STREET-SIGN AND LANE-MARKER RECOGNITION FOR THE CONTROL OF AN AUTONOMOUS GROUND VEHICLE

A Thesis

by

JAMES TERENCE MCCABE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Won-jong Kim
Committee Members,	Gregory Huff
	Pilwon Hur
Head of Department,	Andreas Polycarpou

August 2018

Major Subject: Mechanical Engineering

ABSTRACT

Autonomous vehicles and driver-assistance features have become increasingly more common over the past 5 years. With such an increase in autonomous-vehicle technology, algorithms that identify and respond to the dynamic road environment are essential. Many current navigation programs require that the travel environment be mapped numerous times before an autonomous vehicle can travel with no assistance. However, the vehicle must be able to also navigate in cases where the current path is unfamiliar. This thesis explores the use of computer-vision techniques such as color masking, Hough transformations, and bird's-eye perspective transformations for the purpose of implementing a pure pursuit navigation algorithm on a previously unknown course. The goal of this navigation program is to navigate the vehicle as closely as possible to the middle of the lane while smoothly following the path trajectory. An additional goal of this project is to implement a histogram of oriented gradients (HOG) detector for the identification of street signs and adjust the speed of the vehicle accordingly. This detector should have a near 100% success rate, and perform the detection more quickly than previously implemented object detectors.

The pure path-planning algorithm proved successful in maintaining the vehicle in the lane and proved very adapt at following the slopes of turns. In the three turns on the track, the maximum deviations from the center line were 9.14 cm (3.6 in), 2.3 cm (0.87 in), and 8 cm (3.15 in), which is very good considering the sharpness of the turns. In the straightaways, the vehicle did not perform as well, deviating 11.2 cm (4.4 in) and 15 cm (5.9 in) on the first and second straightaways, respectively. This deviation is mostly due to the vehicle shallowing the steering angle too quickly on the turn exit, leading to the vehicle not being centered in the lane heading down the straightaway. However, the vehicle was able to perform all navigation with low latency, achieving 30 frames per second (fps) compared to the 5 fps of previous lane tracking attempts. Additionally, the vehicle achieved the main objective of remaining within the lane boundaries throughout the entirety of the test. The HOG sign detector proved very successful, achieving a 100% success rate in detection of both stop sign and speed limit signs. The detections occurred in an average 0.716 and 1.07 s for the

stop sign and speed limit sign respectively, both faster than the 1.1 s of previous object detection attempts.

The results of this work demonstrate the potential application of the path-planning system as a backup to the current autonomous-vehicle navigation systems in situations where the route has not been previously mapped manually by a human driver. The success of the HOG detector shows great promise in being used in applications where the object being detected has a consistent shape, such as the numerous warning signs and route signs on highways.

DEDICATION

I would like to dedicate this work to my parents, who have given me guidance and love for every moment of my life. To my brothers who have lead the way and left big shoes to follow. To my friends who have provided the memories I will carry all my life. And to Alejandra for all of the incredible love and support you have given over the past two years.

AMDG

ACKNOWLEDGMENTS

I would like to thank Dr. Kim for all of the mentorship and guidance throughout this research project. I would also like to thank Drs. Huff and Hur for agreeing to be on my advisory committee. Finally, I want to thank all of the professors and teachers who taught me throughout my undergraduate and graduate studies at Texas A&M.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors Won-jong Kim and Pilwon Hur of the Department of Mechanical Engineering and Professor Gregory Huff of the Department of Electrical and Computer Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a College of Engineering Enhancement Fellowship from Texas A&M University and a Graduate Teaching Assistant (GAT) position through the Department of Mechanical Engineering

NOMENCLATURE

CG	Center of gravity
CPU	Central processing unit
DC	Direct current
DOF	Degrees of freedom
ESC	Electronic speed control
FPS	frames per second
HOG	Histogram of oriented gradients
HSV	Hue, Saturation, Value
IDE	Integrated development environment
IP	Internet protocol
JPEG	Joint photographic Experts Group
PC	Personal computer
PWM	Pulse width modulation
RAM	Random access memory
RC	Remote controlled
RGB	Red, Blue, Green
RX0	Receive pin 0
SVM	Support vector machine
TX0	Transmit pin 0
XML	Extensible markup language

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION AND LITERATURE REVIEW

The navigation methods currently employed by many autonomous vehicle companies such as Google, Uber, and Telsa rely upon manually driving roadways and gathering information about the roadway infrastructure with lidar before any autonomous navigation is possible. While this method can be very effective, issues can arise when the roadway undergoes changes such as during construction or road closure. Additionally, mapping all of the over 400 million miles of roadway in the United States is impractical. A navigation method which does not require the route to be mapped prior to driving could allow an autonomous vehicle to navigate virtually anywhere.

1.1 Thesis Motivation

Previous work in this unmapped navigation method is presented in [1]. In this work, a small-scale remote-controlled (RC) vehicle navigated around a single turn using only the computer-vision feedback from an Xbox Kinect. The algorithm used to control the vehicle relied on identifying all pixels in a given frame that were yellow or red, identifying which of those pixels were lane markers, and then estimating a steering point from the identified pixels. The largest drawbacks to this algorithm were the large bottlenecks that occurred from scanning the entire image for red and yellow pixels, and then identifying which of these pixels were adjacent to tile-colored pixels so that the lane lines could be isolated. As such, the time step for each frame was approximately 200 ms, or 5 fps. One of the primary motivations for this thesis is to build upon the previous method of using color for lane marker detection, but create a more efficient algorithm that would allow for a greatly improved frame rate.

A secondary motivation for this thesis is to improve the success rate of object identification while the vehicle was in motion. While the work presented in [1] attempted to identify pedestrians using open-source skeleton tracking libraries from Microsoft, the algorithm had a 40% failure rate. Because autonomous vehicles need to demonstrate almost perfect reliability in order to become commercially viable, this thesis seeks to successfully demonstrate a machine-learning algorithm for the identification of street signs with near 0% failure rate.

1.2 Design Objective

With this motivation in mind, the objective of this thesis is to design an autonomous ground vehicle capable of successfully navigating a closed course, while adjusting speed using only computer-vision. In other words, the vehicle must:

1. identify common traffic control signs and adjust the speed of the vehicle accordingly, and
2. determine lane boundaries and navigate the vehicle as close to the center of the lane as possible without prior mapping.

1.3 Autonomous Vehicles

Primitive versions of autonomous vehicles have existed since at least the 1950s [2], but the first real advances in autonomous vehicle technology occurred in the 1980s. Carnegie Mellon developed an autonomous vehicle in a neural network that forms the foundation many modern control methods are built upon [3]. In the development since the 1980s, a stratification of autonomous vehicle capabilities has emerged. While currently almost all commercial passenger vehicles require driver intervention 100% of the time, more car manufacturers and technology firms are developing vehicles that have increased autonomy.

1.3.1 Levels of Autonomy

As defined by the Society of Automotive Engineers in [4], there are six levels of vehicle automation ranging from "No Automation" to "Full Automation". The first three levels are defined by human drivers monitoring the driving environment. Level-0 means that the driver controls steering and speed completely, even if they might be receiving feedback from the car. An example would be the lane departure warnings or backup camera sensors seen in many currently produced vehicles. Level-1 is defined by the vehicle having control of either steering or vehicle speed, but not both. Cars that have adaptive cruise control while the driver continues steering or parking assistance while the driver controls speed are examples of level-1 vehicles. Level-2 vehicles take care of both driving and steering tasks, but the driver must still have hands on the wheel and feet

on the pedals to be ready at any moment should the system fail.

In level-3 vehicles and above, the driver can begin to have some level of inattention to the roadway safely. Level-3 vehicles are programmed to be able to respond to immediate threats like collisions and obstacles, but still might request driver intervention within a specified time period. Therefore, drivers are still expected to remain awake in a level-3 vehicle. At level-4, the vehicle can respond to any hazard completely autonomously. It might request that a human driver take control, but even if the driver does not intervene, the vehicle can guide itself to safety. Level-5 vehicles have no human intervention whatsoever. An example would be an automated taxi or delivery vehicle between two known points so that no driver is necessary. In order to successfully navigate the dynamic environment the roadway presents, current autonomous vehicles must be capable of handling a number of tasks outlined below.

1.3.2 Lane Keeping

According to [5], 22% of vehicles ran off the edge of a roadway, and 11% of vehicles failed to stay in the proper lane in 5,471 crashes studied from 2005 to 2007. Many current vehicles are equipped with lane departure warning systems which can alert the driver whenever they have drifted out of their lane [6], [7]. In the case that the driver begins to unexpectedly drift from the lane, such systems will respond with a loud beep and a vibration of the driver seat or steering wheel to warn the driver. The systems appear to be helping prevent accidents. From [8], lane departure warning lowers the rates of single-vehicle, sideswipe and head-on crashes by 11%, and the rate of injury that the crashes cause by 21%. The reduction in accidents from these level-0 driver-assistance technologies offers the hope that future autonomous vehicles with even greater control over the steering and navigation can reduce these accidents even further.

1.3.3 Speed Control

Some vehicles on the road currently posses the level-1 technology of adaptive cruise control. Like typical cruise control, this technology allows for a vehicle to maintain a speed as set by the driver, but is unique in that the vehicle will adjust the speed as necessary by traffic conditions

[9]. These systems employ a radar or lidar to continually search for upcoming vehicles. When a vehicle is detected, the car will slow and match the speed of the leading vehicle to maintain a safe driving distance. If the path is cleared, the vehicle will then accelerate back to the set speed. Another example of speed control in autonomous vehicles is automatic braking for collision avoidance [10]. When the autonomous vehicle detects an oncoming vehicle from the side or a fast approaching vehicle from the front, the brakes are immediately applied to reduce the severity of the accident or avoid the accident entirely.

1.3.4 Object Recognition

While lane keeping and speed control can be accomplished by level-0 and level-1 vehicles that are currently on the road commercially, level-2 and above vehicles require an even greater awareness of the surrounding environment to function with no driver intervention at all. At this point, object recognition becomes necessary. Of particular importance, vehicles must be able to detect and avoid pedestrians in urban environments [11]. Additionally, vehicles must be able to recognize traffic control signals such as stop signs, speed limit signs, yield signs, and traffic lights to properly follow traffic laws [12].

1.4 Machine-Learning and Computer-Vision Algorithms

To perform the necessary driving functions, an autonomous vehicle must have a well-trained computer to coordinate the numerous sensors and cameras and respond immediately to potential obstacles and collisions. Machine-learning and computer-vision algorithms like those discussed below will be necessary to teach the computer how to react to these dynamic environments.

1.4.1 Support Vector Machines

For this thesis, the machine-learning algorithm utilized is a support vector machine (SVM). SVMs are a type of supervised learning, meaning that training data are first labeled and classified by a human before they are given to the computer. Supervised-learning algorithms then use those labeled data as the basis for deciding whether subsequent unlabeled data belong to one class or another.

SVMs work by creating a hyperplane, or decision boundary, among the sets of labeled training data supplied by the user. The first priority in selecting the hyperplane is that all of the data from one class falls on the same side of the decision boundary, as shown in Figure 1.1. The second priority is to maximize the distance from the closest training data points on either side of the hyperplane, as seen in Figure 1.2 [13]. The measure of how far the hyperplane lies from the closest training data is known as the margin.

