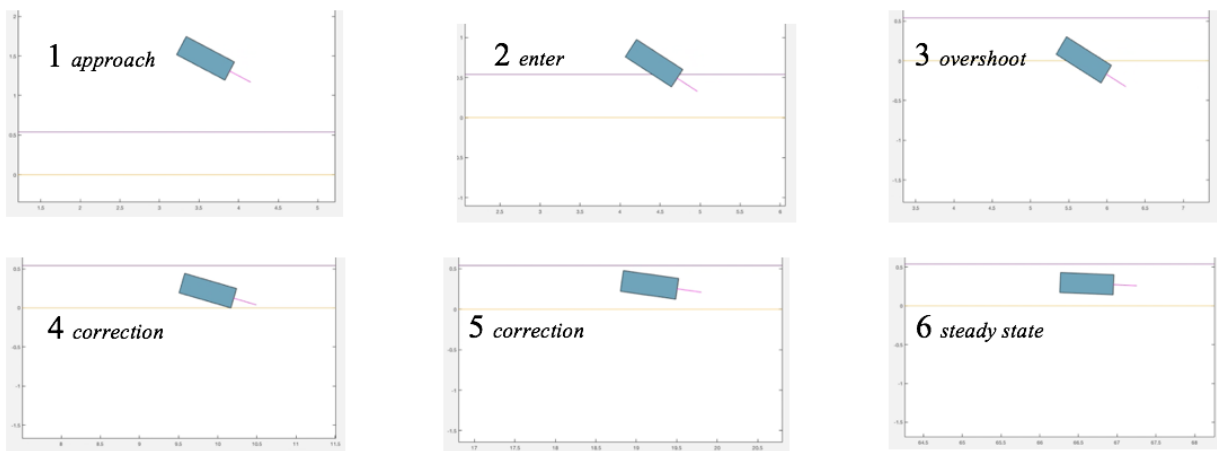


Figure 102. Simulation of approaching centerline from a starting offset distance. (Faint colored lines are lane lines)

The simulation was used to determine gain values that produced favorable results. After the simulation of the controller was performed, a simulation of the computer vision was done, in order to improve it before a final version was used for live lane keeping.

The computer vision algorithm to detect the lane lines makes use of a sobel filter, a white color filter, a hough transform, and an outlier rejecter. The sobel filter detects lines on an image. It will detect the lane lines but also a lot of noise. The white color filter eliminates lines that are not white. The hough transform then takes the input of the filtered black and white image and

outputs the location of its prediction for the lane line. The sequence of operations is summarized in Figure 103 below.



Figure 103. The first photo shows the output of the sobel filter which outputs white for any edge detected in the image. The second photo shows the result of filtering out the noise by ignoring non-white pixels. The last image shows the predicted lane lines (black) overlaying the actual lane lines (white) resulting from the output of the hough transform.

Once good performance was achieved from testing out the computer vision algorithm on videos of lane lines, the next step was to test out the controller in conjunction with the computer vision algorithm deployed on the SSIV.

Figure 104 below shows an image of one of the lane keeping tests.

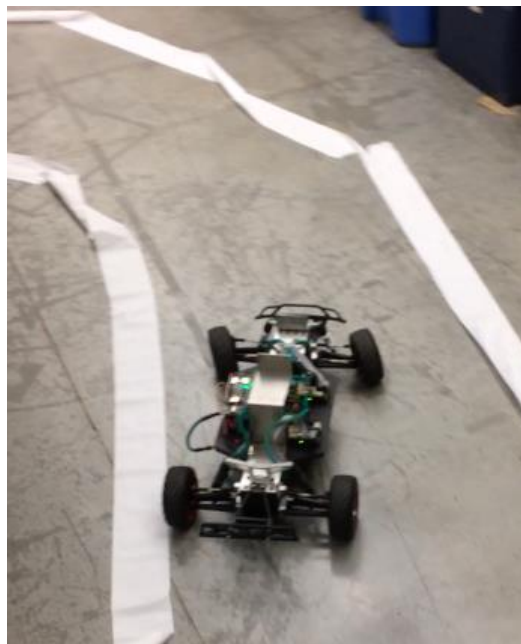


Figure 104. Lane keeping controller tested out on the SSIV. (Notice the wheels turned in preparation for the upcoming curve)

6.4.2 Stability Controller

When implementing a controller like the one described in section 5.4.3, one has to customize it for the system at hand. For us, that meant that we had to modify the controller for the sensor data we were receiving. One of the most difficult parts of this controller design was recreating the error states from the sensor feedback. The stability controller uses data from the IMU, which outputs yaw, pitch, and roll angles, yaw, pitch, and roll rates, as well as accelerations in x , y , and z . The error states of lateral velocity, lateral acceleration, yaw angle, and yaw rate all had to be recreated from the data output of the IMU. Additionally, we had to work out local-to-global axis conversions. The data output from the IMU are in local axes, that is, the axes unique to the current orientation of the vehicle. They had to be converted to the global frame, or the stationary frame that the vehicle is operating in. Figure 105 below better illustrates the distinction between the local and global frames.

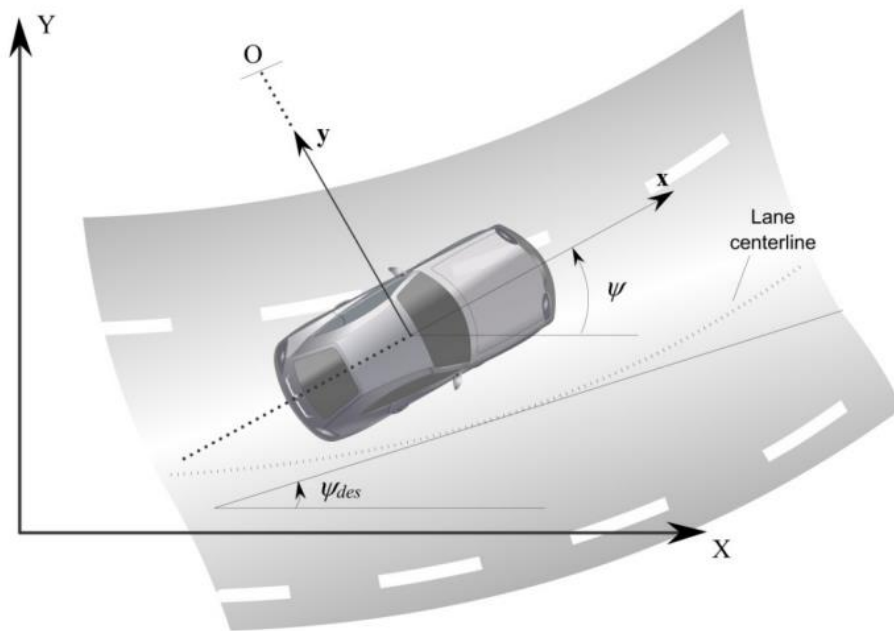


Figure 105. Bicycle model of a vehicle navigating a turn (Rajamani, 27).

The local coordinates x and y are vehicle-centric, whereas the global coordinates X and Y are stationary and relative to the space the vehicle is operating in. As seen in the figure, the actual yaw angle of the vehicle is defined as the angle between x and X . The yaw angle was used to perform a coordinate transform to find the error between the actual and desired yaw angle. With a constant desired yaw rate, to find the yaw rate error we simply subtracted the desired yaw rate from the actual yaw rate. For the yaw angle, the desired yaw angle was simply the integral of the desired yaw rate, so a line instead of a constant. Both the actual and desired yaw angles were converted to global coordinates, and then the angle between them was found using trigonometry, generating the yaw angle error.

Ideally, this error would then have been used to perform a coordinate transform on the positional and velocity data to generate the error states in the position and lateral velocity. However, we had no way of measuring the vehicle's position relative to a desired position, so we were unable to implement this error state. The lateral velocity error can be reconstructed using the relationship below:

$$\dot{e}_1 = \dot{y} + u * e_2 ,$$

where e_1 is the positional error, e_2 is the orientation error, or yaw angle error, \dot{y} is the lateral velocity, and u is the vehicle's longitudinal velocity, which was to be held constant. However, we also have no way of measuring the lateral velocity—while we could simply integrate the lateral acceleration data from the IMU, we thought this was bad practice. Thus, we were unable to include the positional error or the lateral velocity error states in our model. How this affected the controller's performance will be discussed in section 7.3.2.

The yaw controller used in testing is shown in Appendix K.

7 Design Verification

Since this was largely a software design project, it was necessary to do a lot of testing of the design along the way. The tests that were performed are broken up into sensor tests, firmware tests, controller tests, and usability tests.

7.1 Design Verification Plan

The tests that were planned are summarized in the section below. Later sections discuss the test procedures and results in more detail.

Table 20. Design Verification Plan

Project Division	Test	Description	Test Date
Sensor	LiDAR	Test retrieving sensor data on Teensy or Raspberry Pi	3/02
	Camera Lane Detection	Test algorithm on the car	3/16
	GPS Sensor	Confirm accuracy of measurements.	3/09
Firmware	Touch Screen basic functionality	Test out a basic GUI functions	2/26
	GUI Menu System	Navigate through complete Menu system and check for bugs	3/12
	Firmware on Raspberry Pi	Test if/how running the firmware on the Raspberry Pi affects the Simulink model	2/19
Controller	Lane Keeping Simulation	Simulate using vehicle model	3/16
	Yaw Control Simulation	Simulate using vehicle model	3/16
	Adaptive Cruise Control Simulation	Simulate using vehicle model	3/16
	Lane Keeping on Car	Implement on car and observe, compare to simulation	3/23
	Yaw Control on Car	Implement on car and observe, compare to simulation	5/29
	Adaptive Cruise Control on Car	Implement on car and observe, compare to simulation	3/23
	Vehicle Model	Compare to car's experimental response	3/09
Usability	Focus Group	Verify that vehicle is easy to use in the form of a focus group in which students will be given a simple task to implement on the car in a given period of time.	5/21

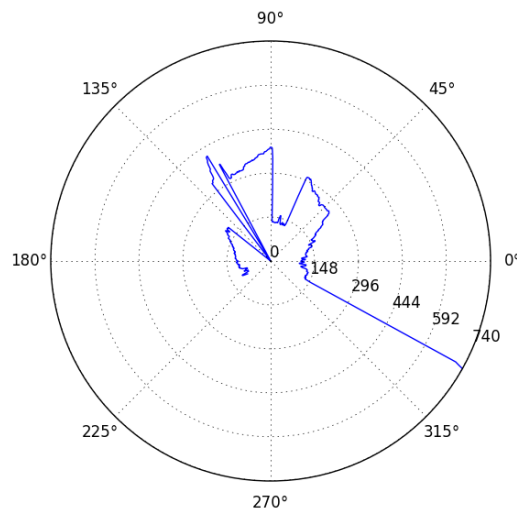
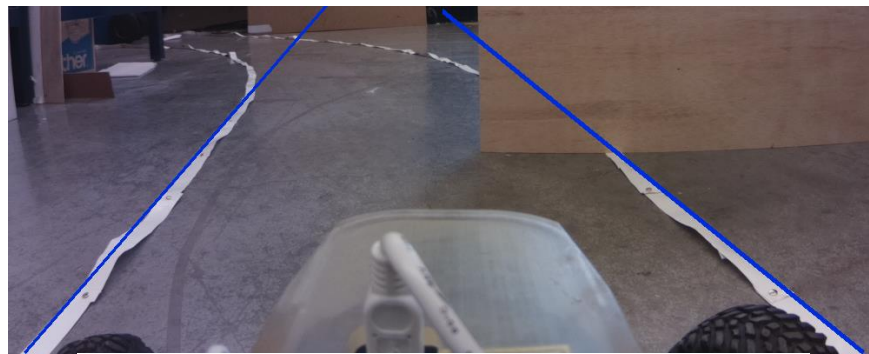
7.1 Sensor Testing

As discussed in section 5.3, once we are interfacing with all the sensors we want to validate the data that they are outputting and also test the data in some control algorithm to confirm that the communication of the sensor data to the Simulink model is functioning. Within the Simulink model we will have to update the communication black box to have the inputs and outputs required by the new sensors.

7.1.2 LiDAR and GPS

The sensor integration was tested to confirm that the correct values were being received by the sensors and that data communication was functioning properly. The LiDAR and GPS sensors

were tested by having the sensor firmware run in parallel with a Simulink model. As described in the manufacturing section, the sensors were interfaced with Simulink by creating a device driver Matlab system block. These blocks read from files that are constantly being updated with new data by the sensor firmware, hence why the sensor firmware and the Simulink file need to run in parallel. A test was done to test out each sensor outputting its corresponding sensor values to Simulink while a controller was being run. It was decided that the lane keeping controller would be running and that the LiDAR data would be used in real time to stop before hitting an obstacle. The GPS data would just be monitored to ensure that it was refreshing and updating to the expected location as there was not enough time to integrate it into the controller by, for example, using the GPS to follow a path and arrive at a destination point. The test performed as expected with the car driving in the lane and then the car stopping before it hit the obstacle. There was however, more noise than expected on the LiDAR sensor (see figure 106 below).



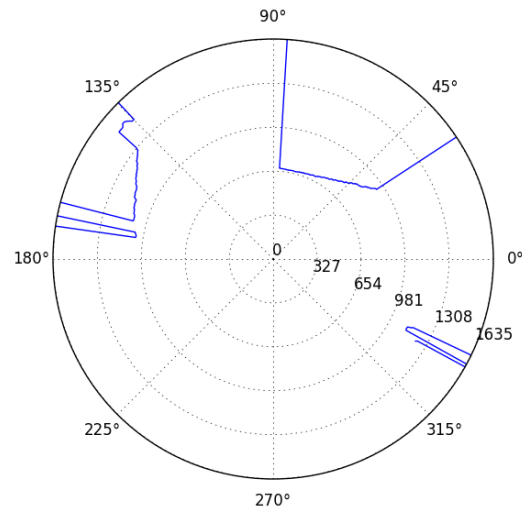


Figure 106: The top image shows the environment the car was placed in to try to get it to both follow the lane line and stop before hitting obstacles. The middle image shows unfiltered LiDAR data of one of the obstacles. Note that a few data points are stuck at zero, which is unexpected. The third image shows LiDAR data where noise is rejected by ignoring data points that are zero and replacing those values with averaged values near the zero point. Plot dimension is in mm.

The LiDAR was found to often times output values of zero at random angles. These data values do not represent an object detected. It is not known why the LiDAR outputs these values. When the LiDAR was connected with a computer software offered by the manufacturer to show a visual of the data, the zero values appeared as well, confirming that it is not a communication issue. In order to remove the zero values, an algorithm was used to convert the zero values to the average of the non-zero neighbors around it.

The GPS sensor data also had quite a bit of noise in it. The manufacturer claims $\pm 2\text{cm}$, but at some locations, the location fluctuated by as much as 8cm. The GPS sensors initially tracked the car accurately, but after a certain distance, the data started to become inaccurate. However, when the same setup was run with software on the computer provided by the manufacture outputting the GPS location, the measurements more accurately represented the actual path. It is possible that not all of the configurations were properly set on the mobile GPS sensors. In addition, the manufacturer recommends that the GPS sensors be placed vertically on a wall for best coverage. Not only were the GPS stationary beacons placed on the ground, but also there was a lot of furniture around the room that could be adding noise to the system. The manufacture recommends a large open area.