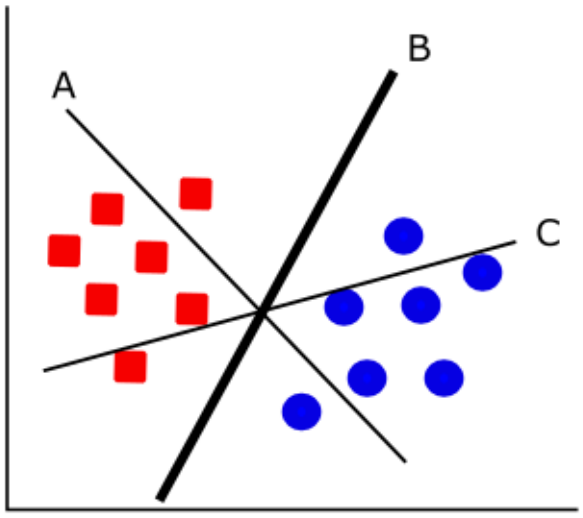


Figure 1.1: Hyperplane B correctly segregates the data of each type into two distinct classes, adapted from [13]

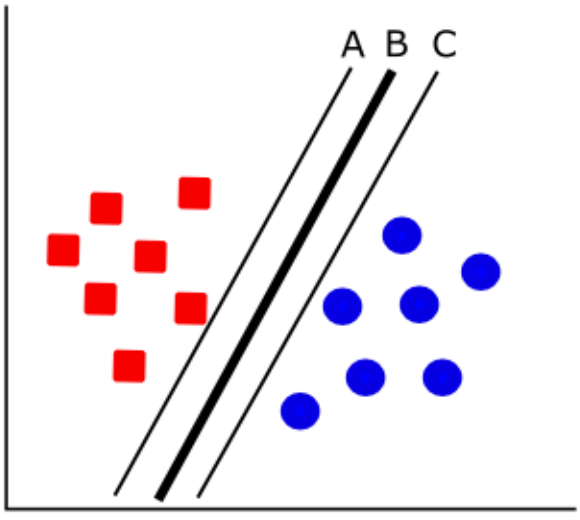


Figure 1.2: Hyperplane B maximizes the margin between the two classes of data, adapted from [13]

SVMs are commonly used in computer-vision applications for object detection. In this case, the two classes would be comprised of positive training data from images with the object, and negative training data from images without the object. Once the hyperplane has been created from this training data, subsequent images can be identified as containing or not containing the particular object based upon which side of the hyperplane the new image falls.

1.4.2 Histogram of Oriented Gradients

In order for the SVM to create a hyperplane, the images of the object to be detected must be transformed into a numerical form that the algorithm can understand. This transformation results in what is known as a feature vector, and the algorithms that perform these transformations are called feature descriptors. The feature descriptor used in this thesis is the histogram of oriented gradients (HOG). HOG has already been utilized in autonomous vehicle applications for the identification of pedestrians [14], bicyclists [15], and other vehicles [16].

The use of HOG as a feature descriptor became widespread after its usage in [14] for the detection of pedestrians. As the name implies, the image features described by HOG are the directionality of image gradients and how these gradient directions are distributed. The first step in finding the HOG descriptor is to calculate these gradients. Dalal and Triggs [14] use several kernels for calculating the image gradient, but found most success with 1-D centered point derivatives of the form $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ for the horizontal gradient g_x and $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}^T$ for the vertical gradient g_x. This means that the gradient of a pixel in the image is calculated by comparing the darkness of the pixels on the left and right to one another, and then repeating for the top and bottom pixels, as shown in Figure 1.3. In this example, the pixels on the bottom-right are darker than the pixels in the top-left. Using the gradient calculation above, the horizontal and vertical gradients are given by

$$\nabla f(x,y) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} f(x+1,y) - f(x-1,y) \\ f(x,y+1) - f(x,y-1) \end{bmatrix} \tag{1.1}$$

For the given example, both the horizontal and vertical gradient are then calculated as

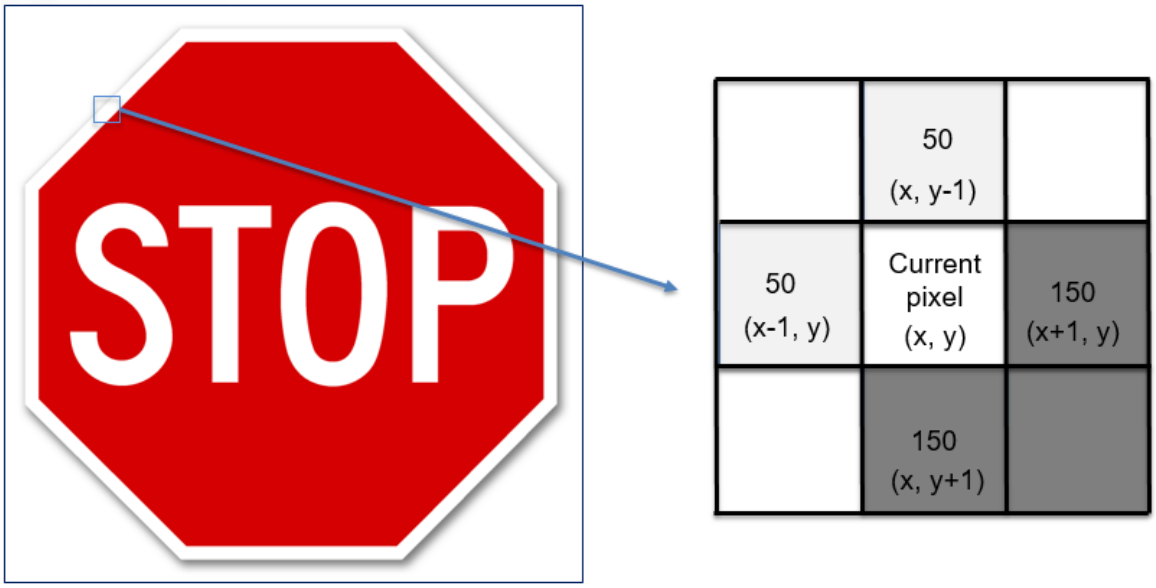


Figure 1.3: The darkness of a selected pixel is compared to adjacent pixels for gradient calculation

$$\begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} f(x+1,y) - f(x-1,y) \\ f(x,y+1) - f(x,y-1) \end{bmatrix} = \begin{bmatrix} 100 \\ 100 \end{bmatrix} \tag{1.2}$$

The gradient magnitude and orientation can then be found by:

$$g = \sqrt{g_x^2 + g_y^2} = \sqrt{100^2 + 100^2} = 100\sqrt{2} \tag{1.3}$$

$$\theta = \tan^{-1}(\frac{g_y}{g_x}) = \tan^{-1}(\frac{100}{100}) = 45^\circ \tag{1.4}$$

This indicates that the gradient from light to dark is oriented down and to the right at a 45° angle from the pixel being examined. The angle of θ is typically made to be unsigned and limited to between 0° and 180° for simple objects, although [14] notes that signed orientations may be necessary for more complicated object recognition tasks. The gradient calculation is then performed for every pixel in the image.

With the image gradient calculated, the image is divided into boxes of pixels called cells. The number of pixels per cell can be increased or decreased depending on how fine the details of the image are, but for this example the cell size will be assumed to be 8×8 pixels. The trade-off of

capturing more detail in the image is the increased computation time needed to process the image. Within these cells, the gradients are sorted into one of 9 bins corresponding to the angles 0, 20, 40,..., 160°. If the gradient orientation falls exactly on one of the bins, for example at 120°, the full value of the gradient magnitude would be added to that bin. If the gradient orientation falls between bins, the value of the gradient magnitude is split proportionally between the nearest bins. For example, at 90° the value of the gradient would be split equally between the 80° and 100° bins. An example of splitting the gradient orientations into a histogram is shown below in Figure 1.4.

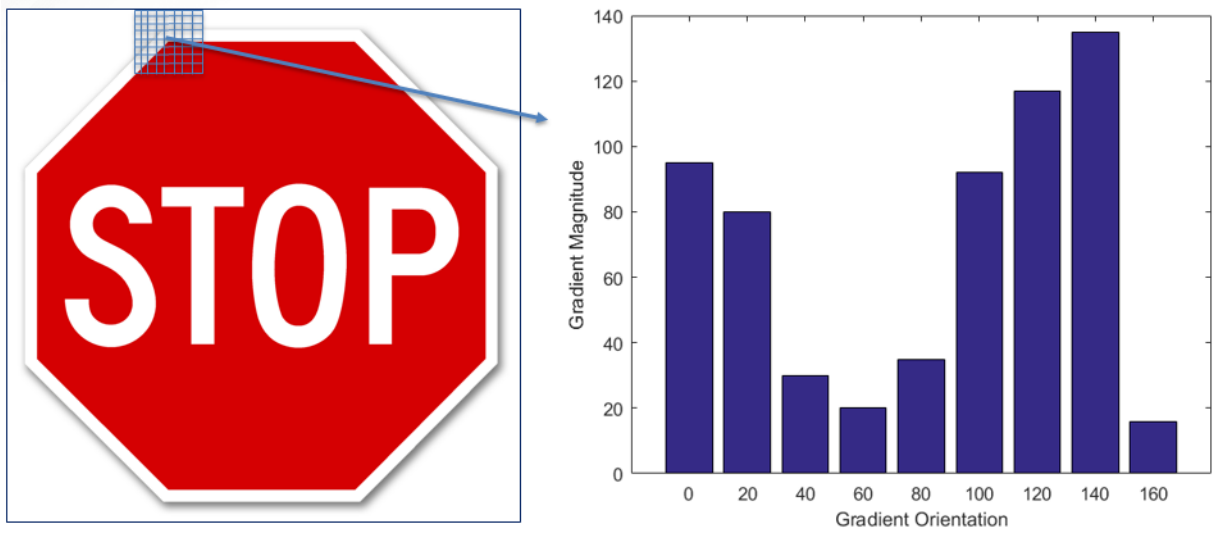


Figure 1.4: Gradient magnitudes of a cell are distributed by gradient orientation

Once histograms have been calculated for all 8×8 pixel cells, the histograms must be normalized to account for differences in lighting. Otherwise, histograms from particularly light or dark areas of the image would have gradients with relatively high or low magnitudes. To normalize the histograms, the 8×8 pixel cells are grouped into larger blocks of 16×16 pixels. Since there are 4 cells of 8×8 pixels in each 16×16 pixel block, and each cell contains 9 values representing the gradient magnitudes at each bin orientation, the entire block can be represented by a 36×1 vector. The values within the vector are normalized by dividing each element by the L2 norm of the entire vector. The L2 norm has the form

$$|x| = \sqrt{x_1^2 + x_2^2 + ...x_n^2} \tag{1.5}$$

where $x_1, x_2, ..., x_n$ are the elements of the vector being normalized. Once the vector has been normalized, the block is shifted over by 8 pixels so that there is overlap with the previous block. The normalization is conducted again, and the cycle of shifting and normalizing is repeated until the end of the row. The block then shifts downwards 8 pixels and the process repeats until the entire image has been normalized. This process is known as sliding-window normalization, and is shown in Figure 1.5.

Figure 1.5: Sliding window normalization of adjacent 16×16 blocks

When the 8×8 pixel blocks from the original image are replaced with the strongest normalized oriented gradients from that particular box, the resulting image shows the general contour of the object being studied. An example of the normalized gradient field image is shown in Figure 1.6. With all blocks normalized, the final feature vector which represents all information about the image is created by concatenating the 36×1 vectors from each block. If the size of the image being analyzed was 64×64 pixels, there would be 7 blocks in the horizontal direction and 7 in the vertical direction. With 36 elements in each block, the total vector representing the entire image would be 1,764 elements long for an input image vector of size 12,288 (64×64×3). This 1,764 element feature vector is used by the SVM for deciding whether subsequent images passed to the program have sufficiently similar feature vectors to be considered the same object.

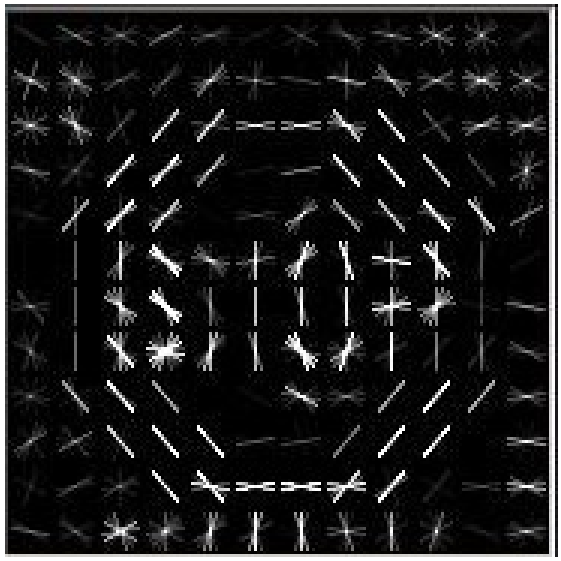


Figure 1.6: Histogram of oriented gradients vector field output for a stop sign

1.4.3 Color Modeling

In many computer imaging tasks, a user may wish to detect a certain colored object. From a given input image, the computer will examine each pixel and determine whether the pixel color properties fall within certain thresholds. The rejection or retention of pixels based on these color properties is known as color masking. For example, a red color mask of a stop sign would appear as shown in Figure 1.7.

Figure 1.7: Color masking performed on a stop sign, with white pixels indicating retained pixels, and all other pixels rejected

In order to perform color masking, there must be a method to determine which pixels from an image qualify as falling within the threshold of the color being sought. A common method of classifying pixel colors is using the red-green-blue (RGB) model, wherein a pixel is classified by its combination of red, green, and blue light colors. As can been seen in Figure 1.8, colors like magenta, cyan, and yellow can be created from combinations of the primary three colors. The limitation with this model is that it is not robust to changes in lighting conditions, so the same object will have drastically different red, green, and blue values in shadow and bright light.

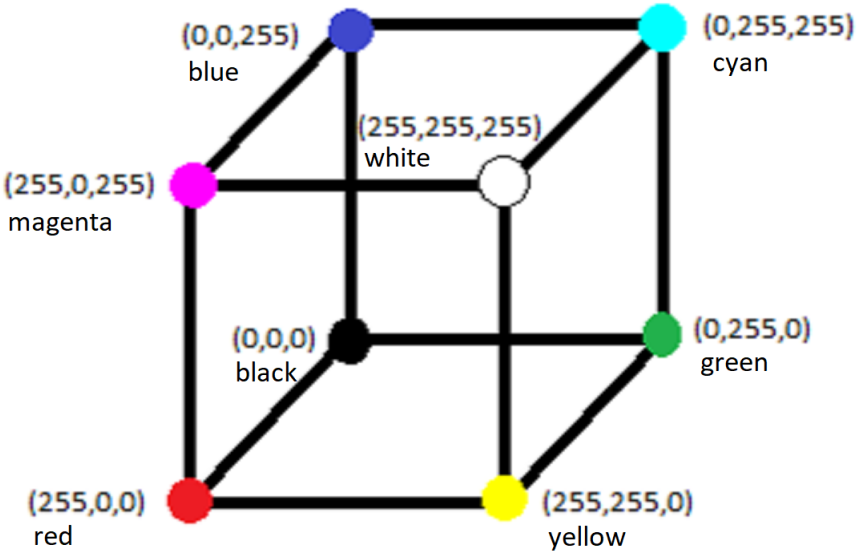


Figure 1.8: Cubic representation of the RGB color model showing alteration of color as red, green and blue values are varied, adapted from [17]

An alternative method used for this thesis is the Hue/Saturation/Value (HSV) color model. A visual representation of this color model is shown below in Figure 1.9. With this model the pure color is entirely determined by the hue value, with red usually considered to be the low end of the scale. Saturation determines how much white is present in the color, or how vibrant the color appears. Low saturation means that the color will appear faint, whereas full saturation gives the pure, unaltered color. Value determines how dark the color appears. Low value will make the color seem black, whereas high value will again give the pure color. Therefore, the only difference between a given color in bright light and shadow will be slight differences in saturation and value, but the underlying hue will remain the same. By setting appropriate thresholds for the acceptable

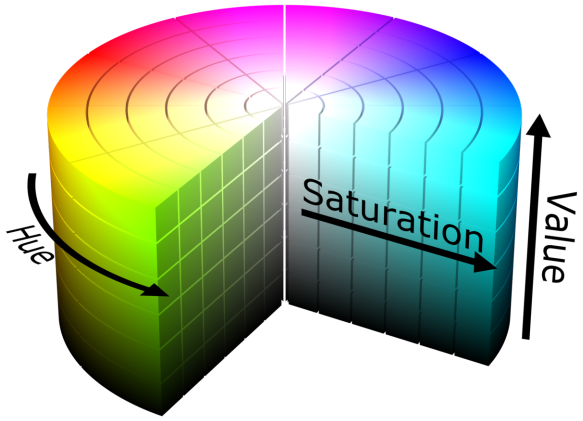


Figure 1.9: Cylindrical representation of the HSV color model demonstrating variance in color as hue, saturation, and value are modified [18]

range in saturation and value for a given color, it becomes easy for a computer to identify that color even as the lighting conditions change.

1.4.4 Hough Lines

An important task in many image recognition problems is the identification of object edges. However, the computer does not automatically recognize that a group of collinear pixels belong to the same line. The method employed by this and many other papers is to use the Hough transform to detect and group these pixels into line segments. The process of grouping these independent points into lines is described in [19]. To show how a Hough transformation works, suppose a line lies on an *xy* plane as shown below in Figure 1.10. If a normal is drawn from the line through the origin, the line can be described by the length of the normal, ρ, and the angle of the normal from the *x*-axis, θ, by

$$x\cos\theta + y\sin\theta = \rho \tag{1.6}$$

If θ is restricted to be between 0 and π, the parameters ρ and θ uniquely describe the line. Therefore, a line in the *xy* plane can be completely described by a single point in the θ-ρ domain. Along similar lines, suppose that instead of being given a line, *i* points (x_i, y_i) are given in the *xy* plane as in Figure 1.11.

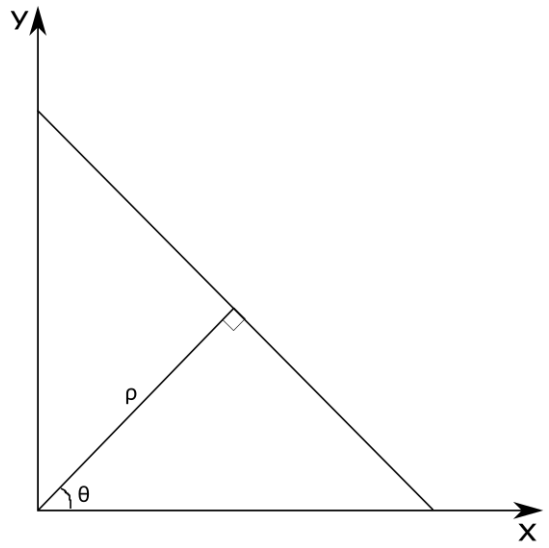


Figure 1.10: Representation of a line on the *xy* plane represented by a distance ρ and angle θ, adapted from [19]

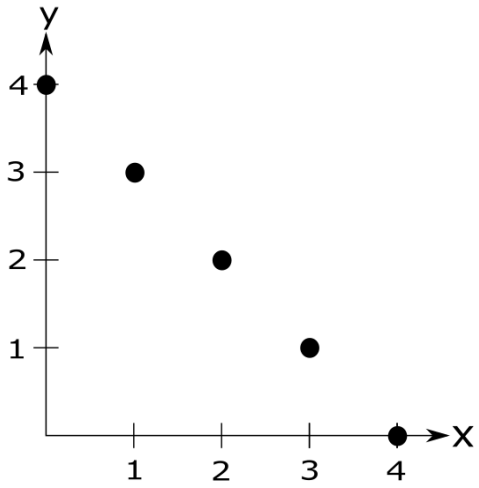


Figure 1.11: A collection of points lying on the plane *xy*, adapted from [19]

The points can once again be represented by (1.4). Substituting the individual points in for x and y results in the equations $4\sin\theta$, $\cos\theta + 3\sin\theta$, $2\cos\theta + 2\sin\theta$, $3\cos\theta + \sin\theta$ and $4\cos\theta$. Plotting these functions as a function of θ from 0 to π results in the curves seen in Figure 1.12.

It is clear that all of the sinusoidal curves intersect at the point $(\frac{\pi}{4}, 2\sqrt{2})$. As was shown previously, since a line in the *xy* plane can be uniquely represented by a single point in the plane $\theta - \rho$, all of the points that were given in the original *xy* domain must belong to the same line. Thus,

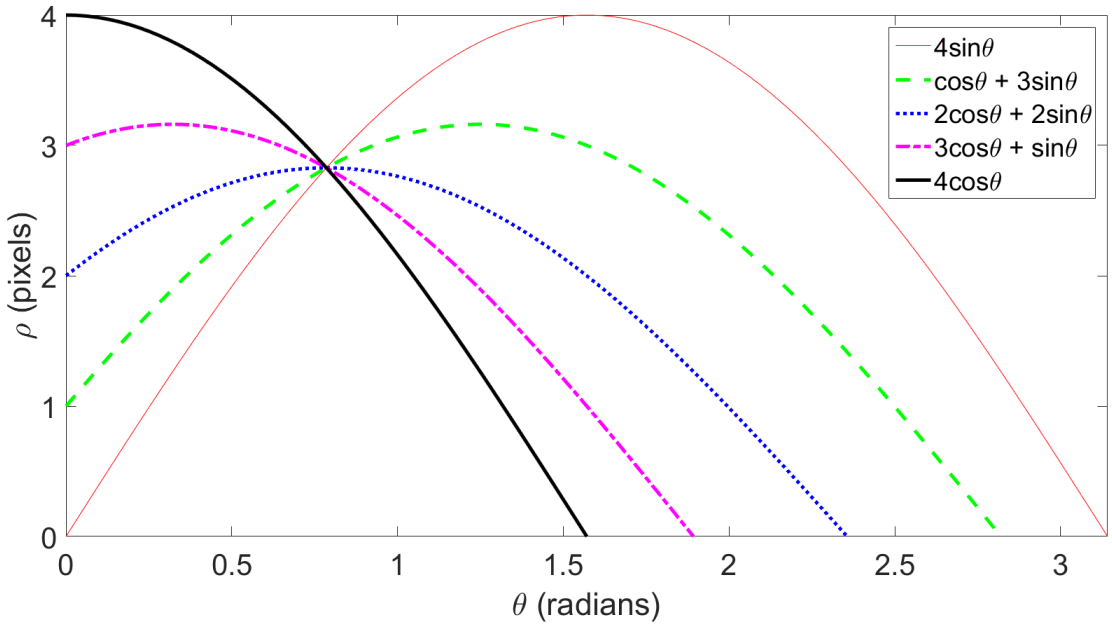


Figure 1.12: Sinusoidal curves from resulting from the substitution of points (x_i, y_i) into equation 1.4 and plotting as a function of θ

the Hough transformation has successfully grouped these points, which would represent pixels in an actual image, into a line. Repeating this over every pixel in an image results in hundreds of (θ, ρ) intersection points representing every line segment in the entire image. In order to filter out any extraneous line segments, the Hough transformation algorithm will only consider a (θ, ρ) pair to constitute a line segment if it has received enough "votes", meaning enough intersections have occurred at that particular (θ, ρ) point. Additionally, since not every line detected will be perfectly straight, a pixel which is part of a line might not end up voting for the same (θ, ρ) as the other points on the line. To make up for this, the Hough transformation algorithm will allow points within a certain threshold of a (θ, ρ) to be considered part of the same line.

1.5 Thesis Contributions

This thesis presents an improved pure-pursuit lane navigation method that does not require pre-mapped navigation points that are typical of other pure-pursuit controllers. Through the use of more efficient computer-vision algorithms such as color masking and the Hough transformation, the vehicle responsiveness to the changing curvature of the track is dramatically increased from 5 to

25 navigational reference points per second as compared to [1]. Additionally, the use of machine-learning algorithms for the identification of street signs shows 100% accuracy in identification, marking a vast improvement over the 60% accuracy of the pedestrian identification methods presented in [1]. Additionally, the detection time was improved from an average of 1.1 s to an average of 0.72 s for stop sign recognition, while remaining the same for speed limit signs.