7.1.4 Encoder

The decision was made that it would not be feasible to implement the encoders in this iteration of the vehicle. The initial encoders that were chosen to be the most cost-effective options do not fit on the car and given that the motors were not replaced and thereby the motor housing was not replaced, it did not make sense to change the design of the motor housing just to make space for

the encoders. It makes more sense to integrate new encoders when new motors and motor controllers are integrated. The current motors already have encoders included.

7.2 Controller Testing

In section 5.4 we developed preliminary control diagrams that we will further refine to eventually implement on the vehicle. The major milestones of testing for the controllers are: the testing of the vehicle model, the simulation of the controllers, and the implementation of the controllers. Testing the vehicle model will be composed of experimentally running the car under fixed inputs and then simulating the vehicle model under the same inputs and then comparing the results. One specific example is longitudinal acceleration. The car can be given a constant throttle input and be timed to test how long it takes for the car to travel a certain distance. The simulation tests of the controllers will be very similar except for the simulation will not be arbitrary, but rather simulations of the specific controllers themselves. At this point it is expected that because of the tests performed on the vehicle model and its subsequent tuning, the simulation of the control algorithms will be representative of the actual performance of the vehicle. The last tests will be running the control algorithms on the vehicle and tuning the controller to yield the best results.

7.2.1 Lane Keeping Control

In order to implement the lane keeping model on the SSIV, a few parameters had to be configured including the look ahead distance, the gains, the image size, and the pixel displacement transformation. The look-ahead distance affects how fast the car reacts to a curve or turn on the road. If the look-ahead distance is too short then the car might react a bit late, but if the distance is too early the car might react too early (take a turn before the start of the curve). Also, it was found that choosing too large of a look ahead distance produced less reliable lane line readings given that the lanes thin out as they appear further back in the image. A final look-ahead distance of roughly 15 inches from the car front bumper was chosen. The look-ahead distance options were partly constrained because of the placement of the LiDAR sensor which partially obstructs the bottom view of the camera. For the fifteen-inch look-ahead distance it was important to calculate the distance in inches corresponding to the distance in pixels. In other words, the width of one pixel was needed to determine how far the car was from the centerline based on how many pixels to the left or right of the centerline the car was facing. A thin strip of paper of known width was put at various locations of the image to determine the corresponding thickness in pixels. For an image of size 320X240, at a look-ahead distance of 15 inches, the pixel width was 7 pixels for each inch width. Images larger than 320X240 demonstrated lags in the lane detection algorithm. Initially the gains of the controller were tuned in order to improve the performance of the controller. However, the lane keeping performance did not improve too much with changes in the gain. The lane keeping controller was found to be very sensitive to noise in the image. A few filters were considered and implemented including a filter to remove outliers and a filter to ignore non-white lines. The noise in the images caused the car to see false lane lines that triggered the steering. The lane keeping controller, although successful in maintaining the car in the lane (most of the times), was not robust in the sense of keeping the car stable at the centerline. It is recommended that some additional filtering in the image processing

be added. For example, it would probably help quite a bit to have a Kalman filter to try to predict the state of the lane line offset and lane line angle.

7.2.2 Yaw Control

As described in the Design Verification Plan, the testing of the yaw controller consisted of simulation testing and on-the-car testing. In simulation, the controller plant was the vehicle model. It was set up to output the same values that would have come out of the IMU. These values were then manipulated to produce the error states in position, lateral velocity, yaw angle, and yaw rate as described in section 6.3.2. Inputs of constant desired yaw rate and speed were given. The yaw angle and steering behavior as a result of the simulation is shown in Figure 107 below.

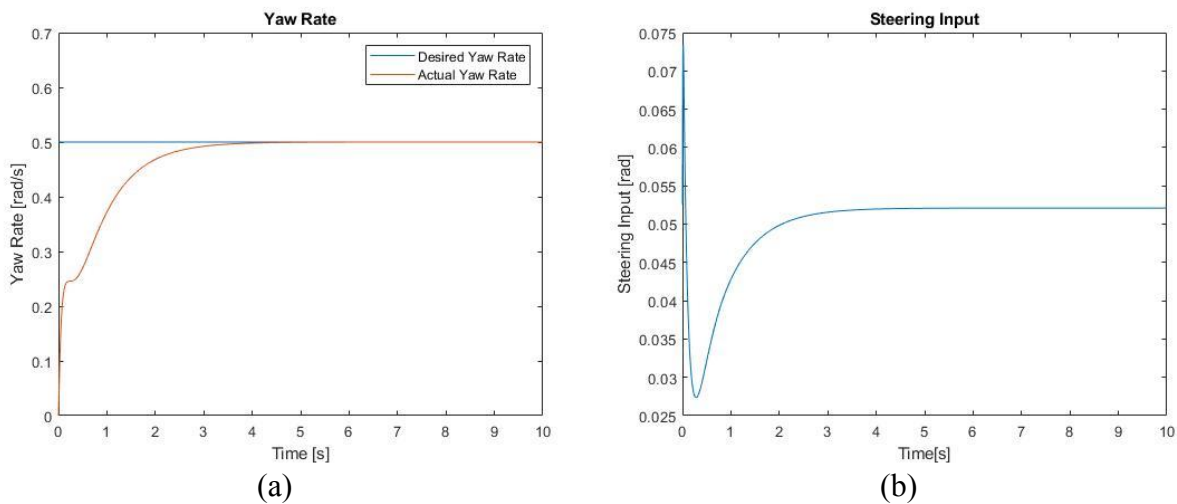


Figure 107. (a) Actual and desired yaw rate as a result of a simulation of the yaw controller. (b) Steering input as a result of a simulation of the yaw controller.

As shown in Figure 107 (a), the actual yaw rate meets the desired yaw rate within approximately three seconds. Figure 107 (b) shows the steering input that was necessary to achieve that yaw rate with the set desired speed of the vehicle.

As explained in section 6.3.2, we were unable to implement the yaw controller with the positional error and lateral velocity error states included. Figure 108 below shows the results of the simulation with those states excluded.

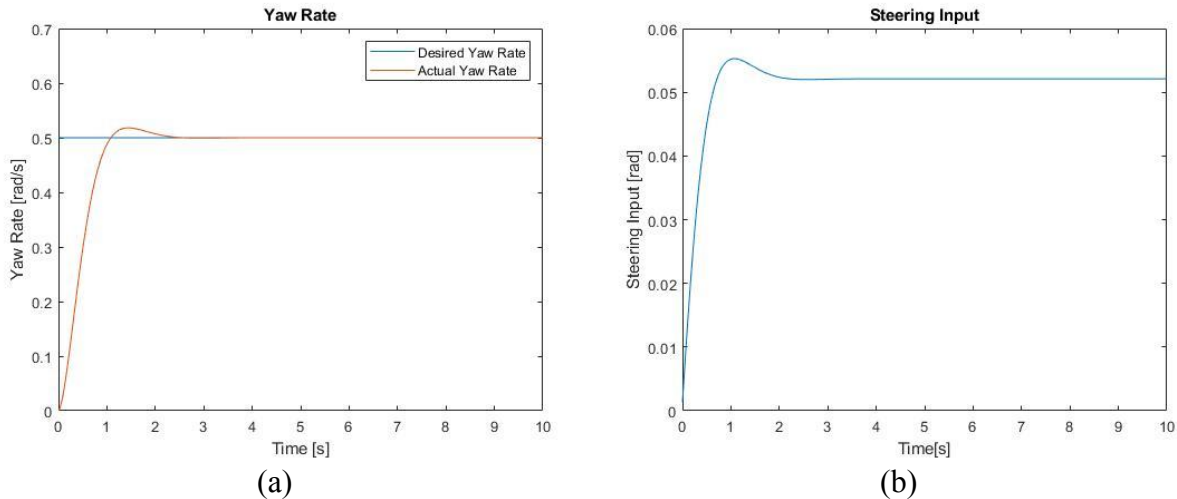


Figure 108 (a) Actual and desired yaw rate as a result of a simulation of the yaw controller with the positional error and velocity error states excluded.
 (b) Steering input as a result of a simulation of the yaw controller with the positional error and velocity error states excluded.

The exclusion of the states appears to change the nature of the response slightly—it appears to be slightly underdamped than overdamped. However, the desired yaw rate is still achieved at around the three second mark, and the value of the steering input remains unchanged. While the nature of the response may have changed, the controller is still commanding the vehicle to perform the same behavior as before.

The next test that was performed for the yaw controller was the vehicle test. In this test, the yaw controller was run on the car. Data was gathered from the IMU in order to assess the results. The inputs were much the same as in the simulation: a constant throttle command and a constant desired yaw rate. It was necessary to match up the desired yaw rate with the throttle command—the car couldn't meet a certain desired yaw rate if the throttle wasn't high enough. When too high of a desired yaw rate was matched up with too low of a throttle input, the car would saturate the steering angle. As error built up, it would then switch the steering over to the other side, creating a sawing effect. Once the throttle was increased or the desired yaw rate was decreased, this effect was eliminated, and the car drove in a circle for the duration of the simulation. Figure 110 below shows the results of the simulation.

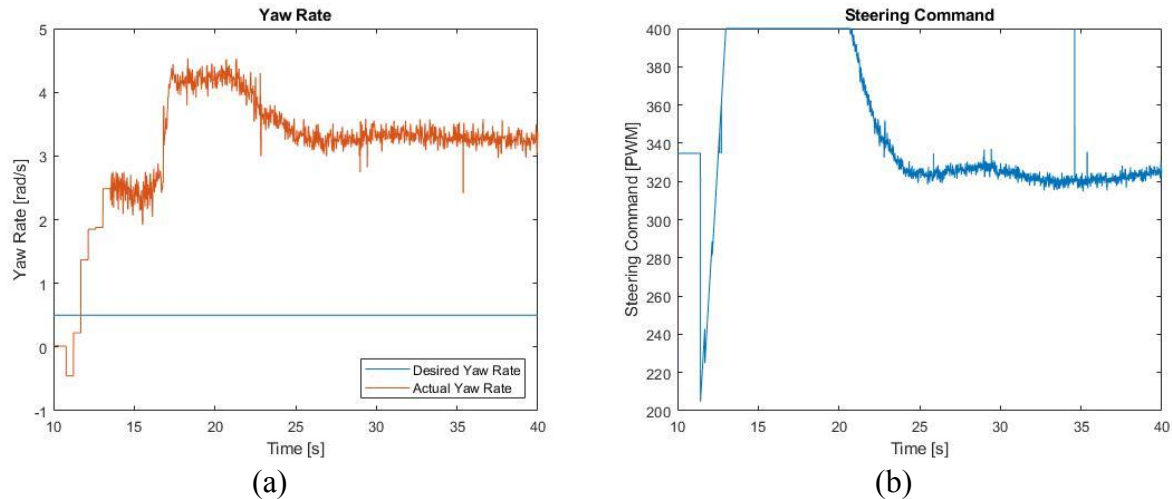


Figure 109 (a) Actual and desired yaw rate as a result of on-the-car test of the yaw controller with the positional error and velocity error states excluded.

(b) Steering input as a result of on-the-car test of the yaw controller with the positional error and velocity error states excluded.

As shown in Figure 109 (a), there is quite a bit of steady-state error in the yaw rate. The achieved yaw rate is approximately six times that of the desired yaw rate. There are a number of potential reasons for this. The first is that we had quite a bit of trouble getting the IMU to output data that intuitively made sense. The measured yaw rate of the vehicle was off by a factor of approximately 10. The reason the yaw rate is only off by a factor of 6 in the results is that the yaw angle error was having a much greater effect on the controller than the yaw rate error. The yaw angle error minimized the influence of the bad yaw rate readings and brought the behavior of the car closer to what was desired. The second reason is simply gain tuning. While the controller used LQR gains in order to attempt to reach the desired yaw rate, the vehicle model doesn't even come close to mirroring the real system, so it's understandable if in practice those gains had to be tuned. While we attempted to do some of this in our testing, we spent a lot of time making sure the IMU was outputting correct data, since that was a more primary concern. As such, we ran out of time to do painstaking gain tuning.

Future work on the yaw controller is recommended in section 9.2.

7.2.4 Vehicle Model

The idea behind the vehicle model testing was to compare the behavior of the car as predicted by our bicycle model with the behavior of the car in reality. We designed a simple test where the car was given a series of throttle and steering commands, and its position, velocity, and acceleration were plotted. These plots would be compared to those generated by the vehicle model with the same commands. Unfortunately, it took longer than planned to get the IMU and indoor GPS sensors integrated, and the controller testing took priority as we ran out of time. Thus, this test was never run. We are, however, relatively happy with the performance of our controllers, and confident that the bicycle model performed well.

7.3 Usability Testing

One of the main requirements for the SSIV is that it is easy to use and accessible to students without a mechatronics background. In order to determine if we had met this requirement, we wanted to write a short, simple lab for the car and gather a group of engineering students to perform that lab in groups. We then wanted to get their feedback on the accuracy of the instructions, the performance of the car in the lab environment, and whether they ran into any obstacles.

Unfortunately, it took longer than we expected to get the firmware, sensor integration, and controller design to the point that we could hand the car off to other students in order to be able to perform this test. We also know that the current form of the SSIV is not its final form, so we didn't want to be doing any premature testing, when we know that students this summer and going into next year will continue to work on the SSIV. We would, however, recommend that this test be performed before the SSIV is implemented in a classroom setting.

8 Project Management

Because this project deals with so many different technical fields in mechatronics, controls, and vehicle dynamics, each of the team members had a technical specialty that focused the areas of their work.

Sarah focused on vehicle dynamics and control theory. She took two vehicle dynamics classes this year, as well as an advanced control theory class, AERO 553, which the other team members also took. She used her vehicle dynamics knowledge in her investigation of implementing different motors, as detailed in section 5.1. She used her acquired knowledge of advanced control theory and vehicle dynamics in her design of the yaw controller.

Paul was primarily concerned with sensor integration on the vehicle. He investigated which sensors needed to be added to provide the functionality we needed, and then worked on integrating them into the existing vehicle. He did a lot of work making sure that the sensor feedback was useful and accessible. He also contributed the lane keeping control algorithm, which used computer vision from camera data in order to keep the vehicle in between two lane lines. Paul was also instrumental in the modifications made to the Raspberry Pi firmware, along with Dominic. Paul implemented the LCD screen and user interface.

Dominic worked on improving the firmware that was begun by the last team. He streamlined the relationship between the Raspberry Pi and the Teensy to make the system friendly to future developments, and more modular. He also did a lot of the sensor integration on the firmware side and made sure that the IMU was outputting correct data. He ensured that timing requirements for each of the controllers were met and dove in to solve any problem that came up. Dominic also designed the Adaptive Cruise Controller and worked really hard to try to get the motor controllers into torque mode so it could be implemented. He was also instrumental in the evolution of the yaw controller. Dominic also provided extensive documentation on the operation of the system, so further teams can learn from our work this year.

In addition to each of our technical roles, each of us also had a role in how the project was managed. This was to evenly distribute responsibility for project communication and deliverables.