1.6 Thesis Overview

The subsequent sections of this thesis explain the methodology used in the thesis experimentation. Chapter 2 presents the hardware and software components used for the creation of the autonomous vehicle. Chapter 3 explains the vehicle system dynamics and the development of a modified pure-pursuit controller and speed state controller. Chapter 4 describes important parts of the codes that were used to make the vehicle operate and how the key thesis objectives were accomplished. Chapter 5 introduces the experimental environment and setup and then presents the experimental results and important findings. Chapter 6 summarizes the conclusions from the experiment and future work related to the experiment.

2. SYSTEM HARDWARE AND SOFTWARE

Due to the project inspiration from [1], the autonomous vehicle is intended to only utilize low-cost, commercially available hardware and open-source software to implement the machine-learning algorithms previously discussed. The hardware and software used for this autonomous vehicle are presented in the following sections.

2.1 Hardware Components

2.1.1 Autonomous Vehicle

The autonomous vehicle selected for the experimentation is a Traxxas Slash RC Car chassis with 4-wheel independent suspension. The vehicle is shown below in Figure 2.1, with the exterior covering removed for easier access to the electronic components.

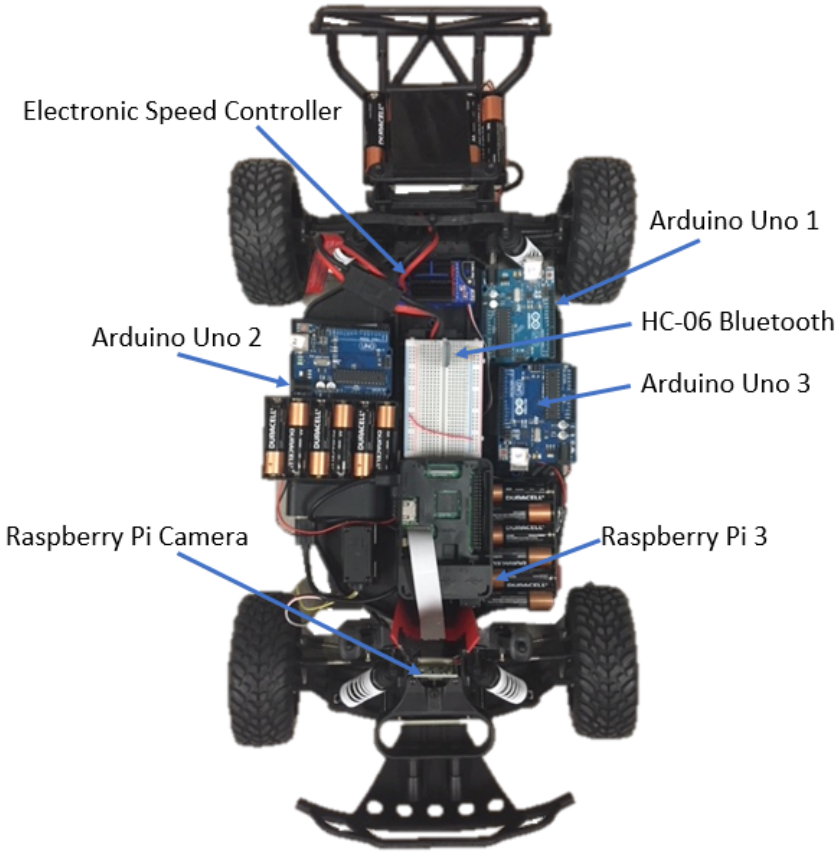


Figure 2.1: Traxxas Slash Remote Controlled Car chassis used for autonomous testing

The wheel base measures 31.75 cm from the front axle to the rear axle, and the wheels sit 22.9 cm apart. The vehicle is rear-wheel driven by a Titan 12-turn 550 direct current (DC) motor, and a differential connects the rear wheels. The motor is driven by a 8.4-V, 3000-mAh battery. The original servomotor which controls the steering wheels at the front of the vehicle is replaced by a Parallax continuous-rotation servo with digital feedback to provide the servomotor's current steering angle. The car's speed is controlled the XL5 Electronic Speed Controller (ESC). The radio receiver of the vehicle has been modified so that the car can be run autonomously without requiring an RC transmitter.

2.1.2 Arduino Uno

Three Arduino Uno boards are chosen for integrating and controlling the electrical components of the vehicle due to the large number of input and output pins, the serial communication ports and the ability to send pulse width modulation (PWM) signals. The Uno's small footprint makes it easy to fit into the little space available on the vehicle's chassis, and the low power requirements allow for the Unos to be powered by three onboard 9-V battery packs without sacrificing power to the motors. Additionally, the integrated development environment (IDE) for the Uno bares strong similarity to the C language conventions and is easy to understand. For this project, one Uno is needed to send the steering inputs to the vehicle's servomotor. The second Uno receives the Bluetooth signal from the PC and passes the steering commands to the first Uno. The third Uno sends the PWM signal that modifies the speed of the onboard ESC based on detected signs from the PC. An example of a typical Arduino Uno board is shown in Figure 2.2.

2.1.3 Raspberry Pi 3 and Arducam

While the Uno is effective for providing digital input and output signals, it does not have the computational power capable of handling the computer-vision elements of the project. For this purpose, a Raspberry Pi 3 is employed. The Raspberry Pi, shown in Figure 2.3, is essentially a small computer, with a 4-core ARM Cortex-A53, 1.2GHz central processing unit (CPU). It also has 1 GB of random-access memory (RAM), so it can handle a fairly heavy load for such a small

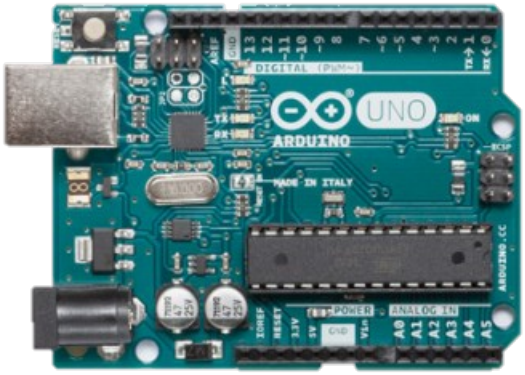


Figure 2.2: Arduino Uno board used for digital input and output pins [20]

device. The Raspberry Pi runs on a Debian platform, and comes pre-installed with a Python compiler. This board is useful for this particular project because all of the computer vision code was written in Python. The board is powered by a 5-V, 10,000-mAh portable battery. The camera used for the autonomous vehicle is an Arducam 5 Megapixel 1080p camera which is made to interface with the Raspberry Pi. The camera is capable of recording both video and still-frame pictures.

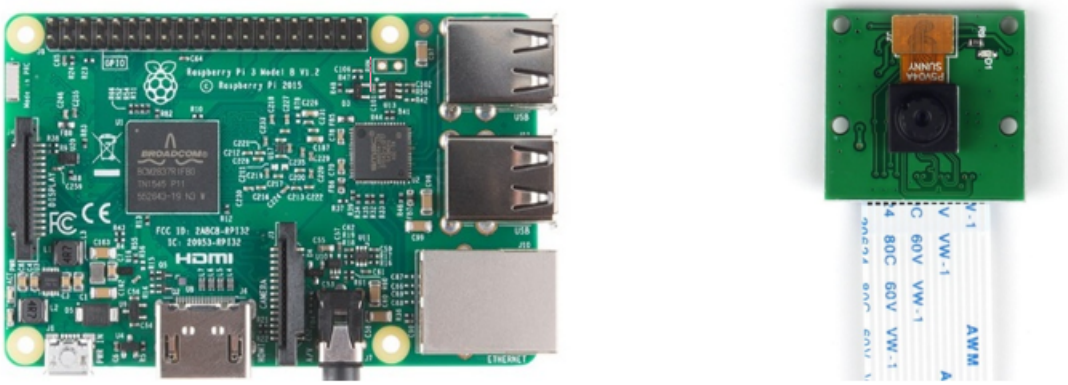


Figure 2.3: Raspberry Pi 3 Model B (left) and Arducam 5 Megapixel Camera (right) [21],[22]

2.1.4 HC-06 Bluetooth Module

Bluetooth communication is necessary for this project in order to pass navigation information to the Uno once it has been calculated on the computer. The module chosen for this function is the HC-06 Bluetooth module shown in Figure 2.4. This module is a slave module, meaning that it can only accept connection from another device, but never initiate connection. The module has 4 pins, two for the 5-V input voltage and ground, and two for the transmission and receiving of the serial data.

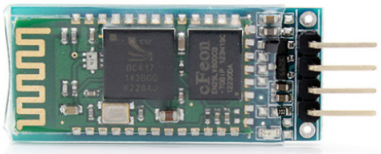


Figure 2.4: HC-06 Bluetooth Module [23]

Before the Bluetooth module can be used for communication, it must be properly paired with the computer. The following method of pairing is used for pairing the HC-06 Bluetooth module with a PC running Linux Ubuntu 16.04. After enabling Bluetooth on the Ubuntu platform, the HC-06 module appears as a possible connection device. The device is initially paired with the computer by selecting the HC-06, and entering the password, which is usually 1234. The device should connect with the PC for a few seconds, and then disconnect. From the terminal in Ubuntu, the command `hcitool scan` is entered into the terminal. The terminal responds with the device address, which should take the from of `00:21:13:01:0B:05`. A configuration file is created using the terminal command `sudo nano /etc/bluetooth/rfcomm.conf`, and the following is typed into the file.

```
rfcomm0{
bind no;
device 00:21:13:01:0B:05;
channel 1;
comment "HC-06"
}
```

This prevents the Bluetooth module from binding to the computer when the computer starts

up, which can cause undesirable communication behavior. The device number typed into the file should be the same as the number received from the `hcitool scan` command, and channel 1 tells the computer from which communication channel to begin broadcasting and listening. When ready to bind the computer to the module, the command `sudo rfcomm bind 0 00:21:13:01:0B:05 1` is typed into the terminal window. This binds the computer and module together for as long as the computer is powered. If the computer is shut off, this command must be retyped in order to reestablish connection.

2.1.5 Hardware Connection

This section details how the hardware components of the vehicle are connected. These connections assure that all components are adequately powered, and that all signals are properly passed from hardware to hardware with no loss of information. The connections of the Uno which controls the Bluetooth will be discussed first, followed by the Uno controlling the steering servo, and then the speed control Uno. To avoid confusion, these Unos will be referred to as Uno 1, Uno 2 and Uno 3, respectively. Since the Raspberry Pi 3 module operates independently of the Unos and is only connected to its own power source and camera, it will not be discussed further in this section.

Due to the power output limitations of both the battery packs and Uno, Uno 1 is needed to power the Bluetooth connection and send the appropriate steering and speed control input to the other two Unos. This Uno is powered by a 9-V battery pack consisting of 6 AA batteries in series. The connection is made to the 2.1-mm DC barrel connector of the Uno. To ease the wired connections for the hardware components, a breadboard is mounted to the vehicle. One positive rail of the breadboard is powered by the 5-V output pin from Uno 1, and the negative rail is connected to a ground pins from Uno 1. The HC-06 Bluetooth module is then mounted to the board so that each pin rests in a different row. The 5-V and GND pins of the HC-06 are attached to the positive and negative rails, respectively. Since the Bluetooth module operates with serial communication, the serial pins of the module should be attached to the serial pins of the Uno 1. The transmit (TX0) of the HC-06 is attached to the pin 0 (RX0) of the Uno, while the receive (RX0) of the module is

connected to pin 1 (TX0). The completed wiring of the HC-06 is shown in Figure 2.5.