The role of team treasurer was performed by Sarah, who handled keeping track of the team's budget. She also performed the role of the team editor, who made sure that every document that the team produced was formatted correctly, organized appropriately, and well-written.

The project timeline, included in Appendix O, gives a rough outline of the tasks that were completed in order to meet major deliverables.

We will be presenting a paper that we wrote about the work that we did on this project at the annual American Society of Engineering Educators (ASEE) conference from June 24th – June 27th, 2018. We will be giving a 20-minute presentation on the work that we did this year, and how it connects to the future of university engineering education. Our paper will be published in the ASEE conference proceedings.

9 Conclusion

In this document, the final design, manufacturing, and test results for the small scale intelligent vehicle are documented. This documentation sums up all the work that the ProgreSSIV team: Sarah De Rosier, Dominic Riccoboni, and Paul Rothhammer-Ruiz, have completed on the Small Scale Intelligent Vehicle (SSIV) project. Appendix O summarizes the timeline that the team members went through in their work on this project. The following sections include recommendations for the next students who work to make the SSIV a finalized platform for use in the course on intelligent vehicles.

9.1 Firmware and Simulink Recommendations

We made significant progress on the Teensy Firmware and Simulink SPI communication. We were not, however, able to get to complete everything we wanted to before the end of Senior Project. Below are several issues and recommendations:

- Currently, Simulink timestep overruns are a possibility. We have experienced a distortion of the external mode simulation clock in computationally intensive models such as the Lane Keeping image processing lab. We have not figured out a way to detect when these occur in Simulink, nor have we benchmarked how much it takes to overload the Raspberry Pi computationally.
- Startup messages in Simulink that prompt the Teensy to initialize the motor controllers, the servo, the IMU, etc. are part of an enabled subsystem that executes for the first ten seconds of a Simulink model before it switches over to the control system subsystem. If this could be done in a more elegant way, perhaps using Simulink preload and post load functions, that would be an improvement. This way, the startup messages are hidden away and only send once.

- At the moment, when a Simulink model initializes the system, it alters the state of the Teensy Register map, the motor controllers, the IMU, etc. If a new Simulink model is to be ran after the first, the system is in an indeterminate state – a state that depends on what the previous model altered. We recommend finding a way to make the initial state of the system for each new execution of a model the same. This would involve writing new Teensy commands such as ones that shut off the motors and motor controllers.
- Related to the previous point, the current state of the CAN driver firmware is very limited. There are many CAN messages available for the Maxon motor controllers that provide a wide variety of functions. Only a few of these have been implemented and none of them are documented thoroughly. A result of this is that the only thing we can do with the motor controllers is start them up and send them a “Velocity Profile Mode” velocity setpoint. Further developing the CAN communication firmware and through documentation of its available functions will allow for things like reading the encoders and configuring the operating mode to control torque output instead of a velocity profile output.
- The Motor controller startup code has no error handling. Nor do any of the peripherals. A means of giving the Raspberry Pi feedback on the state of each peripheral would allow for better development of the systems operational procedures in the future.
- We recommend making the read registers on the Teensy read-only. It is possible now to write to a read register accidentally. Adding in some error checking code to the *spi0_isr* interrupt service routine will accomplish this.
- The inertial measurement unit *gyro_z* value is showing an abnormal magnitude in the Simulink external mode scopes. The other gyro data was not tested for inaccuracies. We know this because we drove the vehicle in a circle at a constant rate and timed a single rotation. The value we estimated was about 5 times smaller than that measured by the scope. This may be a bug somewhere in Simulink or on the Teensy or it may simply be due to a lack of IMU calibration.
- We have not yet gone through the calibration procedure for the IMU. Creating a detailed procedure for how to calibrate the IMU as it is integrated on the SSIV will be beneficial even if it is not the cause for the strange *gyro_z* reading.
- The Simulink subsystems are supplied as a Simulink file. To create a new control system, one either has to rename that file and save it anew or merge it and their controller somehow. This is tedious, and we suggest making a custom block set that can be accessed in the same way as any other Simulink block in a model.

9.2 Controller Recommendations

While we made a lot of progress on the controllers that we wanted to implement on the car, there is still some work to do on each. For the lane keeping controller, it would perform better if some filtering were implemented. Some filtering to handle noise and help the camera deal with changes in lighting conditions would greatly improve its performance. Additionally, we would recommend expanding its capabilities with improvements such as being able to navigate using only one lane line, being able to perform lane-changing maneuvers, detecting lane line colors that are not white, and performing some action when no lane lines are detected.

For the yaw controller, there are some improvements that could be made to the current system. First, the controller would have to be run in tandem with a cruise controller, in order to keep the car at a constant speed while the controller is running. In order to improve its performance, it would be good to figure out some way to measure the positional and velocity errors that had to be left out of the final controller as states. The positional error could potentially be measured using a computer vision algorithm, as with the lane keeping controller. The velocity error could potentially be captured using the ultrasonic GPS—while the GPS outputs positional data, and would therefore need to be differentiated, it is much more precise than the IMU, and so differentiation of the signal could still produce accurate results. Finally, the controller in its current state could greatly benefit from systematic gain tuning: modifying the LQR gains in order to eliminate the scaling effects discussed in section 7.3.2.

In order to implement actual stability control, however, one would need to design a traction control system, and combine that with a yaw moment controller. This could prove to be quite challenging, especially with the limitation that if this controller is to be a part of the course on intelligent vehicles, it has to perform using relatively simple control techniques. The yaw moment controller is also restricted to a steer-by-wire system, such that the system is not too complex for students to understand. Work is currently being done by graduate student Winston Wight to implement a torque-vectoring stability controller. While it will be exciting to see the progress of his work, a different stability controller will have to be implemented if it is to be used in the course, as torque-vectoring stability control will have to make use of a four-wheeled vehicle model and nonlinear controls.

The Adaptive Cruise Controller was unable to be tested on the car, as discussed in section 5.4.1. Once the motors are in torque mode, however, testing should be done on the step response of the vehicle with torque inputs. These data can then be used to characterize the car's actual acceleration time constant so that the simulation of the headway controller can more closely match the physical system. For the arbitrary selection of λ , a better method would include computing the closed loop transfer function of the system and use root locus techniques to see how altering λ affects the roots, and ultimately the behavior of the system. In designing the λ gain then, the root locus design criteria would have to be balanced with the string stability constraint.

The lane detection is still being refined to optimize the algorithm and to determine the ideal look-ahead distance of the camera. The lane detection algorithm is being tested as its being developed. Up until now the testing has been on a gray concrete floor with white duct tape used as the lane

lines. The testing needs to be expanded to consider different environments. The course may have a “lane line kit” that is used for the lane keeping lab and the lane detection algorithm can be tuned to be optimized for this kit, but it would be better if the lane detection algorithm is robust enough to be suitable in a number of environments.

10 Works Cited

Adafruit Industries. “Adafruit VL6180X Time of Flight Distance Ranging Sensor (VL6180).” *Adafruit industries blog RSS*, www.adafruit.com/product/3316.

“Adaptive Cruise Control System Using Model Predictive Control - MATLAB & Simulink.” MathWorks, MathWorks, www.mathworks.com/help/mpc/examples/design-an-adaptive-cruise-control-system-using-model-predictive-control.html.

Barr, Michael. “Firmware Architecture in Five Easy Steps.” *Embedded*, 21 Sept. 2009, www.embedded.com/design/prototyping-and-development/4008800/Firmware-architecture-in-five-easy-steps

“Bicycle Model.” *Bicycle Model*, University at Buffalo, code.eng.buffalo.edu/dat/sites/model/bicycle.html.

Esmailzadeh, E., et al. “Optimal yaw moment control law for improved vehicle handling.” *Mechatronics*, vol. 13, no. 7, Sept. 2003, pp. 659–675., doi:10.1016/s0957-4158(02)00036-3.

Gonzalez, Rafael; Woods, Richard. *DIGITAL IMAGE PROCESSING, 3rd Ed.* Print.

Hewson, P. (2005) Method for estimating tyre cornering stiffness from basic tyre information. *Proceedings of the Institution of Mechanical Engineers Part D Journal of Automobile Engineering*, 219 (12). pp. 1407-1412. ISSN 0954-4070

Hoehener, Daniel, et al. “Design of a lane departure driver-Assist system under safety specifications.” 2016 IEEE 55th Conference on Decision and Control (CDC), 29 Dec. 2016. IEEE, doi:10.1109/cdc.2016.7798632.

Maxon Motor - Online Shop, MaxonMotorUSA.com, www.maxonmotorusa.com/maxon/view/category/motor?etcc_med=Product.

Kural, Emre, and Bilin Aksun Guvenc. “Model Predictive Adaptive Cruise Control.” *Systems Man and Cybernetics*, 22 Nov. 2010. IEEE.

Lin, Yu-Chen, et al. “Adaptive neuro-Fuzzy predictive control for design of adaptive cruise control system.” 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC), 3 Aug. 2017, doi:10.1109/icnsc.2017.8000187.

Marino, Riccardo, et al. “A nested PID steering control for lane keeping in vision based autonomous vehicles.” 2009 American Control Conference, 10 July 2009. IEEE, doi:10.1109/acc.2009.5160343.

Miley, J., Phillips, E. and Grant, C. (2017). Final Design Report MLAREN – Small-Scale Intelligent Vehicle Design Platform. [online] California Polytechnic State University, San Luis Obispo. Available at:

<http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1424&context=mesp> [Accessed 25 Oct. 2017].

Mintzlaff, Johannes. "Longitudinal Dynamics." ME 416. California Polytechnic State University San Luis Obispo, San Luis Obispo. September - October 2017. Lecture

Mousavinejad, Iman Eman, et al. "Control strategies for improving ground vehicle stability: State-of-the-Art review." 2015 10th Asian Control Conference (ASCC), 10 Sept. 2015. IEEE, doi:10.1109/ascc.2015.7244795.

National Instruments. *PID Theory Explained*. National Instruments, 2011.

Noxon, Nikola. "A MODEL PREDICTIVE CONTROL APPROACH TO ROLL STABILITY OF A SCALED CRASH AVOIDANCE VEHICLE." *California Polytechnic State University*, California Polytechnic State University, 2012.

Rajamani, Rajesh. *Vehicle Dynamics and Control*. New York: Springer, 2012. Print.

Raspberry Pi, "RASPBERRY PI TOUCH DISPLAY, The 7" touchscreen monitor for Raspberry Pi", Raspberry Pi Accessories, Buy Now, www.raspberrypi.org/products/raspberry-pi-touch-display/

Freescale Semiconductor, Inc. (2015). *K66 Sub-Family: Reference Manual*. Document Number K66P144M180SF5RMV2 Revision 2

Schramm, Dieter, et al. *Vehicle Dynamics Modeling and Simulation*. Springer Berlin, 2014.

Siegwart, Roland, et al. *Introduction to autonomous mobile robots*. MIT, 2011.

Singh, Kanwar. (2014). Estimation of tire-road friction coefficient and its application in chassis control systems. *Systems Science & Control Engineering: An Open Access Journal*. 3. . 10.1080/21642583.2014.985804.

Sivaji, V. V., and M. Sailaja. "Adaptive Cruise Control Systems for Vehicle Modeling Using Stop and Go Maneuvers." *International Journal of Engineering Research and Applications*, vol. 3, no. 4, 2013. IEEE.

Son, Young Seop, et al. "Robust Multirate Control Scheme With Predictive Virtual Lanes for Lane-Keeping System of Autonomous Highway Driving." *IEEE Transactions on Vehicular Technology*, vol. 64, no. 8, 8 Sept. 2014, pp. 3378–3391. IEEE, doi:10.1109/tvt.2014.2356204.

Stabile, Cameron, Mathworks Support "Generating code for Simulink model using Raspberry Pi Support package" message to Dominic Riccoboni. 1 February 2018. E-mail.

Stevens, Thomas Fitzgerald. "A LiDAR Based Semi-Autonomous Collision Avoidance System and the Development of a Hardware-in-the-Loop Simulator to Aid in Algorithm Development

and Human Studies.” *California Polytechnic State University*, California Polytechnic State University, 2015.

J. Taylor, Camillo & Košecká, Jana & Blasi, Robert & Malik, Jitendra. (1999). A Comparative Study of Vision-Based Lateral Control Strategies for Autonomous Highway Driving. I. J. Robotic Res.. 18.

“U.S. Software System Safety Working Group.” Massachusetts Institute of Technology, 5th Meeting of the U.S. Software System Safety Working Group, 12 Apr. 2005.

Wang, Wenwei, et al. “Lateral stability control of four wheels independently drive articulated electric vehicle.” 2016 IEEE Transportation Electrification Conference and Expo (ITEC), 25 July 2016, doi:10.1109/itec.2016.7520218.

Yoon, Jong-Hwa. “A Cost-Effective Sideslip Estimation Method Using Velocity Measurements from Two GPS Receivers.” *IEEE*, 2013.

Yuejian, Wang, et al. “Longitudinal Acceleration Tracking Control of Low Speed Heavy-Duty Vehicles .” *TSINGHUA SCIENCE AND TECHNOLOGY*, vol. 13, no. 5, Oct. 2008. IEEE.

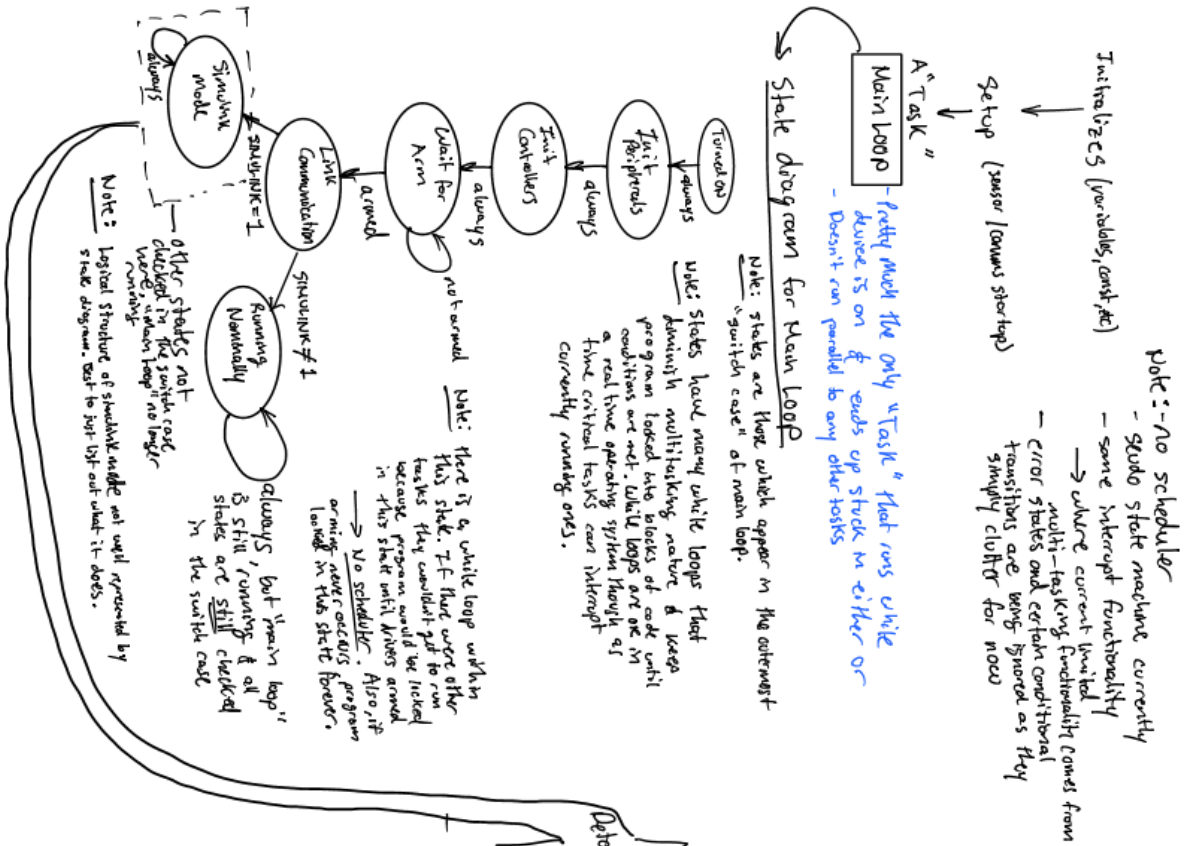
Zhang, Chengjie, et al. “The acceleration slip regulation control for two-Wheel independent driving electric vehicle based on dynamic torque distribution.” 2016 35th Chinese Control Conference (CCC), 29 Aug. 2016. IEEE, doi:10.1109/chicc.2016.7554286.

Zhongpu, XIA, and ZHAO Dongbin. “Hybrid Feedback Control of Vehicle Longitudinal Acceleration.” *IEEE*, 27 July 2012. IEEE.

Appendices

1. Appendix A: Current Firmware Code Outline
2. Appendix B: Quality Function Deployment
3. Appendix C: Functional Decomposition Diagram
4. Appendix D: Mechanical Ideation Sketches
5. Appendix E: Mechanical Pugh Matrix
6. Appendix F: Sensor Pugh Matrix
7. Appendix G: Controller Design Pugh Matrices
8. Appendix H: User Interface Pugh Matrix
9. Appendix I: Motor Selection Code
10. Appendix J: Vehicle Model
11. Appendix K: Yaw Controller
12. Appendix L: Mastermind State Transition Diagram
13. Appendix M: Design Hazard Checklist
14. Appendix N: Teensy Register Map
15. Appendix O: Project Timeline

Appendix A: Current Firmware Code Post-Summer 2017 Outline



Details

Simulink Model

A couple initial actions are run then

while loop - gets stuck in here rather than going back through main loop fast & checking status of all the states in the outermost switch case

1) SPI message - send value most recently transmitted from Raspberry Pi via SPI to its respective hardware → If lost transfer urgent monitor or some thing, restoring hardware. This approach provides the potential for unpredictable findings of values in the control loop on loss of data.

2) Sensor Data - Reads each sensor one by one, subtracts & stores it in the sensor array. → sensor will ask for these one at a time & via SPI interrupt service routine. No guarantee the sensor was updated by this yet.

3) write - Stores values for steering & throttle that have been written to the telemetry via SPI into array. → Not sure the reason for this. May be falling SPI message with just data to send.

difficult to encompass the logical structure in a diagram architecture not very uniform or modular. Need knowledge of the entire program to add any functionality and tuning not very deterministic.

Appendix B: Quality Function Deployment

QFD: House of Quality
 Project:
 Revision:
 Date:

Correlations

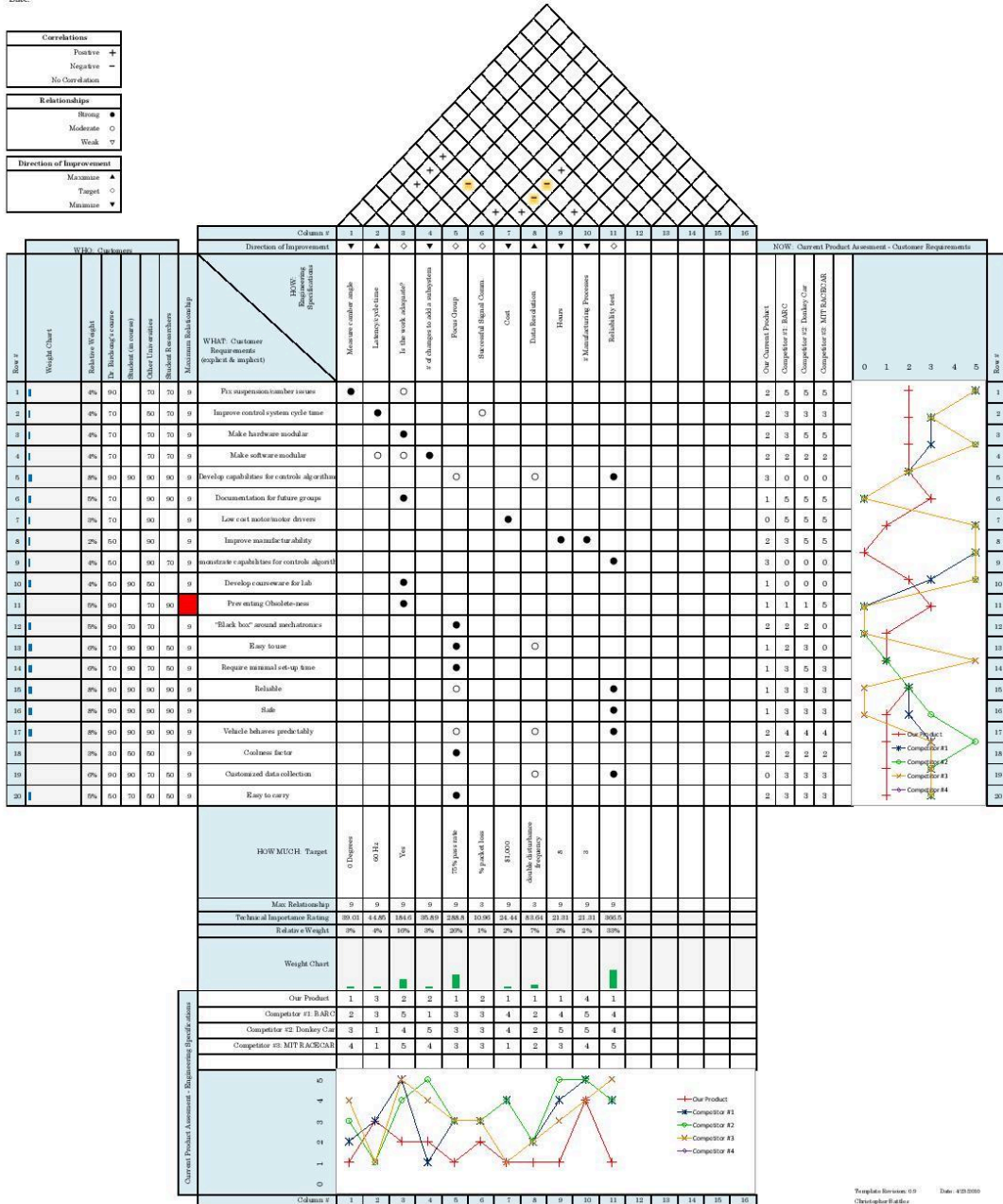
- Positive +
- Negative -
- No Correlation

Relationships

- Strong ●
- Moderate ○
- Weak ▽

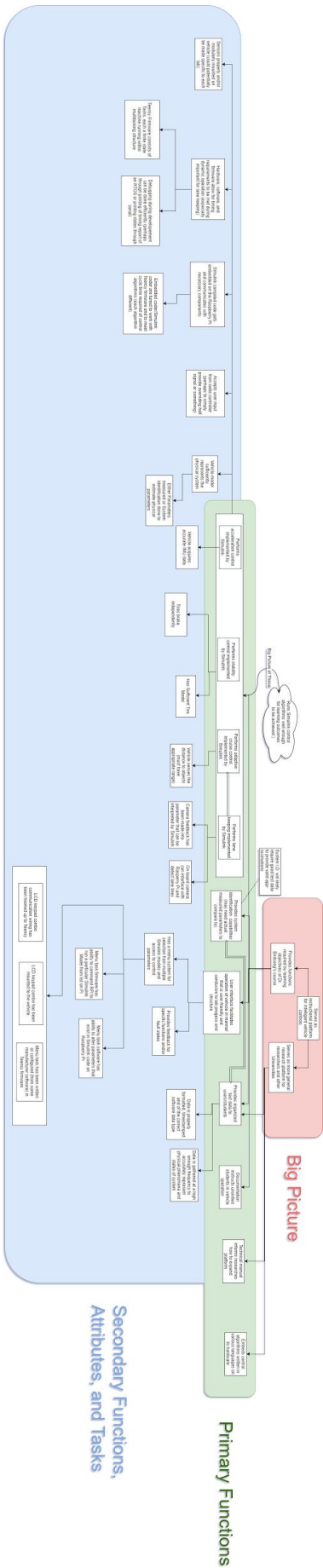
Direction of Improvement

- Maximize ▲
- Target ○
- Minimize ▼



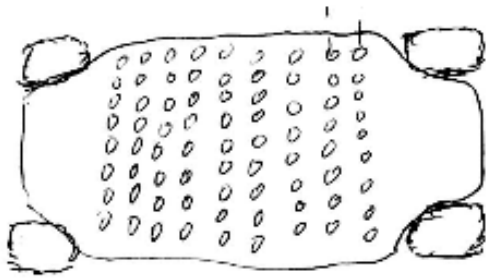
Template Revision: 0.9 Date: 6/20/2008
 Christopher Radtke

Appendix C: Functional Decomposition Diagram

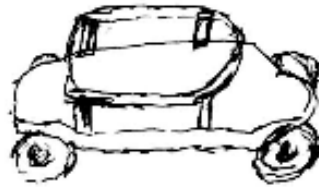


Appendix D: Mechanical Ideation Sketches

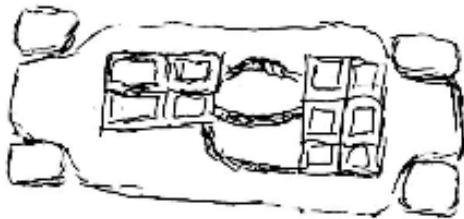
Idea 1: "Peg Board"



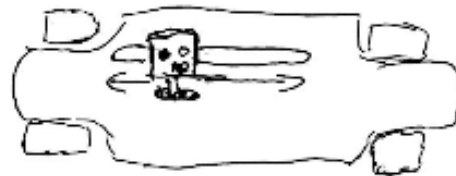
Idea 2: "Double Decker"



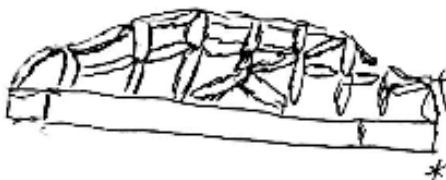
Idea 3: "Modular Housing"



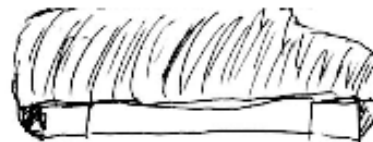
Idea 4: "Sensors on Rails"



Idea 5: "Roll Cage"



Idea 6: "Fiber Glass Shell"



Current System



Appendix E: Mechanical Pugh Matrix

Criteria	Weight	Current Car	Pegboard	Double Decker	Modular Housing	Sensors on Rails	Roll Cage	Shell
Electronics securely mounted	5	3	5	3	5	3	4	3
Protection from outside	5	0	0	2	5	0	5	5
Wiring not exposed	5	0	0	2	2	0	5	5
Sensors have good range	4	2	4	5	3	3	4	4
Sensors can be adjusted/mounted	3	1	5	3	3	4	2	2
Easy to add/fit new electronics	3	2	5	3	3	1	2	2
Easy to manufacture	5	5	4	3	2	3	1	1
Cheap	3	5	4	3	2	3	1	1
Durability	4	3	4	2	4	2	5	4
Coolness	2	2	3	4	3	3	5	5
Σ		73	125	113	124	83	131	127

Appendix F: Sensor Pugh Matrix

Criteria	Weight	Nerian Stereo Camera	Zed Stereo Camera	DUO DDK	Hokuyo LiDAR	Indoor GPS	Optical Flow	Encoder
Integrates well with current software/hardware	5	-	-	-	0	+	-	+
Sensors used in Industry (student gain valuable skill interfacing with sensor)	3	+	+	+	+	+	+	+
Cost	2	-	-	-	-	0	+	+
Reliable Data	4	+	+	+	+	+	0	+
Long term support for sensor	3	+	+	+	+	+	0	0
Compatible with MATLAB/Simulink	2	+	+	0	0	+	+	+
Easy for students to work with	1	-	-	-	0	0	0	+
Σ		4	4	2	8	17	2	17

Appendix G: Controller Design Pugh Matrices

Pugh Matrix for the selection of an acceleration controller

		Method				
Criteria	Weight	PI D	PID + Fuzzy	PID + Fuzzy + PSO	MMC + PID	MMC + SMC
Easy to implement	0.3	D	-0.3	-0.3	-0.3	-0.3
Teachable	0.2	A	-0.2	-0.2	-0.2	-0.2
Smooth operation	0.2	T	0	0.2	0.2	0.2
Ability to handle non-linearity	0.1	U	0.1	0.1	0.1	0.1
Low error	0.1	M	-0.1	0.1	0.1	0.1
Rapid response	0.05	o	0	0	0.05	0.05
Low overshoot	0.05	o	0	0.05	0.05	0.05
		Σ	-0.5	-0.05	0	0

Pugh Matrix for selection of Adaptive Cruise Controller

o		Method				
Criteria	Weight	PI D	MP C	ANFP C	MP C	ANFP C
Easy to implement	0.2	D	-1	-1	-0.2	-0.2
Teachable	0.3	A	-1	-1	-0.3	-0.3
Ability to handle non-linearity	0.1	T	1	1	0.1	0.1
Distance control	0.1	U	1	1	0.1	0.1
Speed control	0.15	M	1	1	0.15	0.15
Acceleration control	0.15	o	1	1	0.15	0.15
				Σ	0	0

Pugh Matrix for selection of Lane Departure Assist System controller

○		Method						
Criteria	Weight	PID	Controlled Invariance	State Feedback	State Feedback w/ Integrator	Controlled Invariance	State Feedback	State Feedback w/ Integrator
Easy to implement	0.15	D	-1	-1	-1	-0.15	-0.15	-0.15
Teachable	0.2	A	-1	0	0	-0.2	0	0
Ability to handle non-linearity	0.1	T	1	1	1	0.1	0.1	0.1
Lateral offset minimization	0.2	U	1	1	1	0.2	0.2	0.2
Smooth activation	0.15	M	1	1	1	0.15	0.15	0.15
					Σ	0.1	0.3	0.3

Second Pugh Matrix for selection of Lane Departure Assist System controller

Criteria	Weight	State Feedback	State Feedback w/ Integrator	State Feedback w/ Integrator
Easy to implement	0.15	D	0	0
Teachable	0.2	A	0	0
Ability to handle non-linearity	0.1	T	0	0
Lateral offset minimization	0.2	U	1	0.2
Smooth activation	0.15	M	0	0
			Σ	0.2