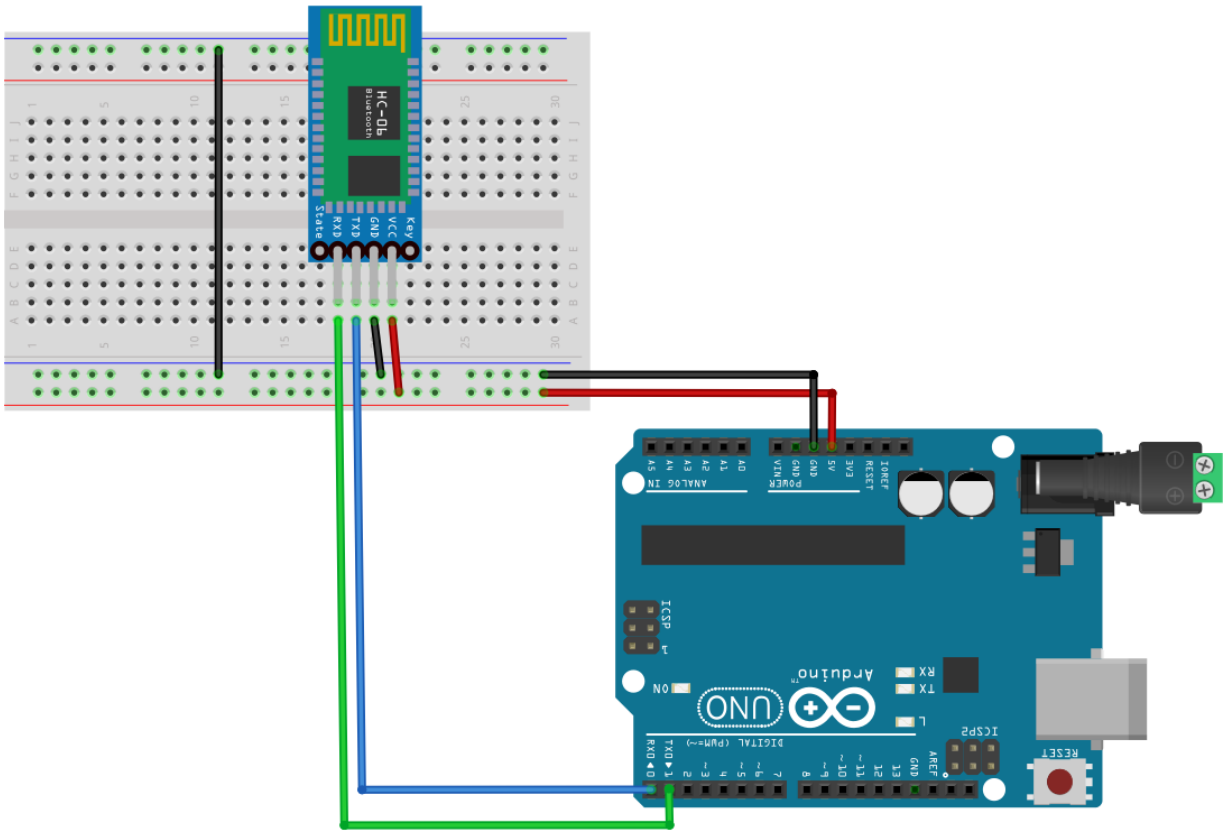


Figure 2.5: Wiring connections from the Uno 1 to the HC-06 Bluetooth module

Once the HC-05 Bluetooth module receives a steering input from the navigational computer, that input must be passed to the steering servo. Due to the great voltage and current requirements of the servo during operation, this is performed on Uno 2 with its own 9-V power supply. The two Unos must communicate with one another using serial communication to pass the steering input. A software serial port is created on pins 10 and 11 of Uno 1, with pin 10 being the receiving pin (RX1) and pin 11 being the transmitting pin (TX1). These respective pins are connected to pins 1 (TX0) and 0 (RX0) of the Uno 2. In order for serial communication to work properly, the Unos must share a common ground, so the grounded rail from one side of the breadboard is connected to the other. Once a ground pin from Uno 2 is connected to the ground rail, the Unos share a common ground. The proper wiring configuration between the serial ports and ground wires of the Unos is shown below in Figure 2.6.

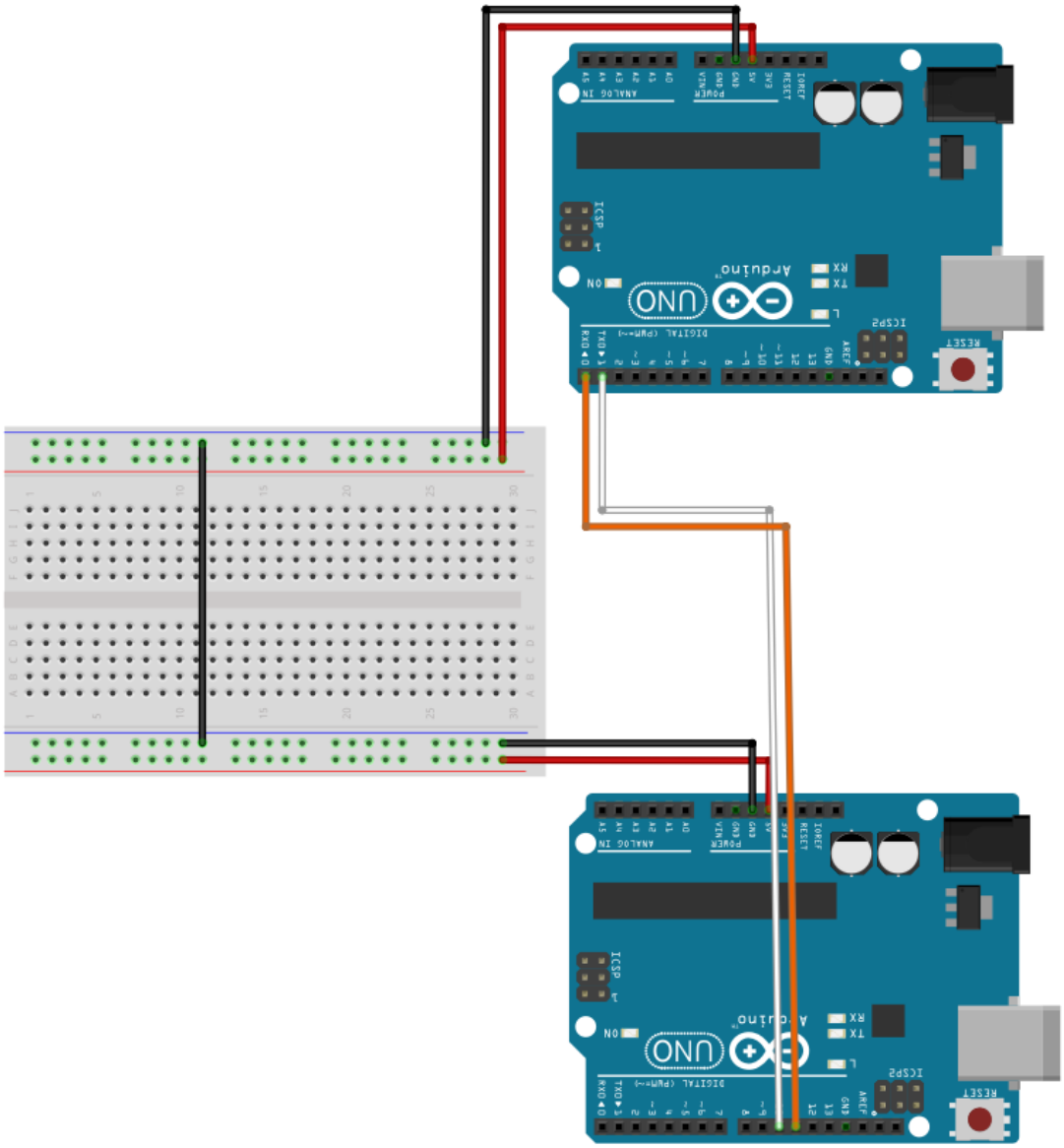


Figure 2.6: Serial wiring connections (orange and white) from pins 10 and 11 of Uno 1 to pins 0 and 1 or Uno 2 with common ground (black)

Now that Uno 2 is receiving the steering angle from Uno 1, that angle must be transmitted to the vehicle steering servo. This servo requires a 5-V power connection and ground supplied by Uno 2. Pin 5 of Uno 2 is selected as the signal output pin to write the appropriate angle to the servomotor. Since the servomotor has a fourth wire which provides the current servo position, this line is connected to the A0 analog pin of Uno 2 so that positional control may be achieved. This is shown in Figure 2.7.

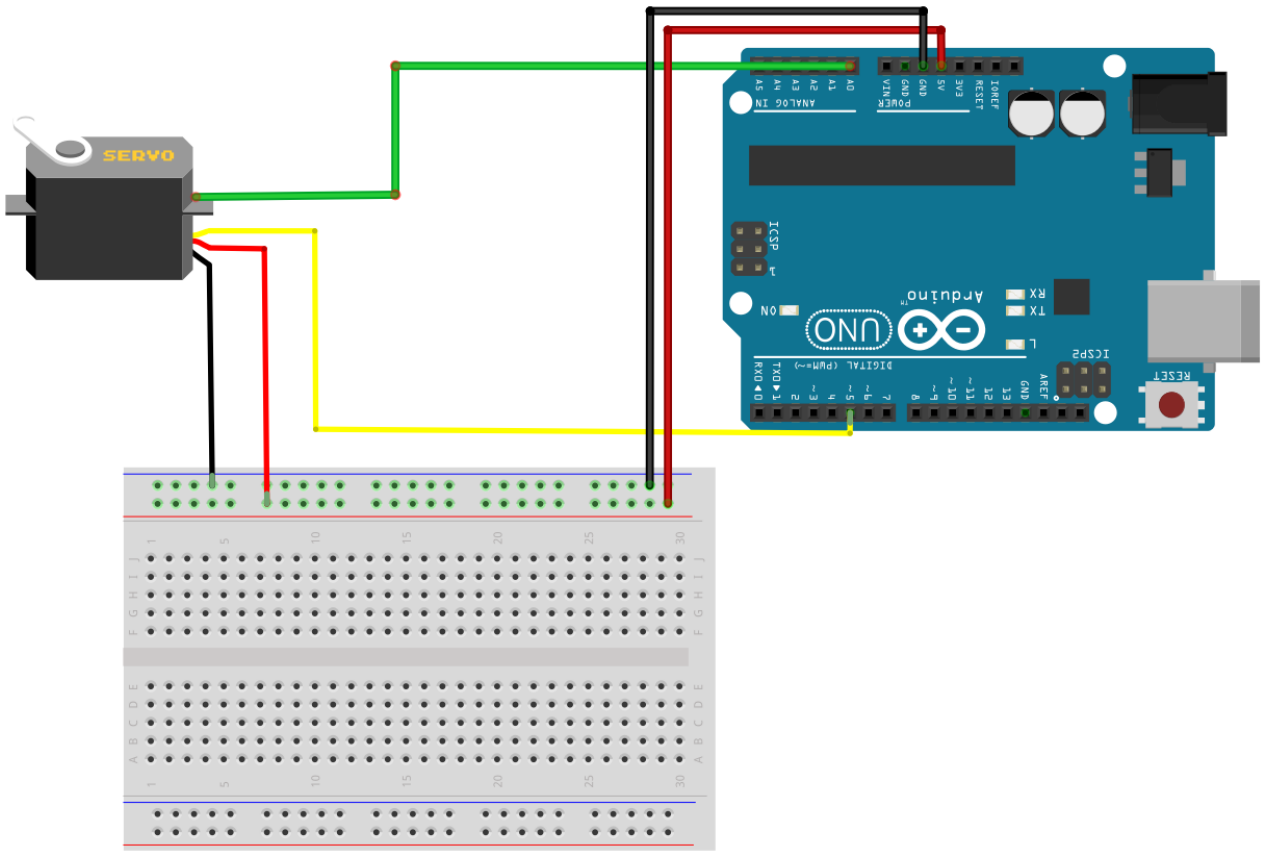


Figure 2.7: Steering servomotor powered by Uno 2 and controlled from pin 5 (yellow). Positional feedback is provided to pin A0 (green)

Uno 3 is responsible for controlling the vehicle speed, and is powered by a third 9-V power supply. To know when to speed or slow the vehicle, Uno 3 must receive the proper command from Uno 1. The initial vehicle start signal is given by the connection of digital pin 7 on Uno 1 to digital pin 7 on Uno 3. When Bluetooth communication is established, Uno 1 sends a `HIGH` signal to tell Uno 3 to begin driving forward at the default speed of 0.46 m/s. Changes in speed are accomplished with serial communication. After Uno 3 is connected to the common ground, a software serial port is created on pins A0 and A1 of Uno 1, with pin A0 being the receiving pin (RX2) and pin A1 being the transmitting pin (TX2). These respective pins are connected to pins 1 (TX0) and 0 (RX0) of the Uno 3. The wiring connection between the serial ports and ground pins of Unos 1 and 3 are shown in Figure 2.8.