Pugh Matrix for selection of stability controller

○		Method				
Criteria	Weight	PI D	Fuzz y	Optima l	Fuzz y	Optima l
Easy to implement	0.2	D	-1	-1	-0.2	-0.2
Teachable	0.2	A	-1	0	-0.2	0
Ability to handle non-linearity	0.1	T	1	1	0.1	0.1
Yaw moment minimization	0.15	U	0	1	0	0.15
Yaw rate minimization	0.1	M	1	1	0.1	0.1
Slip angle minimization	0.15	○	1	1	0.15	0.15
Smooth activation	0.1	○	-1	1	-0.1	0.1
				Σ	-0.2	0.4

Appendix H: User Interface Pugh Matrix

Criteria	a	b	c	d	e	f	g
Ease of programming menu/printing/interacting with Teensy	D	0	0	0	-	-	-
User friendly		0	0	0	0	0	0
Time to integrate/manufacture	A	0	-	-	-	-	-
Power consumption		0	-	-	0	0	+
Amount that can be displayed at once	T	0	0	-	0	0	-
How nice does it look and feel		0	+	+	0	0	0
Reliable	U	0	0	0	-	-	-
Cost effective		0	?	?	+	+	+
Ability to change numerical values	M	-	0	0	-	0	0
Σ	0	-1	-1	-2	-3	-2	-2

a) Crystalfontz LCD product with number keypad
b) Crystalfontz LCD product with rotary encoder
c) "Screenkeys" product and Graphic LCD screen
d) "Screenkeys" product and character LCD screen
e) Graphic LCD with directional keys and rotary encoder
f) Graphic LCD with directional keys and number keypad
g) Character LCD with directional and number keypad

Appendix I: Motor Selection Code

Parameters	
m [kg]	4.67
g [m/s ²]	9.8
W [N]	45.766
uk	0.025
Ff [N]	1.14415
Cd	0.4
A [m ²]	0.053904
ρ [kg/m ³]	1.225
θ [rad]	0.05236
a [m/s ²]	1.4
etaDG	0.75
etaEL	0.8
r [m]	0.038

The friction force, f_f was calculated from

$$f_f = \mu_k W .$$

Calculations						
Vmax [m/s]	Drag Force [N]	Grade Resistance [N]	Acceleration Resistance [N]	F _D [N]	T _m [Nm]	ω [RPM]
0.00	0.00	2.40	6.54	10.08	0.13	0.00
0.50	0.00			10.08	0.13	125.65
1.00	0.01			10.09	0.13	251.30
1.50	0.03			10.11	0.13	376.95
2.00	0.05			10.13	0.13	502.59
2.50	0.08			10.16	0.13	628.24
3.00	0.12			10.20	0.13	753.89
3.50	0.16			10.24	0.13	879.54
4.00	0.21			10.29	0.13	1005.19
4.50	0.27			10.34	0.13	1130.84
5.00	0.33			10.41	0.13	1256.49
5.50	0.40			10.48	0.13	1382.14
6.00	0.48			10.55	0.13	1507.78
6.50	0.56			10.64	0.13	1633.43

The drag force was calculated from

$$f_d = 0.5C_D A \rho V_{max}^2,$$

where C_D is the drag coefficient, A is the frontal area and ρ is the air density. The grade resistance was calculated from

$$f_g = W \sin(\theta),$$

where W is the vehicle weight and θ is the angle of incline. An angle of 3 degrees was selected for this study, since the vehicle could realistically experience this angle. The acceleration resistance was calculated using

$$f_a = ma,$$

where m is the vehicle mass and a is the maximum acceleration we are designing for. The total drag on the vehicle, F_D , is the summation of the friction force, drag force, grade resistance, and acceleration resistance. The required torque from the motor was then calculated using

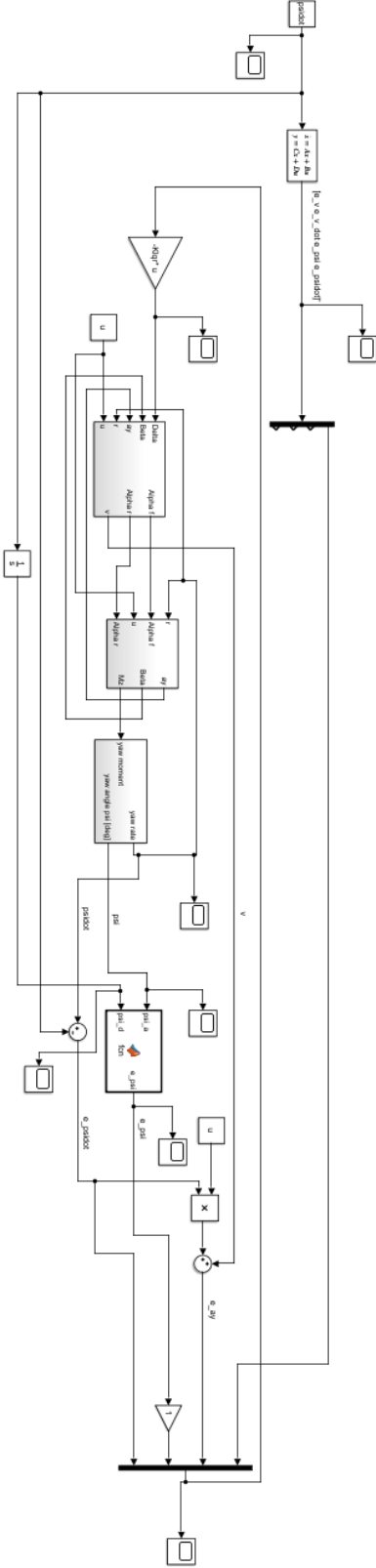
$$T_m = \frac{F_D r}{4\eta_{DG}},$$

where r is the radius of the wheels and η_{DG} is the efficiency of the drivetrain, assumed conservatively here to be 75%. The speed of the motors was found using

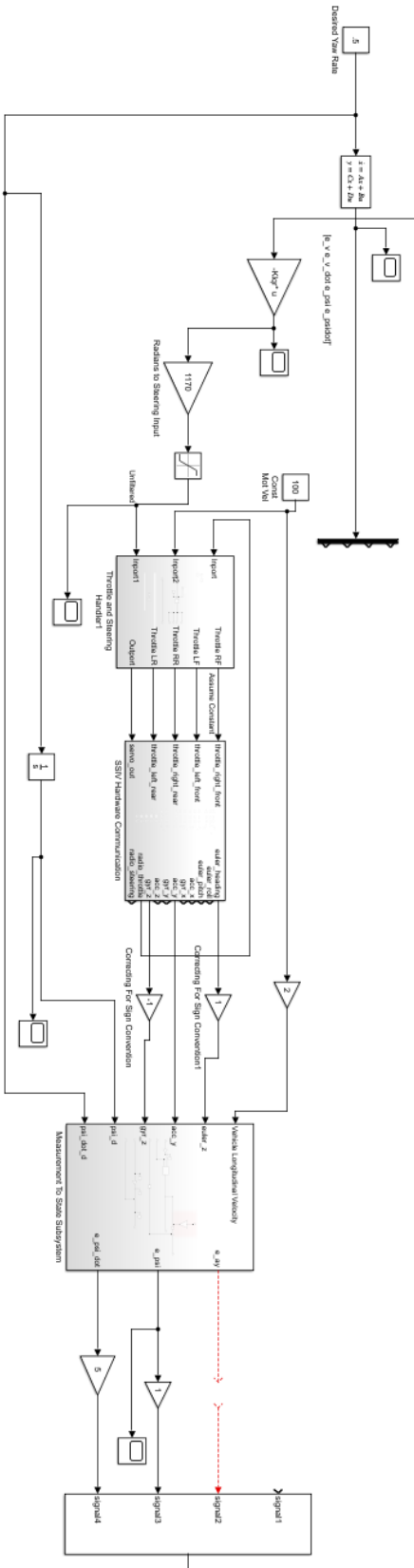
$$\omega = \frac{60V_{max}}{2\pi r}.$$

The required torque and speed of the motor was used in plotting the system curve.

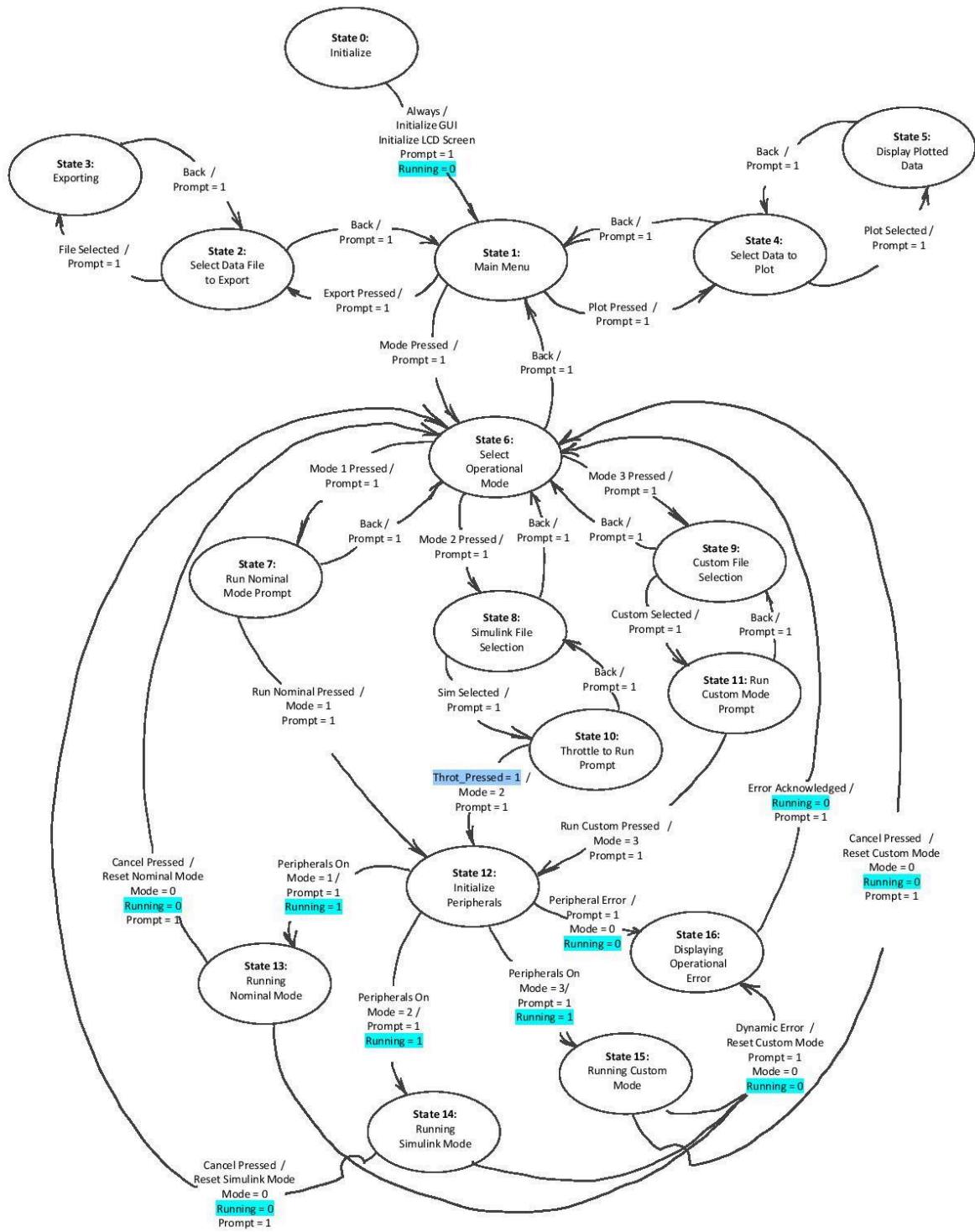
Appendix K: Yaw Controller Simulation Model



Testing Model



Appendix L: Mastermind State Transition Diagram



Appendix M: Design Hazard Checklist

DESIGN HAZARD CHECKLIST

Team: ProgreSSIV

Advisor: Schuster

Date: 11/9/17

Y N

- X c 1. Will the system include hazardous revolving, running, rolling, or mixing actions?
- c X 2. Will the system include hazardous reciprocating, shearing, punching, pressing, squeezing, drawing, or cutting actions?
- X c 3. Will any part of the design undergo high accelerations/decelerations?
- c c 4. Will the system have any large (>5 kg) moving masses or large (>250 N) forces?
- c X 5. Could the system produce a projectile?
- c X 6. Could the system fall (due to gravity), creating injury?
- c X 7. Will a user be exposed to overhanging weights as part of the design?
- c X 8. Will the system have any burrs, sharp edges, shear points, or pinch points?
- c X 9. Will any part of the electrical systems not be grounded?
- X c 10. Will there be any large batteries (over 30 V)?
- c X 11. Will there be any exposed electrical connections in the system (over 40 V)?
- c X 12. Will there be any stored energy in the system such as flywheels, hanging weights or pressurized fluids/gases?
- c X 13. Will there be any explosive or flammable liquids, gases, or small particle fuel as part of the system?
- c X 14. Will the user be required to exert any abnormal effort or experience any abnormal physical posture during the use of the design?
- c X 15. Will there be any materials known to be hazardous to humans involved in either the design or its manufacturing?
- c X 16. Could the system generate high levels (>90 dBA) of noise?
- c X 17. Will the device/system be exposed to extreme environmental conditions such as fog, humidity, or cold/high temperatures, during normal use?
- X c 18. Is it possible for the system to be used in an unsafe manner?
- c X 19. For powered systems, is there an emergency stop button?
- c X 20. Will there be any other potential hazards not listed above? If yes, please explain on reverse.

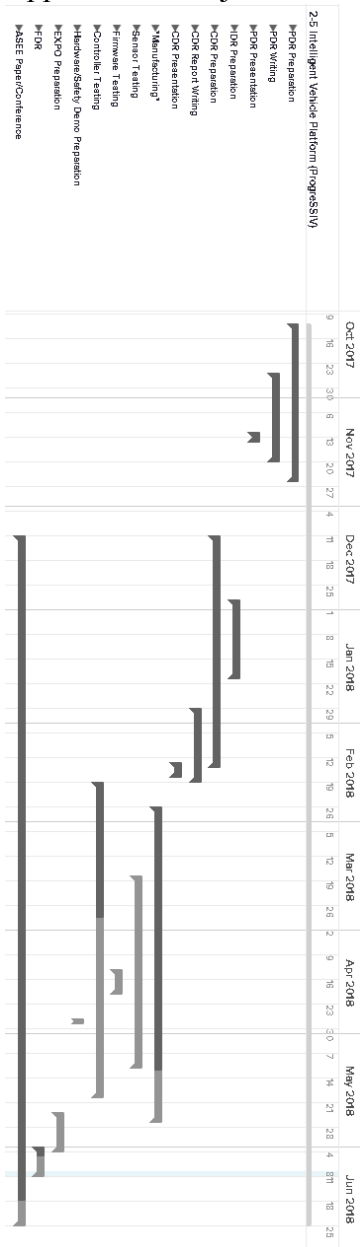
For any “Y” responses, add (1) a complete description, (2) a list of corrective actions to be taken, and (3) date to be completed on the reverse side.

Description of Hazard	Planned Corrective Action	Planned Date	Actual Date
Design contains revolving action at wheels and potential pinch/shear points in structure	Affix warning to car to disconnect battery power while programming or maintaining—forced turning movement only occurs under battery power	5/10/18	
Design can undergo high acceleration due to powerful motors	This is part of intended operation, but we will ensure that the software cannot prompt excessive acceleration without user input	4/10/18	
Design will be able to become an impact hazard	Can install default software limitation and lock maximum speed to controller input levels	4/10/18	
There will be a large battery in the system running at high voltage.	We will purchase a fire-resistant bag for storing and charging, will use a polarized connector to prevent shortening or applying reverse voltage, and will design several possible protective systems to prevent impact or puncture	3/10/18	
It will be possible to use the system in an unsafe manner by driving into people or walls	While this is out of our control, we will develop a user manual with an introduction that discusses the safety hazards that the user should be aware of.	4/10/17	

Appendix N: Teensy Register Map

Register Address	Data Type	Data Name	Description
0	uint8_t	print_registers	Flag that prompts the the name, stored value, and address of every data point in the registers to be printed to the Teensy serial port only once.
1	uint8_t	begin_data_collection	Flag that prompts the data sensor data collection code to run.
2	uint8_t	print_imu	Flag that prompts the IMU values to print to the Teensy serial port at a frequency of 10hz.
3	uint8_t	print_radio	Flag that prompts the radio transceiver values to print to the Teensy serial port at a frequency of 10hz.
4	uint8_t	init_servo_radio	Flag that prompts the radio transceiver and servo initialization code to run.
5	uint8_t	init_motor_controllers	Flag that prompts the Maxon Motor controller initialization code to run.
6	uint8_t	reset_imu	Flag that prompts a resetting of the IMU. A non-zero write to this register will cause the IMU to be reset to new initial conditions.
7	uint8_t	dead_switch	Flag that prompts a dead switch to be active during operation. There will also be a dead switch in Simulink, this one is for safety/redundancy. When active, the motor controllers will only update if the radio transceiver trigger is actively held down. To turn off the dead_switch, write a zero to this register.
8	int16_t	euler_heading	IMU euler heading angle in degrees*16.
10	int16_t	euler_roll	IMU euler roll angle in degrees*16.
12	int16_t	euler_pitch	IMU euler pitch angle in degrees*16.
14	int16_t	accl_x	IMU x-direction acceleration in m/s ² *100
16	int16_t	gyr_x	IMU x-direction rotational speed in rps*100
18	int16_t	accl_y	IMU y-direction acceleration in m/s ² *100
20	int16_t	gyr_y	IMU y-direction rotational speed in rps*100
22	int16_t	accl_z	IMU z-direction acceleration in m/s ² *100
24	int16_t	gyr_z	IMU z-direction rotational speed in rps*100
26	int16_t	radio_throttle	Radio transceiver throttle value. When the user presses the radio trigger, these two registers change.
28	int16_t	radio_steering	Radio transceiver steering value. When the user turns the steering wheel, these two registers change.
30	int16_t	throttle_right_front	Right front motor controller velocity setpoint.
32	int16_t	throttle_left_front	Left front motor controller velocity setpoint.
34	int16_t	throttle_right_rear	Right rear motor controller velocity setpoint.
36	int16_t	throttle_left_rear	Left rear motor controller velocity setpoint.
38	int16_t	servo_out	Servo PWM actuation signal.

Appendix O: Project Timeline



How to run a Simulink Model on the SSIV

The SSIV was designed to be run through the use of Simulink models. A Simulink model can be run on the car in one of two ways- External mode and Deployed mode.

In the external mode usage, the Simulink model is run on the Raspberry Pi, but the laptop or computer from which the Simulink model was sent to the Raspberry Pi remains in the loop, that is, the Raspberry Pi will process the Simulink model, but the Simulink model will be updated on the laptop as well. For example, if there is a scope or display box in the Simulink model, the signal will be updated on the laptop's Simulink model as if the model were running on the laptop. External mode requires: 1) that a laptop with Simulink and the Raspberry Pi support package be used to initialize the model and 2) an Ethernet cable between the laptop and Raspberry Pi or a common WIFI network.

In deployed mode, a Simulink model is "deployed" to the Raspberry Pi and then that model is saved on the Raspberry Pi and can then later be run without the need of a laptop or internet. The initial deployment of the Simulink model also requires the Raspberry Pi support package of Simulink and an IP connection between the Raspberry Pi and the laptop.

1. Make sure that an SD card configured with the Simulink support package is inserted into the Raspberry Pi. If the SD card is not configured with the Simulink Support package, refer to Raspberry Pi Simulink support on the MathWorks website for details on how to achieve this.
2. Make sure that the laptop or computer that will be used to send the Simulink models over to the Raspberry Pi has the proper Raspberry Pi Simulink Support package. A good indication on whether this support package has properly been installed on the computer is whether the Raspberry Pi blocks appear in the block library.

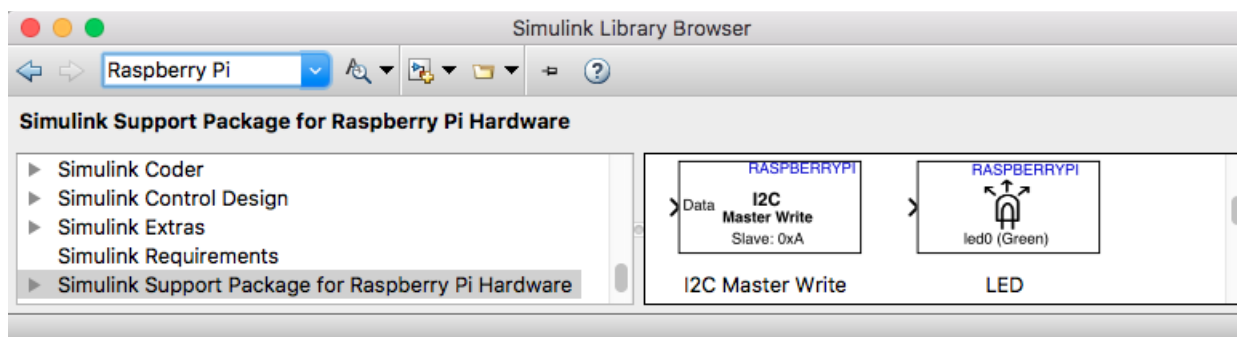


Figure 1: Library of Raspberry Pi Blocks

3. If the desire is to run a Simulink model with the capabilities to receive feedback from sensors on the SSIV and to send throttle commands to motors, the most up to date "General_SSIV_Plant.slx" model must be downloaded and used. Any controller that will be run on the car should be built around this model. The model has all the communication blocks that retrieve sensor data and send motor commands. If the desire is to simply blink

some LEDs on the Raspberry Pi or test out a computer vision algorithm using the Pi-Cam without running the motors, the General_SSIV_Plant.slx model is not required.

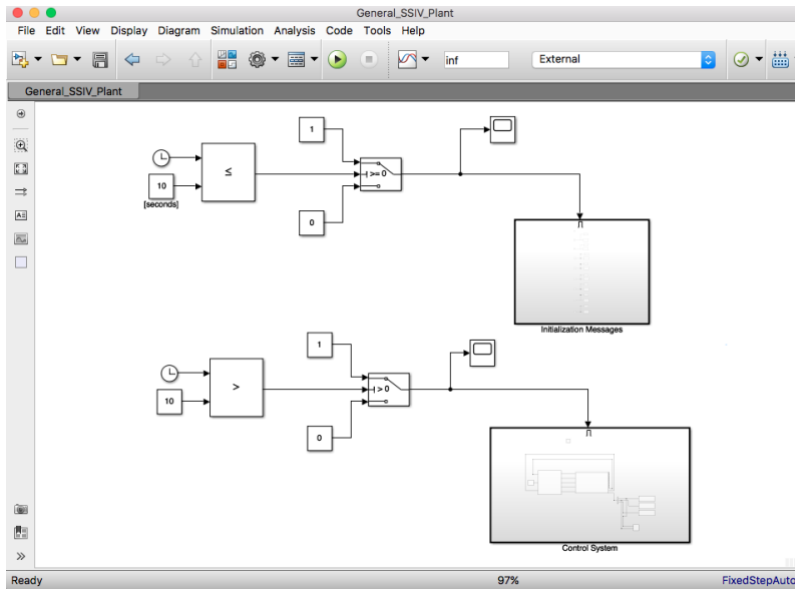


Figure 2: The General_SSIV_Plant

4. The General_SSIV_Plant.slx uses blocks that communicate to the Teensy microcontroller. Make sure that the Teensy microcontroller has been flashed with firmware that supports the latest version of the General_SSIV_Plant. In addition, make sure that the Teensy and Raspberry Pi are connected through their corresponding SPI pins.
5. Along with the “General_SSIV_Plant.slx” file, make sure that the “read_lidar” and “read_gps” .m files and “include” and “src” directories are included in the path of the Simulink model. The General_SSIV_Plant along with the gps and lidar files should all be included within a folder when downloaded. If this file hierarchy is maintained (name of the folder and simulink file can and should be changed) when the Simulink model is run, there should be no issue. The lidar and gps Matlab System blocks within the General_SSIV_Plant.slx model require the supporting GPS/Lidar files and subdirectories.

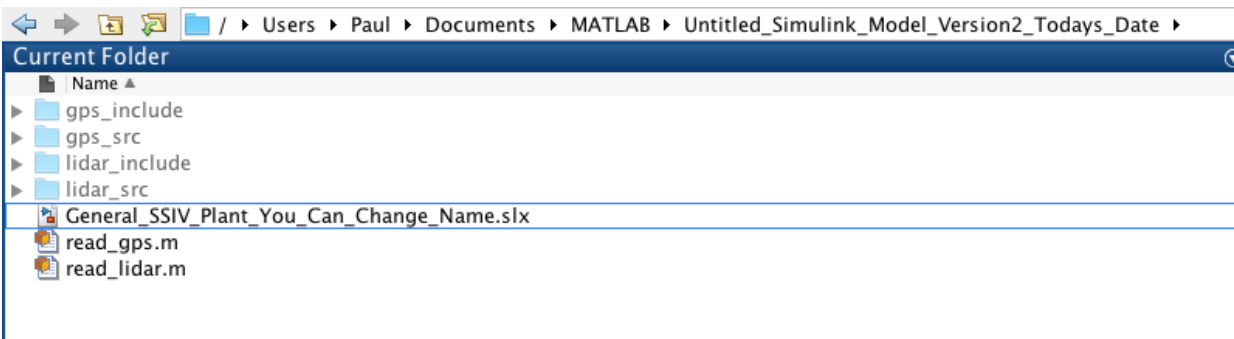


Figure 3: File hierarchy required to run Simulink model.

- Open the General_SSIV_Plant or equivalent renamed Simulink file. Navigate to the top icon bar of Simulink and change to External mode (default is usually “Normal”).

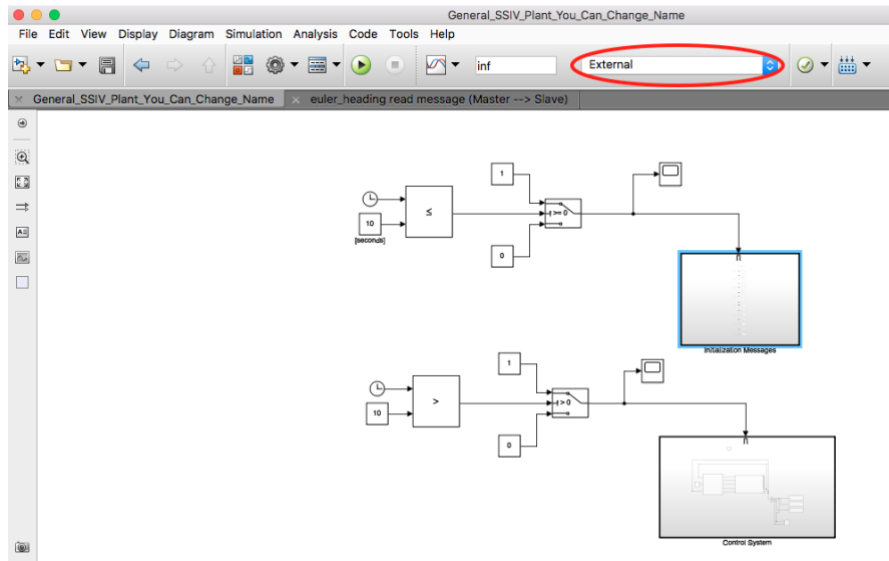


Figure 4: Select External Mode.

- Navigate to tools->run on target hardware->options. Under Hardware implementation make sure that the Raspberry Pi hardware is selected and that the username, password, and ip address under target hardware resources->board parameters are correct.

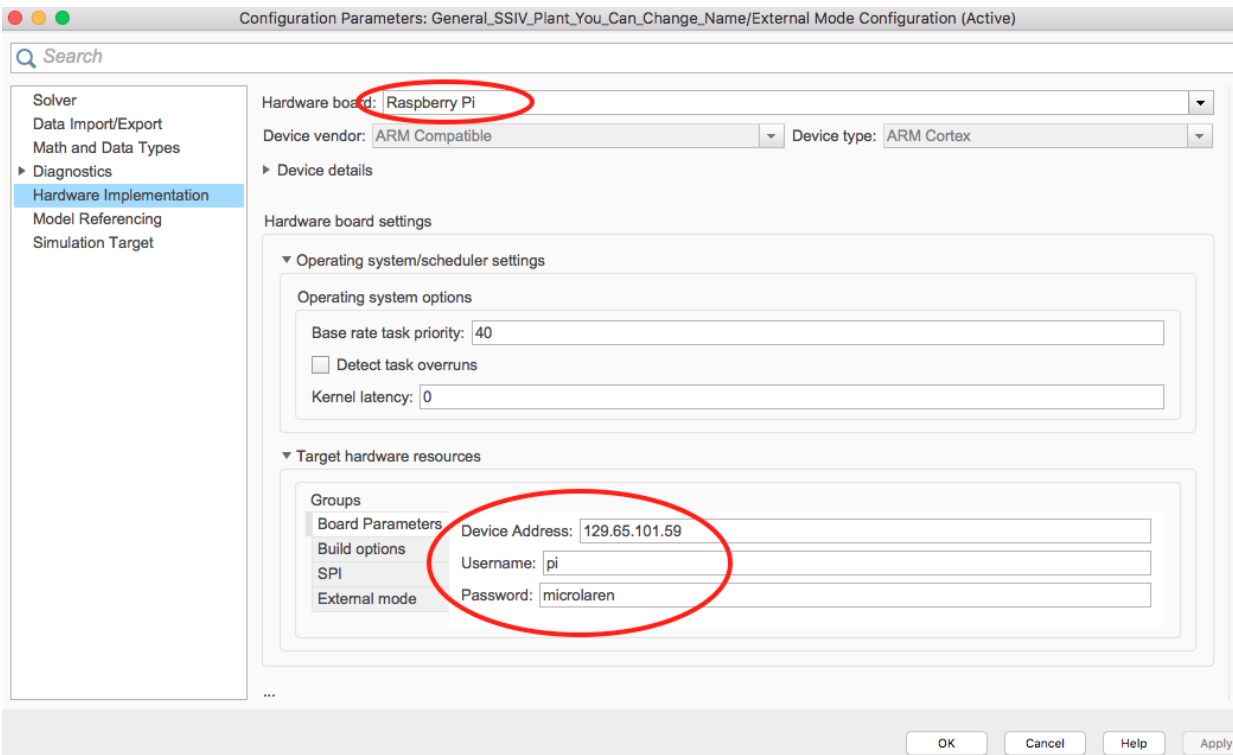


Figure 5: Confirm board parameters.