The vehicle comes with the XL5 Electronic Speed Controller, which needs a PWM and signal

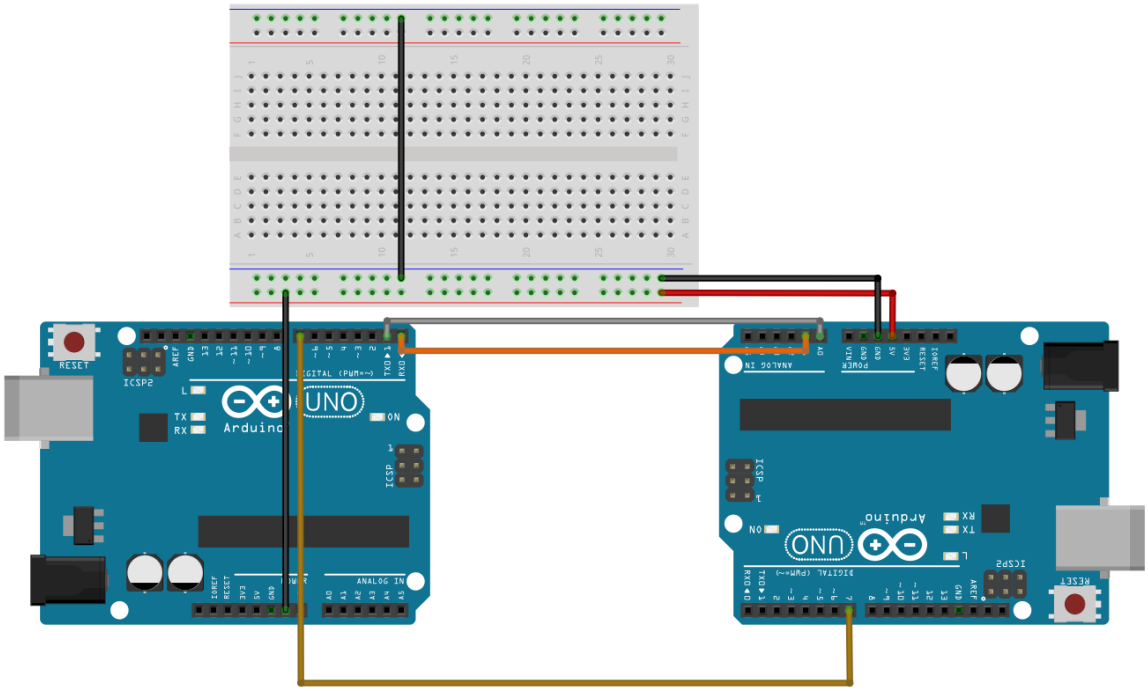


Figure 2.8: Serial wiring connections (orange and gray) from Uno 1 to Uno 3. The initial start command is given by pin 7 of Uno 1 (brown)

ground to operate. After grounding the XL5 to the Uno, a signal wire is connected to PWM pin 5. To speed and slow the vehicle the duty cycle of the PWM signal may be increased or decreased, resulting in increased power to the drive DC motor. This wiring is shown in Figure 2.9. The overall system wiring is shown in Figure 2.10.

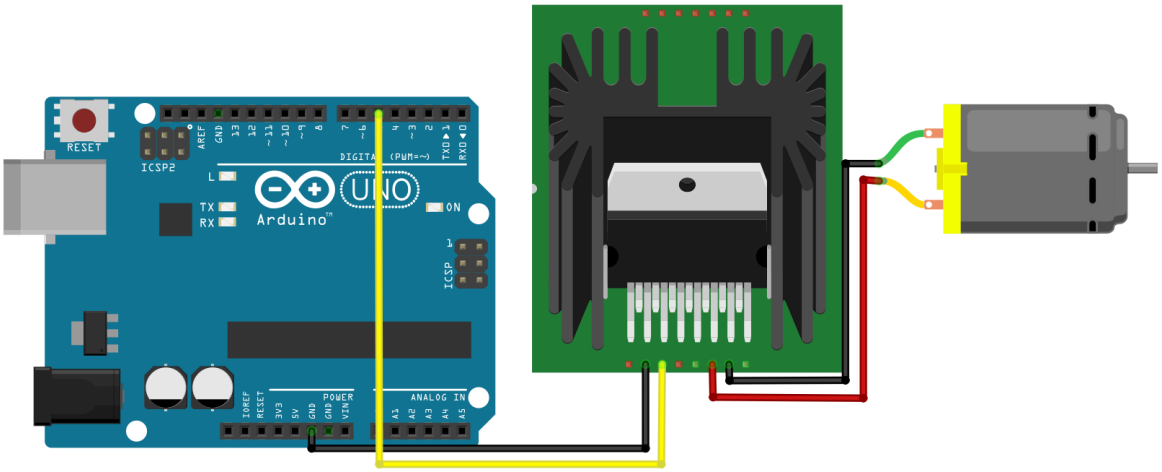


Figure 2.9: Wiring connections from the Uno 1 to the Traxxas XL5 electronic speed controller and driving DC motor

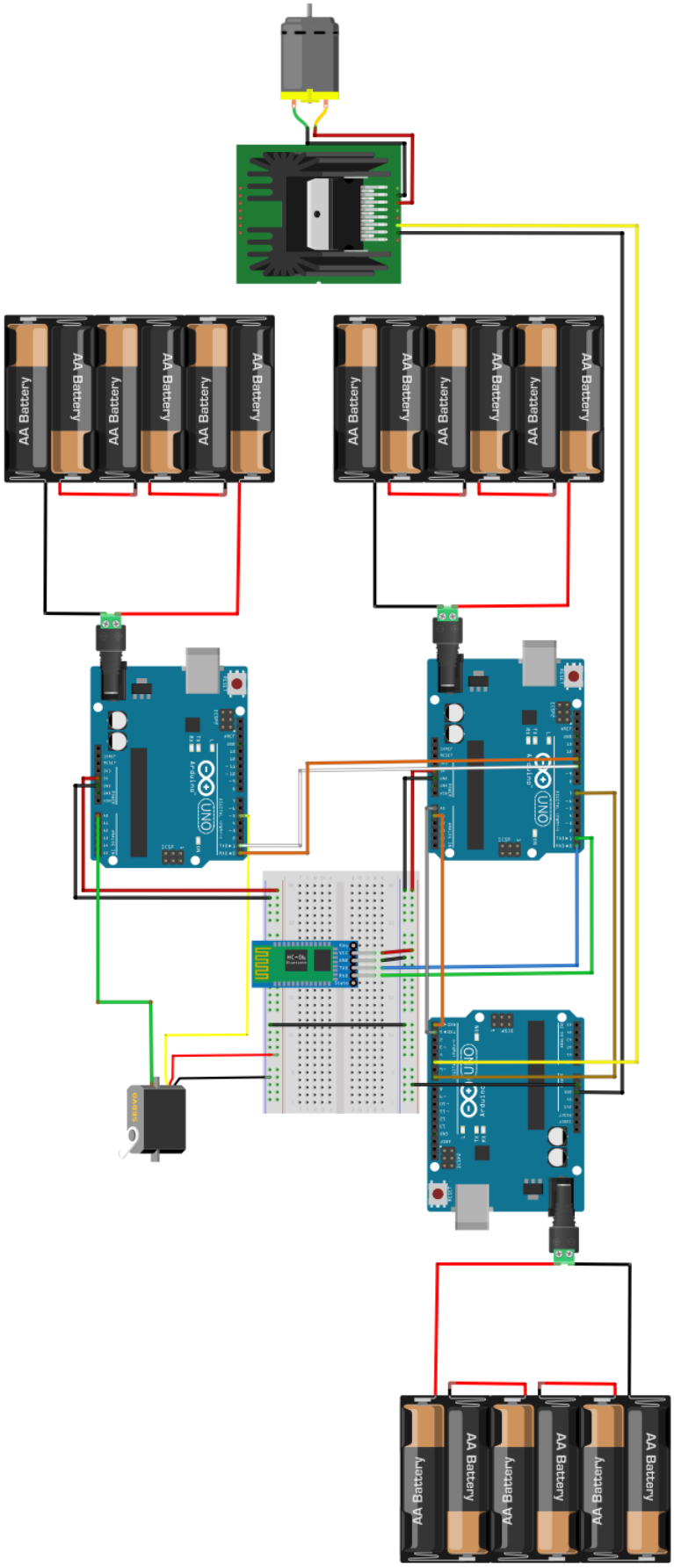


Figure 2.10: Overall vehicle wiring diagram

2.2 Software

All programming on the computer side for this project took place on an Linux Ubuntu 16.04 operating system.

2.2.1 OpenCV

OpenCV is an open-source library intended to assist in computer-vision tasks [24]. The library is created in C++, but has bindings to other languages like Python, which is the language chosen for this project. The release for this project is version 3.0.0. This library is necessary for the lane recognition tasks, which require extensive manipulation of the Raspberry Pi video feed to obtain the navigation information in real-time. Of particular use from the OpenCV library are the pixel color detection, perspective transformation, and Hough line transformation functions, which will be discussed extensively in Chapter 4.

2.2.2 Dlib

Dlib is also an open-source library written in C++ with Python compatability, but is intended more for use in machine learning applications [25]. The version of Dlib used for this project is version 19.4.99. This library is used extensively for the sign recognition tasks. The imglab tool available with the software allows for the user to create an Extensible Markup Language (XML) file from training images of the object being identified. Other functions within the library allow for the training of an SVM using HOG and give the ability to scan images for these trained objects. The step by step use of these functions in sign recognition will be more completely explained in Chapter 4.

2.2.3 Arduino Software

The Arduino integrated development environment (IDE) is used solely for the creation of programs meant to run on the Arduino. Arduino commands are based on C/C++ commands with the exception of a few program-specific keywords. Since the Arduino boards themselves have numerous input and output pins, this IDE is most useful for creating the programs that control the

steering servo and velocity control of the driving DC motor. Several libraries within the Arduino environment, such as the Servo and Serial communication are used in these programs.

3. CONTROLLER DESIGN

This chapter discusses the derivation of the dynamic vehicle model. This model is ultimately used for the design and tuning of the steering controller which accurately navigates the vehicle.

3.1 Dynamic Vehicle Model

Since the lateral positioning of the vehicle within the lane is the motion of interest, it is only necessary to study the vehicle motion about the yaw-axis. With this assumption, the vehicle dynamics can be simplified to the two-degree-of-freedom (2-DOF) bicycle model shown below in Figure 3.1 and in [26], [27].

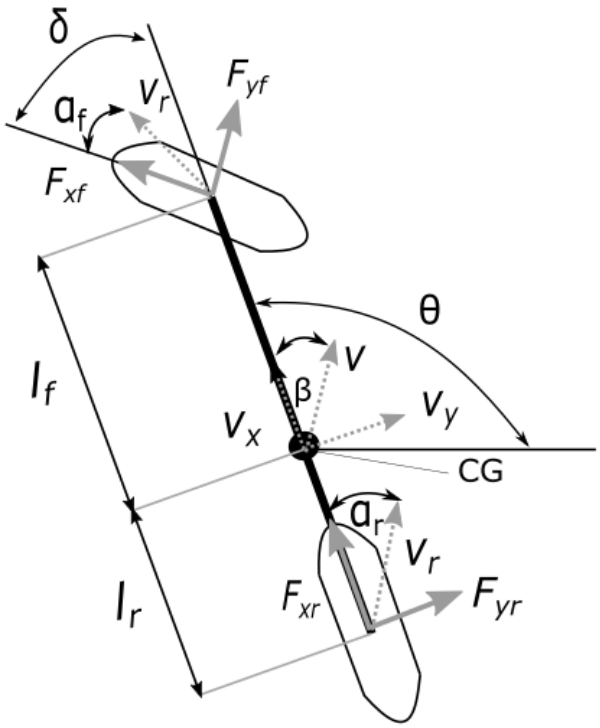


Figure 3.1: 2-DOF bicycle model representing vehicle dynamics, adapted from [26]

In this model the four wheels are combined into a single front and back wheel. The center of gravity (CG) lies at the point (X_c, Y_c) in the global reference frame and the vehicle is oriented at angle θ_c in this frame. The wheels are exposed to longitudinal forces F_{xf} and F_{xr} and to lateral forces F_{yf} and F_{yr} applied at the front and rear axles a distance l_f and l_r from the vehicle CG. The

front and rear wheels have velocity vectors v_f and v_r with the slip angles α_f and α_r measuring the difference between the direction of travel and the force vector causing the wheel to corner. Similarly, the body slip angle β describes the angle between the vehicle orientation and direction of travel, and is mathematically equivalent to v_y/v_x, where v_x and v_y are the longitudinal and lateral velocity. The angle δ is the steering angle of the front wheel relative to the vehicle's orientation. The yaw rate of the vehicle about the CG is denoted by $\dot{\psi}$.