8. Next adjust the “Build Options” also under Hardware implementation. Adjust the settings to what matches the figure below.

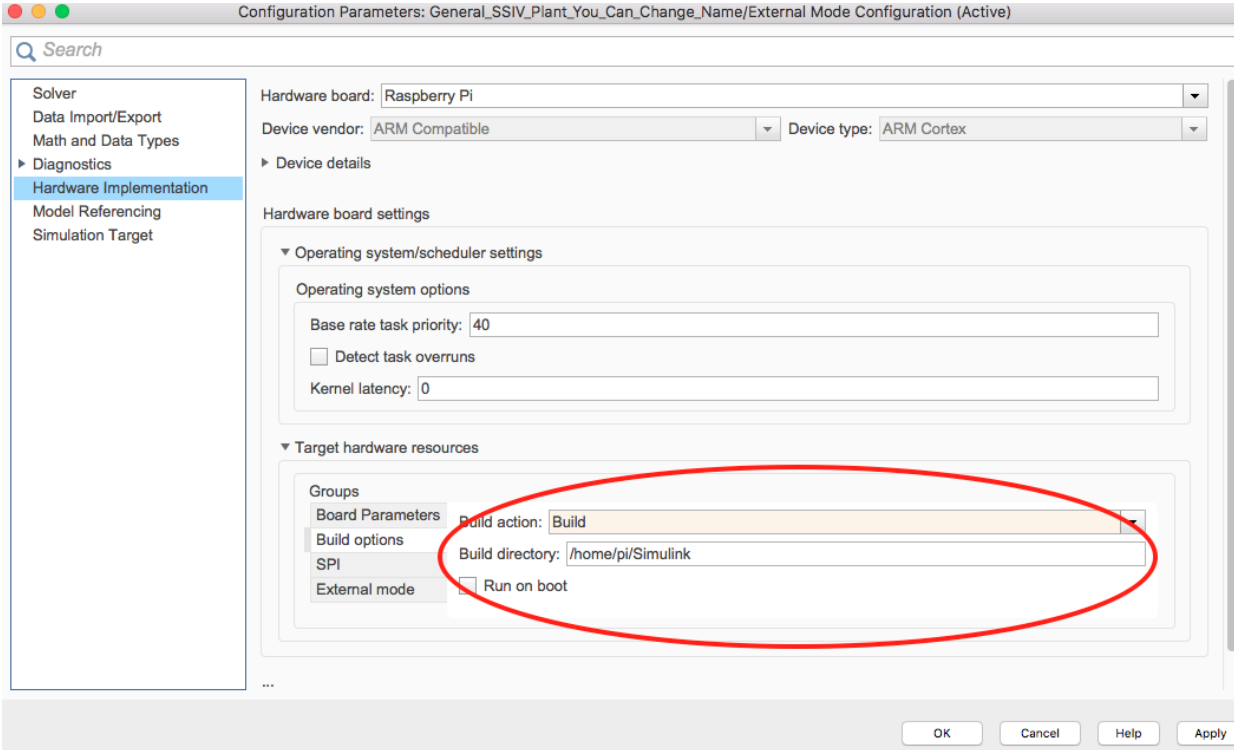


Figure 6: Confirm Build Options.

- Now go to the SPI settings and set both (currently we are only using the CE0, so technically the setting on CE1 does not matter) baud rates to 500KHz.

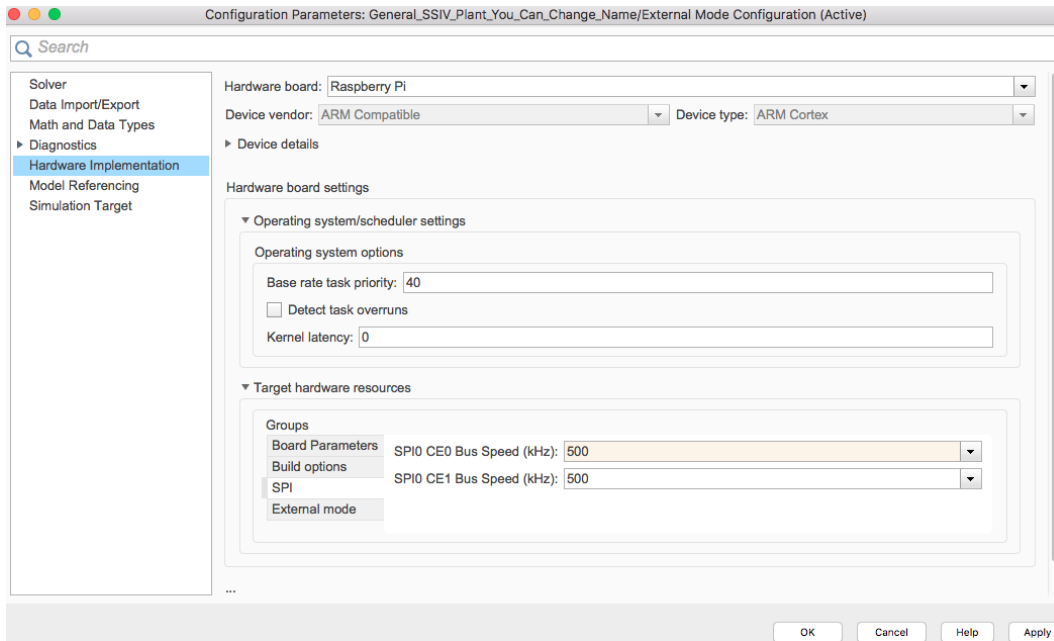


Figure 7: SPI CE0 Bus Speed.

- Under the Solver Section of Configure Parameters adjust the step size to be 0.015 roughly 67 Hz.

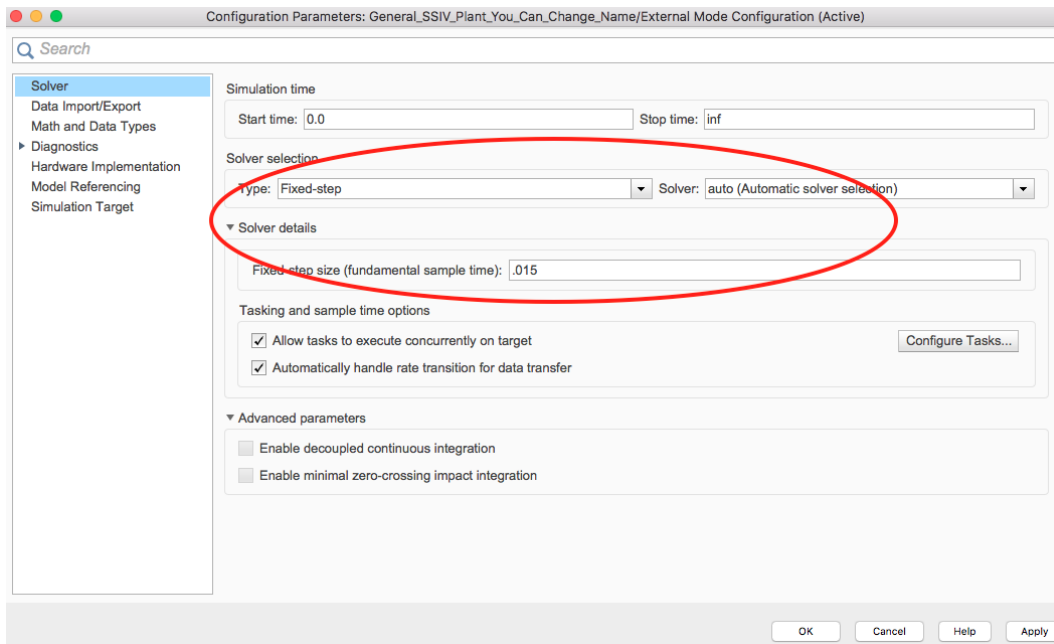


Figure 8: Adjust time step.

- Now that all the proper settings have been configured, it is time to design the controller. Go inside the “Control System” enable subsystem block and build a controller around the communication blocks by picking signals from the sensor ports and inputting motor and steering commands into the throttle and steering ports.

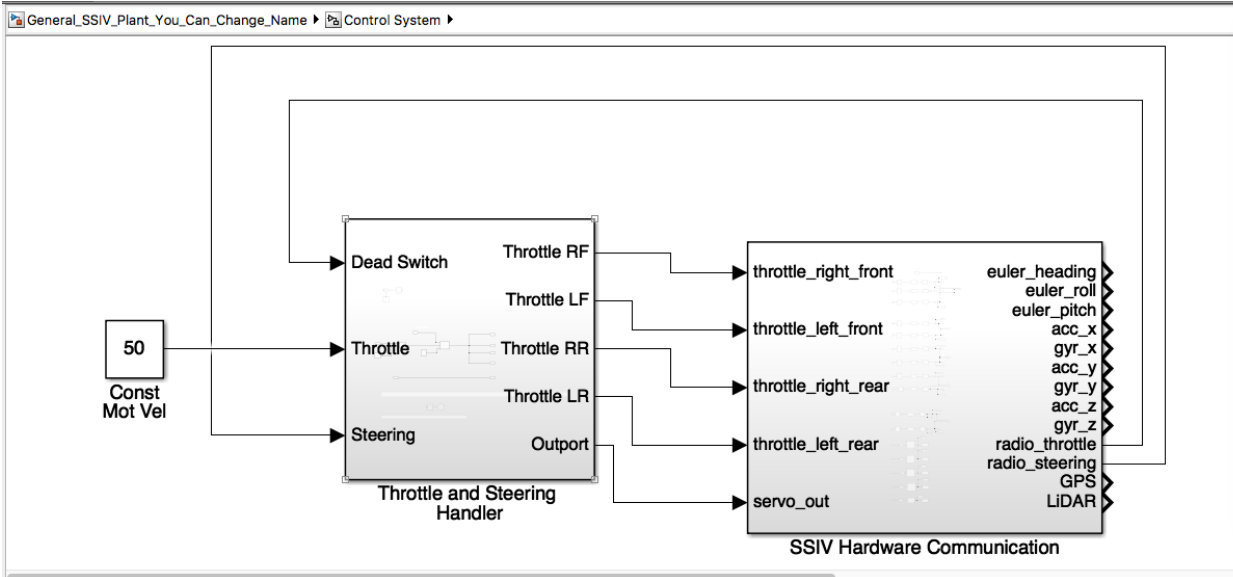


Figure 9: Building Controller around communication blocks.

- When the controller design is finished, the model can be run in external mode by pressing the green play button (confirm that step 7 was followed and mode is set to “external”).

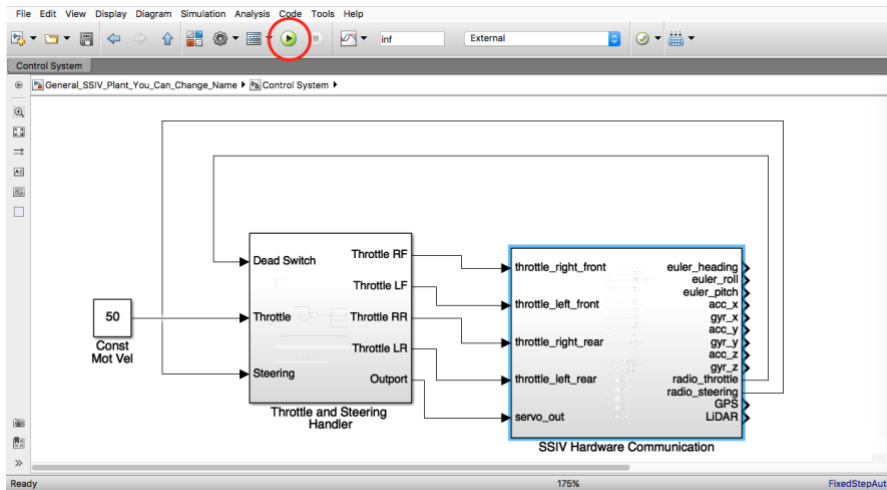


Figure 10: Running in External mode.

- The model can also be run in deployed mode by selecting “deploy to hardware” under the blue icon shown in the figure.

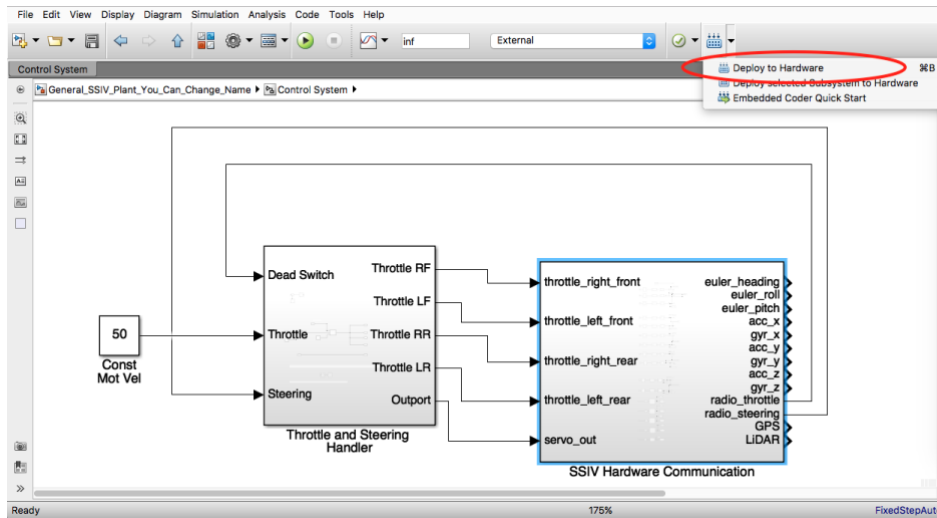


Figure 11: Deploying to hardware.

- In order to run a Simulink Model that was deployed to the Raspberry Pi, either- use the LCD UI, or connect the Raspberry Pi to a monitor, or SSH into the Raspberry Pi.
- If using the LCD screen to run a deployed model, scroll through the models using either the up or down button until the desired model is found. Then hit the Select button followed by the right button. If the model is not found make sure that step 8 was correctly followed.



Figure 12: Selecting deployed model using LCD.

16. If using the monitor or SSH to run the deployed model, navigate to the `/home/pi/Simulink` directory. Then confirm that the correct file is present in that directory. If the file is not found make sure that steps 8 and 13 were correctly followed, and no errors were reported. If the file is present, type in `"sudo ./filename.elf"` where filename is replaced by the name of the file that is to be run. Note that the Simulink executable has a `".elf"` extension. To stop running the model control-c should be pressed.