With the vehicle having mass m, taking the sum of forces in the lateral direction of the vehicle body reference frame attached to the vehicle CG results in

$$F_{yf}\cos\delta + F_{xf}\sin\delta + F_{yr} = m(\dot{v_y} + v_x\dot{\psi}) \tag{3.1}$$

Taking the sum of moments about the center of gravity of the vehicle with inertia I_z gives

$$l_f(F_{yf}\cos\delta) - l_r(F_{yr} - F_{xf}\sin\delta) = I_z\ddot{\psi} \tag{3.2}$$

The two tire slip angles for the front and back tires are given as

$$\alpha_f = \tan^{-1}(\frac{v_y + l_f\dot{\psi}}{v_x}) - \delta \tag{3.3}$$

$$\alpha_r = \tan^{-1}(\frac{v_y - l_r\dot{\psi}}{v_x}) \tag{3.4}$$

Making the assumption that assumption that these slip angles are small allows for the lateral force to be a function of the slip angle and the front and rear tire cornering stiffness c_f and c_r

$$F_{yf} = -c_f\alpha_f = -c_f\tan^{-1}(\frac{v_y + l_f\dot{\psi}}{v_x}) \tag{3.5}$$

$$F_{yr} = -c_r\alpha_r = -c_r\tan^{-1}(\frac{v_y - l_r\dot{\psi}}{v_x}) \tag{3.6}$$

Since the vehicle travels forward with constant velocity v_x, $\dot{v_x} = 0$ and the longitudinal force

on the front tire $F_{xf} = 0$. Substituting (3.5)-(3.6) into (3.1)-(3.2) and solving for the body slip angle rate $\dot{\beta}$ and yaw acceleration $\ddot{\psi}$ gives the following nonlinear equations of motion.

$$\dot{\beta} = \frac{-c_f[\tan^{-1}(\frac{v_y + l_f\dot{\psi}}{v_x}) - \delta]\cos\delta - c_r\tan^{-1}(\frac{v_y - l_r\dot{\psi}}{v_x})}{mv_x} - \dot{\psi} \tag{3.7}$$

$$\ddot{\psi} = \frac{-l_f c_f[\tan^{-1}(\frac{v_y + l_f\dot{\psi}}{v_x}) - \delta]\cos\delta + l_r c_r\tan^{-1}(\frac{v_y - l_r\dot{\psi}}{v_x})}{I_z} - \dot{\psi} \tag{3.8}$$

Using the small angle assumption, (3.7)-(3.8) can be linearized to

$$\dot{\beta} = -\frac{(c_r + c_f)}{mv_x}\beta + (-1 + \frac{l_r c_r - l_f c_f}{m{v_x}^2})\dot{\psi} + \frac{c_f}{mv_x}\delta \tag{3.9}$$

$$\ddot{\psi} = \frac{(l_r c_r - l_f c_f)}{I_z}\beta - (\frac{{l_r}^2 c_r + {l_f}^2 c_f}{I_z v_x})\dot{\psi} + \frac{c_f l_f}{I_z}\delta \tag{3.10}$$

Writing the model in state space form of $\dot{\boldsymbol{x}} = A\boldsymbol{x} + Bu$ where the states are given by $\boldsymbol{x} = [\beta, \dot{\psi}]^T$ and $u = \delta$ results in

$$\begin{bmatrix} \dot{\beta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} -\frac{(c_r + c_f)}{mv_x} & -1 + \frac{l_r c_r - l_f c_f}{m{v_x}^2} \\ \frac{(l_r c_r - l_f c_f)}{I_z} & -\frac{{l_r}^2 c_r + {l_f}^2 c_f}{I_z v_x} \end{bmatrix} \begin{bmatrix} \beta \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} \frac{c_f}{mv_x} \\ \frac{c_f l_f}{I_z} \end{bmatrix} \delta \tag{3.11}$$

3.2 Pure-Pursuit Path Navigation

The controller used for navigating the vehicle is a modified form of pure-pursuit path tracking algorithm similar to that presented in [28]. The geometry for this tracking algorithm is shown in Figure 3.2. In normal pure-pursuit path tracking, the vehicle tracks towards a known point (X_p, Y_p) on a reference path of constant curvature $\kappa = \frac{1}{R}$. The look-ahead angle ϕ measures the angle between the point (X_c, Y_c) located on the rear tire and the reference point (X_p, Y_p). The straight line distance between these points is known as the look-ahead distance, noted as L_d. Using the law of sines, the curvature κ is calculated as

$$\kappa = \frac{2\sin\phi}{L_d} \tag{3.12}$$

The steering angle δ is then given as

$$\delta = \tan^{-1}(\kappa L_a) \tag{3.13}$$

where L_a defines the distance between the front and rear axles. While this pure-pursuit method can appropriately steer the vehicle to the desired point, it does not take into consideration the tangential angle of the desired point, so there is no guarantee that the vehicle will arrive at the desired point with the correct heading. Additionally, the vehicle in this project navigates without having prior mapped points to track, so these pursued points must be created from observation of the track in real time.

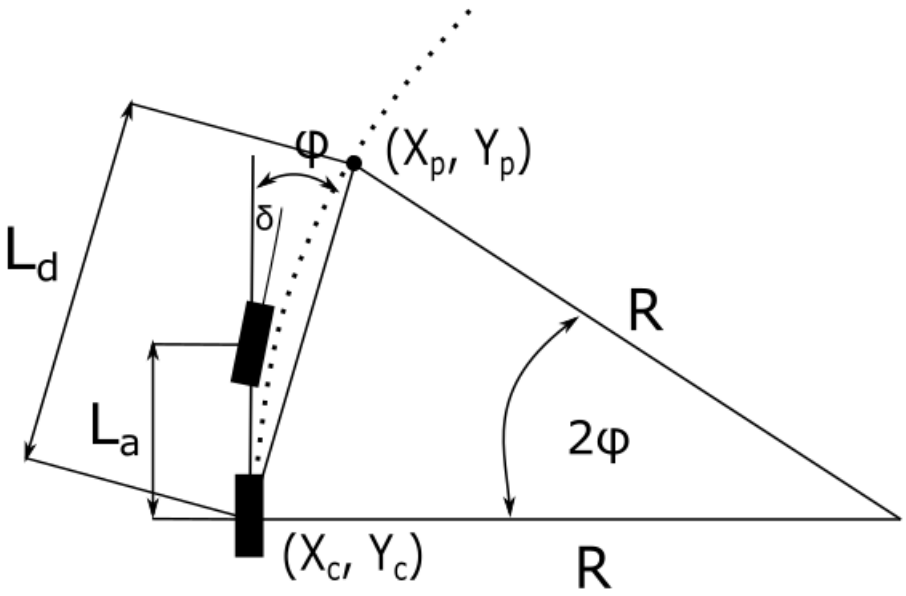


Figure 3.2: Pure-pursuit tracking geometry, adapted from [28]

3.3 Modified Pure-Pursuit Path Navigation

The largest difficulty in calculating the the pursued point (X_p, Y_p) in real time is the fact that the vehicle only has a single fixed camera that always points straight ahead at the current vehicle

heading. Therefore there are moments, such as during a sharp turn, where the vehicle might not even be able to see the pursued point. However, by calculating the track slope at the point immediately ahead of the vehicle and using the known track width, the pursued point can be found.

As the vehicle approaches a turn, the camera points forward at the current vehicle heading and scans for the lane boundary at a set distance L_c as indicated in Figure 3.3. Using the computer vision algorithms discussed later in Section 4.2, the slope of the lane boundary θ_p is calculated at the scan point.

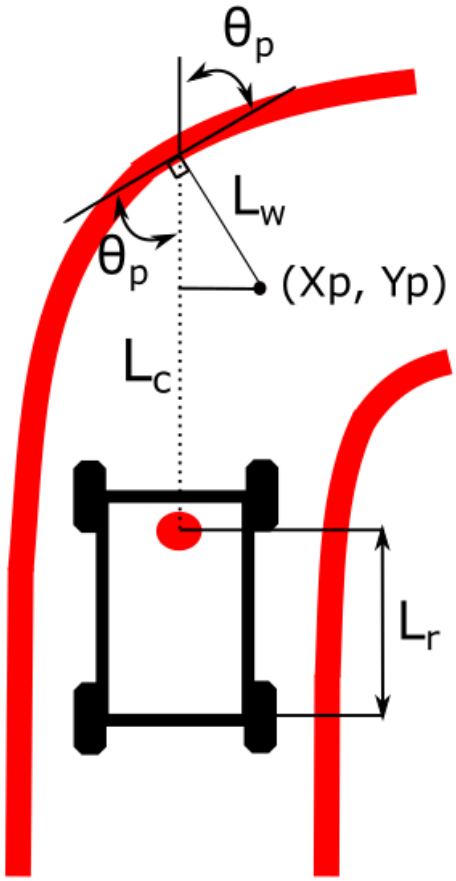


Figure 3.3: Pursued point projection from the lane boundary

From the calculated slope at the scan point, a normal projection is made away from the scan point. Ideally, the vehicle will navigate exactly in the middle of the lane. Since the width of the lane is known, the projection has a length L_w of half of the lane width. The lateral distance Y_p of the point to the camera can then be calculated as

$$Y_p = L_w \sin(\frac{\pi}{2} - \theta_p) \tag{3.14}$$

By the pure-pursuit method, the longitudinal distance must be measured from the rear axle, so the distance X_p is given as

$$X_p = L_c - L_w \cos(\frac{\pi}{2} - \theta_p) + L_r \tag{3.15}$$

With the pursued points (X_p, Y_p) now calculated, the method for determining the proper steering angle taking into account the pursued point heading is a modified pure-pursuit method presented in [28]. From the pursuit geometry shown in Figure 3.4, the slope of the path θ_p at the scan point can be denoted geometrically as

$$\theta_p = \theta + 2\phi_d \tag{3.16}$$

where θ is the current heading angle of the vehicle and ϕ is the look-ahead angle from the rear axle to a point offset from the pursued point by a distance d. Rearranging, the heading angle is given as a function of the path slope and current vehicle trajectory by

$$\phi_d = \frac{\theta_p - \theta}{2} \tag{3.17}$$

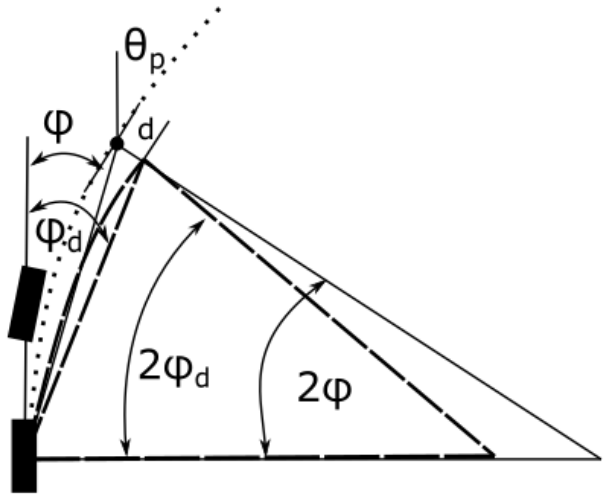


Figure 3.4: Modified pure-pursuit tracking geometry, taking into account the path curvature

The distance d and the new look-ahead distance are given by

$$\begin{bmatrix} d \\ L_d \end{bmatrix} = \begin{bmatrix} \cos(\theta_p + \frac{\pi}{2}) & -\cos(\theta + \phi) \\ \sin(\theta_p + \frac{\pi}{2}) & -\sin(\theta + \phi) \end{bmatrix}^{-1} \begin{bmatrix} X - X_p \\ Y - Y_p \end{bmatrix} \tag{3.18}$$

Using the new look-ahead distance and look-ahead angle, the modified steering angle can be calculated as in the pure-pursuit method.