When and How to Charge the Battery

The SSIV uses a lithium polymer battery (LiPo). This type of battery is used because of its light weight. There are a number of precautions that must be taken with using a LiPo battery. LiPo batteries are usually advertised as having a cell count and a capacity. The SSIV can run on either a 6 cell battery (6S) or a 7 cell battery (7S). Each cell has a “nominal voltage” of 3.7 and a fully charged voltage of 4.2 volts. The batteries typically advertise a battery voltage alongside the cell count. This battery voltage is simply the number of cells multiplied by the nominal voltage of a cell (3.7). So, a 6 cell battery will display 22.2V (6*3.7V) somewhere around the battery packaging. The ideal voltage for the SSIV is 24V given that the motors are designed to operate at 24 volts. The capacity of the battery is given in mAh, for example 3600 mAh. Both of these values are important to note when charging the batteries.

Failure to follow proper procedure can lead to battery wear or even fire!!

Warnings:

- *Puncturing of batteries can lead to Fire!*
- *Always check battery voltage before using*
- *Do not leave batteries charging unattended*
- *Do not leave battery plugged into car when car is not in use*
- *Always use safety alarm dongle to alert user that battery charge is low*
- *Do not leave safety alarm dongle on battery when battery is not in use*
- *Do not use batteries if any battery cell is below 3.5V*
- *For long term storage, charge/discharge LiPo battery to 3.8V*
- *Store battery in proper LiPo safe pouch*

1. The SSIV can be charged one of two ways: using a battery or using the power supply. The power supply can, obviously, only be used if the car will not be driven and will just be powered on to make changes on the raspberry pi or to simulate a controller while the car is on the jack stand. If used, the power supply should be set to voltage = 24V and current = 4 A. Note how a different connection is used depending on whether the car is powered on using the power supply or the battery (figure 1).



Figure 1: Battery Powered OR Power Supply Powered

2. When first using battery, check its voltage. This is done with a device as shown in figure 2. If battery cell(s) is below 3.7 volts it is highly recommended that the battery be charged before it is used. Note that the cell voltage is different than the battery voltage, which is the sum of the cell voltages. To check the individual cell voltages using the battery monitoring device shown press the “Cell” middle button. Note how the cell voltage strip is connected to the device. The black wire of the cell voltage strip is connected to the (-) port. In the case of a cell voltage strip that contains multiple black wires- the black wire at the extreme (i.e. far right/far left depending on perspective) should be connected to the (-).



Figure 2: Checking battery cell voltages.

3. To fully charge a battery connect it to a proper LiPo charger. This involves connecting both the power cable and the cell voltage strip. Both the power cable and cell voltage strip can only be connected properly given the polar design of the cables. Connect the cell voltage strip to the port with the correct cell count. Figure 3 below shows the proper connection.



Figure 3: Connecting to charger.

4. On the charger make sure that the LiPo battery mode (figure 4) is selected- hit enter/select.



Figure 4: Display of proper battery type.

5. Select the “Balance” method of charging (figure 5). Other options of charging include “charge”. “Balance” is best because it charges the battery and then balances the cell voltages. Balancing takes longer than traditional charging, but it increases the longevity of the battery.

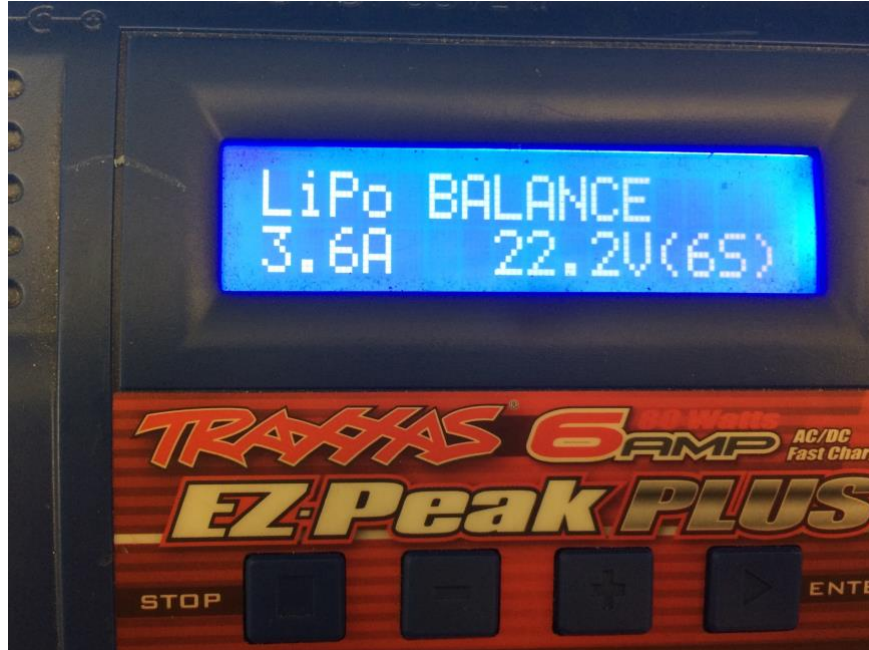


Figure 5: Select “Balance” mode.

6. Upon selecting “balance” mode, user will be prompted for adjusting the Amps and voltage (figure 6). For the current adjust using the + and – buttons so that the Amps matches the capacity of the battery. If the battery capacity is 3600 mAh, adjust the Amps to 3.6A (3600/1000). Adjust the voltage so that it matches the total battery nominal voltage (e.g. 6 cell = 22.2V). Instead of looking at the voltage reading, one can simply select the proper voltage based on the cell count. Once the proper adjustments have been made, hold down the enter/select button.



Figure 6: Adjust and select the battery charging parameters.

7. The battery will prompt the user to confirm (figure 7). Do not forget to hit enter/select else the batteries will not charge.

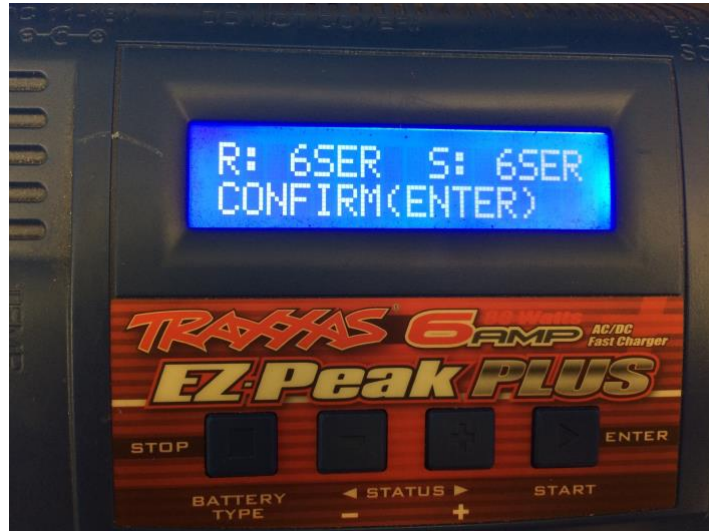


Figure 7: Don't forget to confirm.

8. Do not leave batteries charging unattended. When batteries are fully charged, the charger will beep. Unplug the battery and disconnect and store the charger.
9. When using batteries make sure to connect the battery monitoring dongle (figure 8), so that user is notified when battery charge is low. Do not use batteries past 3.5V. Ideally batteries should be recharged once they reach 3.7V. Battery run time depends on whether car is continuously being driven or is mostly in stand-by mode. It is the responsibility of the user to monitor the battery charge.



Figure 8: Battery Monitoring Dongle.

10. When finished using battery, remove the battery monitoring dongle and properly store the battery in the LiPo safe pouch (figure 9).



Figure 9: Proper storage of LiPo battery.

11. Batteries that become very puffy (big air pockets) or no longer charge to a full charge of 4.2V, it may be time to retire the battery.

How to use the LCD User Interface on the SSIV

The LCD screen is used to determine the IP address of the Raspberry Pi on the car and to run any deployed Simulink models on the Raspberry Pi.

1. The LCD screen should boot up soon after that the SSIV is powered up. A message should appear in the LCD screen as shown in the figure.



Figure 1: LCD screen upon bootup.

2. Once the LCD screen is on, scroll with the up and down buttons.



Figure 2: LCD top and bottom scroll buttons.

3. Once scrolling, the screen displays will alternate among deployed Simulink models. Additionally, there will be a screen displaying the IP address.



Figure 3: Screen displaying the IP address.

4. A Simulink Model that is displayed on the LCD screen can be run by hitting the select button.



Figure 4: Run the model by pressing the select key.

5. After pressing select, the LCD will prompt for a confirmation by pressing the right key. One the right key is pressed the model will run. If any other key besides the right key is pressed, the run request will be canceled.



Figure 5: User prompted to press right key.

6. The LCD screen will notify the user if the model is running.

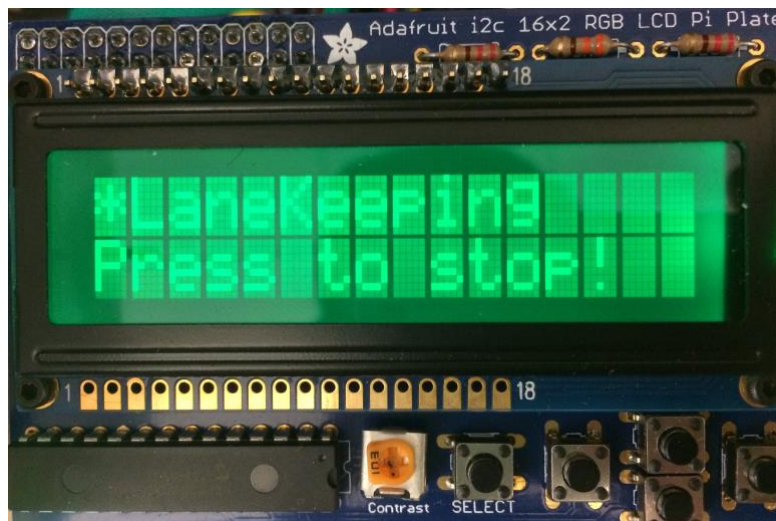


Figure 6: Model Status.

7. To cancel a model that is being run, tap any button. Once the model stops, the LCD screen will return to the default state.



Figure 7: Stop model being run.

8. To turn off LCD, go to the IP address display, tap the right button and then confirm shut down. Note LCD cannot be turned back on until the Raspberry Pi is reboot (unplug/replug battery)



Figure 8: LCD shutting down.

Understanding the Raspberry Pi Firmware on the SSIV

The Raspberry Pi firmware was designed to manage the LCD screen, communicate to the sensors, and to facilitate the running of Simulink models. This document describes the file structure of the Raspberry Pi.

1. Upon booting up the Raspberry Pi, the LCD screen and the sensor firmware are automatically run. This is achieved by running firmware file at the end of the boot up sequence as defined in the `/etc/rc.local` file.

```

BOOT_HOSTNAME=/boot/hostname
ETC_HOSTNAME=/etc/hostname
ETC_HOSTS=/etc/hosts
if [ -f $BOOT_HOSTNAME ]; then
  mv -f $BOOT_HOSTNAME $ETC_HOSTNAME
  NEW_HOSTNAME=`cat $ETC_HOSTNAME`
  echo "Setting new hostname: " > /var/log/hostname.log
  cat $ETC_HOSTNAME >> /var/log/hostname.log
  echo "127.0.0.1 localhost" > $ETC_HOSTS
  echo "::1 localhost ip6-localhost ip6-loopback" >> $ETC_HOSTS
  echo "fe00::0 ip6-localnet" >> $ETC_HOSTS
  echo "ff00::0 ip6-mcastprefix" >> $ETC_HOSTS
  echo "ff02::1 ip6-allnodes" >> $ETC_HOSTS
  echo "ff02::2 ip6-allrouters" >> $ETC_HOSTS
  echo " " >> $ETC_HOSTS
  echo "127.0.2.1 raspberrypi" >> $ETC_HOSTS
  echo "127.0.1.1 $NEW_HOSTNAME" >> $ETC_HOSTS
  echo "rebooting " >> /var/log/hostname.log
  do_expand_rootfs
  shutdown -r now
else
  # Print the IP address
  ifNames=`ip -o link show | awk -F: ' '{print $2}' | grep -v lo`
  eSpeakString=""
  _IP=$(hostname -I) || true
  if [ "$_IP" ]; then
    # Speak IP address of each interface through default audio device
    for interface in $ifNames
    do
      ifIP=`ip addr show $interface | grep "inet\b" | awk '{print $2}' | cut -d/ -f1`
      if [ "$ifIP" ]; then
        eSpeakString=$eSpeakString"My $interface IP address is $ifIP "
      fi
    done
    #Print the IP address
    printf "%s\n" "$eSpeakString"
    espeak -s 100 -p 82 --stdout "$eSpeakString" | aplay -f cd &
  fi
fi

sudo python3 /home/pi/lcd/Adafruit_Python_CharLCD/examples/SSIV_LCD.py &

exit 0

```

Figure 1: Firmware being run at the end of the boot-up sequence.

2. The LCD firmware is located in the path `/home/pi/lcd/`. The `lcd.py` file does a number of tasks including: checking and parsing the wifi status to obtain IP address, parsing and checking executable files in `/home/pi/Simulink`, checking status of the sensors, and displaying the correct messages on the LCD screen based on the buttons pressed.
3. The sensor firmware is located within the path `/home/pi/sensors/`. Within the sensors directory there are subdirectories of “gps” and “lidar”. These subdirectories, respectively, house the `gps.py` and the `lidar.py` firmware files. The `gps.py` and `lidar.py` files communicate to the respective sensors and writes the data received from the sensors to data files.

```
[pi@raspberrypi-UesXkre0HP:~ $ ls /home/pi/sensors/*
/home/pi/sensors/camera:
new

/home/pi/sensors/gps:
GPS gps_data gps.py marvelmind.py marvelmind.pyc online README.md src

/home/pi/sensors/lidar:
LIDAR lidar_data lidar.py online status
```

Figure 3: Sensor file structure.

- The gps and lidar firmware directories also have a file called status, to which the status of the sensors is written. These files are read by the LCD file to check for any errors. The files will contain information on how the error may have been triggered. These status files are continuously written to update their time stamp. The LCD file checks the time stamp to see if the sensor firmware files are online or not.

```
LiDAR not detected. Confirm GPS connection succesful
connection to /dev/tty/ACM0 ~
~
~
~
```

Figure 4: Status files.

- There are also files named gps_data and lidar_data in the gps and lidar directories. These files are where the sensor values are written to by the sensor firmware. The values that are written in those files are then read by the GPS and LiDAR simulink Matlab System blocks.

```
lidar_data      gps_data
100             -2.5
0              3.2
20             1.4
45
23             ~
26             ~
73             ~
78             ~
79             ~
79             ~
79             ~
77             ~
76             ~
```

Figure 5: Data files.

6. Should the data values need to be read outside of the context of Simulink, any python or c script can read from the `gps_data` and `lidar_data` files using standard file descriptor read commands.
7. If any sensor is added to the Raspberry Pi, it is suggested that this sensor follow a similar structure to that of the GPS and LiDAR.