3.4 Vehicle Speed Adjustment

Due to vehicle's electronic speed controller automatically controlling the voltage sent to the drive motor to maintain the input speed, it is difficult to implement a classical speed controller such as proportional-integral-derivative control. Therefore, the vehicle speed can only be modified by changing the PWM setpoint. In this particular project, the vehicle can be in only one of three discrete speed states.

State 1 is the default speed state. In this state, a constant speed of 0.45 m/s is written to the drive motor. This state is set from the moment the first steering angles are received by Uno 1, and continues indefinitely unless a street sign is detected. State 2 occurs whenever the vehicle detects the speed limit sign. The vehicle set speed is immediately increased to 0.67 m/s. Now that the vehicle has entered a high speed zone, it will continue to maintain this higher speed state until another street sign is detected. State 3 is the stop state, which as the name implies occurs when the vehicle detects a stop sign. The vehicle speed is then set to 0 m/s, and the vehicle will not move until reset. The determination of the speed state can be seen on the right branch of the logic flow diagram in Figure 4.1.

4. LANE NAVIGATION AND SIGN DETECTION METHODS

To successfully complete the two main project objectives, numerous sub-tasks must be performed. Among these are streaming from the Raspberry Pi to the computer, identifying lane-markers and street-signs in the image, creating navigation paths from these images and then passing that information to an Uno to actuate the vehicle. All of these tasks are achieved through Python scripts and Arduino sketches. The overall structure of the tasks performed for completion of the project objectives are presented in Figure 4.1.

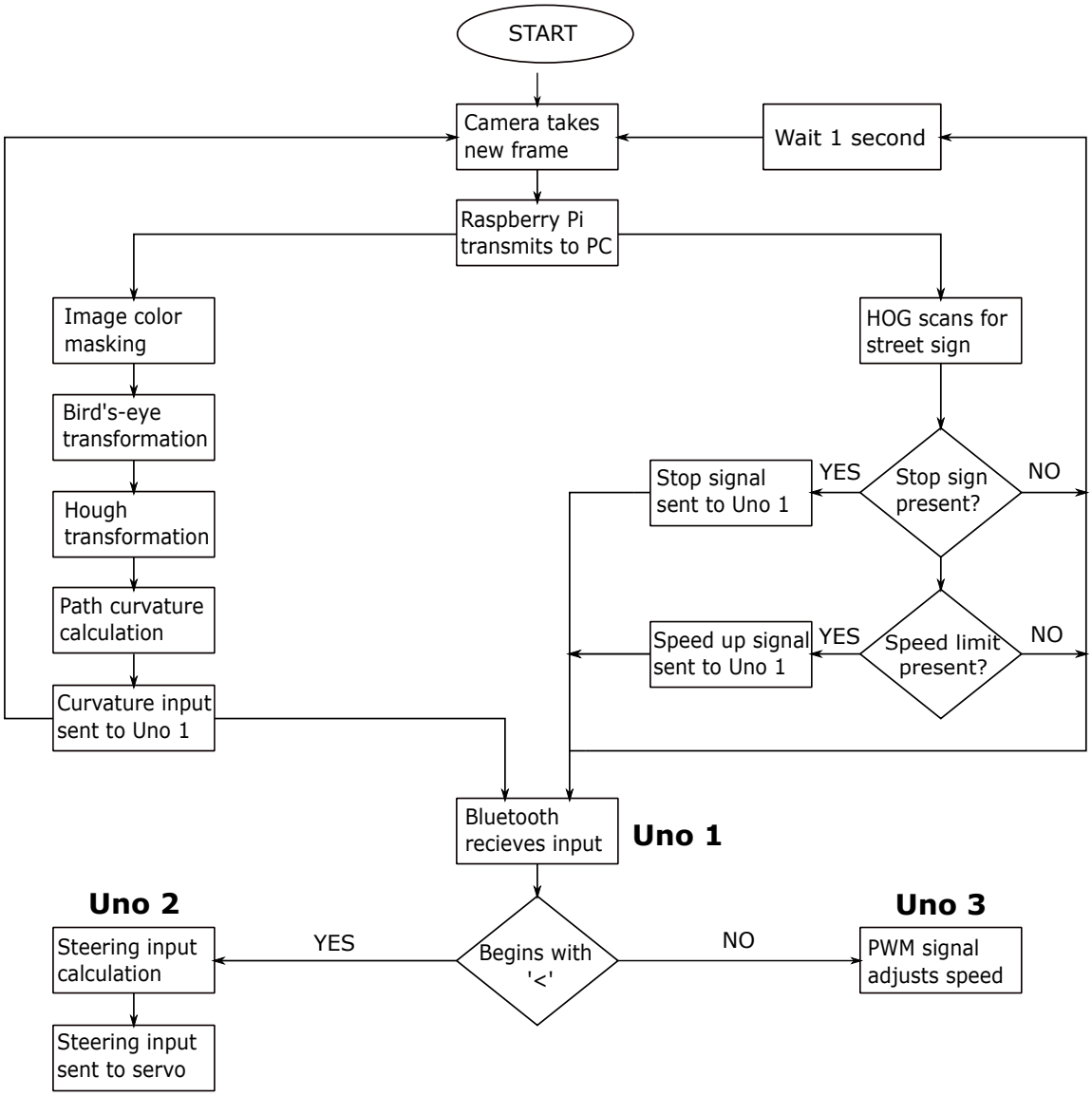


Figure 4.1: Overall program logic flow diagram

4.1 Video Streaming from Raspberry Pi to Computer

While the Raspberry Pi is useful for initially capturing the video images from the Arducam, it does not have the processing power necessary to perform the computationally heavy tasks involved with scanning new images for the street signs and lane markers. On a larger vehicle an on-board computer with higher processing capabilities would perform these calculations, but due to the limited space on the selected test vehicle this option is impractical. In order to perform both tasks, the video stream instead shall be passed wirelessly from the Raspberry Pi to a computer so that the vision processing can be performed there.

4.1.1 Server Streaming

The code to perform this process from the computer (server) side of the streaming is shown below in Code 4.1. This script is not run by itself, but is rather contained within the larger lane detection and sign detection scripts which will be discussed in Sections 4.2 and 4.3. The server side script must be run first so that when the Raspberry Pi begins sending bytes across the stream there is a script ready to listen for and accept the bytes. This code is provided from the PiCamera release-1.13 example programs for capturing to a network stream [29].

Code 4.1: PC(server): Receipt and display of Raspberry Pi video stream

```
import io
import socket
import struct
import numpy as np
import cv2

server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

connection = server_socket.accept()[0].makefile('rb')
try:
   while True:
      image_lenstruct.unpack('<L',connection.read(struct.calcsize('<L'))[0]
       if not image_len:
          break
       image_stream = io.BytesIO()
       image_stream.write(connection.read(image_len))
       image_stream.seek(0)
       image = Image.open(image_stream)
       cv_image = np.array(image)
       cv2.imshow('Steam',cv_image)
       cv2.waitKey(1)
```

```
finally:
    connection.close()
    server_socket.close()
```

This script first creates a socket, which is just another name for an endpoint of a two way communication stream. The socket is bound to 0.0.0.0, which means all interfaces available to be PC, and then begins listening on port 8000 for any bytes coming across that port. The file object `connection` is created and specified with `'rb'` to make the file object open for reading (r) in binary mode (b). The image stream is then unpacked and the bytes on the stream are read as an unsigned 32-bit integer. The stream of incoming bytes is then repackaged into a numPy array of the same shape as the incoming image. Once the bytes for each pixel arrive, the numPy array can be displayed as an image. When the stream stops receiving bytes the connection and socket are closed.

4.1.2 Client Streaming

The Raspberry Pi (client) script shown in Code 4.2 is provided from the PiCamera release-1.13 rapid capture and streaming example [30]. This script must only be run once the server script has been executed or there will be no available connection for the client. The internet protocol (IP) address of the server must also be known before running the client script.

Code 4.2: Raspberry Pi(client): Stream video from camera to PC

```
import io
import socket
import struct
import time
import picamera

class SplitFrames(object):
    def __init__(self, connection):
        self.connection = connection
        self.stream = io.BytesIO()
        self.count = 0

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            size = self.stream.tell()
            if size > 0:
                self.connection.write(struct.pack('<L', size))
                self.connection.flush()
                self.stream.seek(0)
                self.connection.write(self.stream.read(size))
```

```
            self.count += 1
            self.stream.seek(0)
        self.stream.write(buf)


client_socket = socket.socket()
client_socket.connect(('server_IP_address', 8000))
connection = client_socket.makefile('wb')
try:
    output = SplitFrames(connection)
    with picamera.PiCamera(resolution='VGA', framerate=30) as camera:
        time.sleep(2)
        start = time.time()
        camera.start_recording(output, format='mjpeg')
        camera.wait_recording(130)
        camera.stop_recording()

        connection.write(struct.pack('<L', 0))
finally:
    connection.close()
    client_socket.close()
    finish = time.time()
print('Sent %d images in %d seconds at %.2ffps' % (
    output.count, finish-start, output.count / (finish-start)))
```

The script first creates a socket and connects the socket to the IP address of the computer at port 8000. The file object `connection` is then created and specified with `'wb'` to make the file object open for writing (w) in binary mode (b). The object `output` is then created from the `SplitFrames()` class. `SplitFrames()` allows the Raspberry Pi to split individual JPEG images from the MJPEG file being recorded by the Raspberry Pi and then send the image over the socket connected to the other computer. Since all JPEG images begin with the same two bytes, `buf.startswith()` looks to see if those two bytes begin the buffer, and if they do the image is written to the server. Now that there is data available to the PC, an image will appear on the computer similar to Figure 4.2. The Raspberry Pi camera records for a specified number of seconds, after which the recording is stopped and the connection closed. The Raspberry Pi then reports how many frames were sent across the stream and outputs the fps.

Figure 4.2: Representative streaming image from the Raspberry Pi to the controlling PC

4.2 Lane Detection and Navigation

By far the most intensive program is the script used to perform the lane detection and navigation. This code must be capable of identifying lane markers, using the identified lane to calculate a trajectory for the vehicle and then actuating the vehicle steering servomotor to the necessary angle. The entirety of the program is included in the appendices, but important sections of the code are highlighted below.

4.2.1 Color Masking

Once the video is streaming to the PC, the process of detecting the lane lines begins by isolating the lines from the background. This is done with color masking, which retains all of the red pixels in the image and eliminates the others. The color masking code is shown below in Code 4.3.

Code 4.3: PC: Red color masking of received video from Raspberry Pi

```
import cv2

hsv = cv2.cvtColor(cv_image, cv2.COLOR_RGB2HSV)

lower_red = np.array([0,100,100])
upper_red = np.array([20,255,255])

mask = cv2.inRange(hsv, lower_red, upper_red)
```

The image is transformed from RGB color space to HSV color space, which was discussed in Section 1.4.3. Now that the image is in the HSV color space, a range is created which will correspond to the upper and lower limits of what will be considered "red" in the input image. The `inRange()` function then scans the input image for pixels matching that color threshold, and rejects all pixels not falling within the range. For the given input image from Figure 4.2, the resulting output image would resemble the frame shown in Figure 4.3.

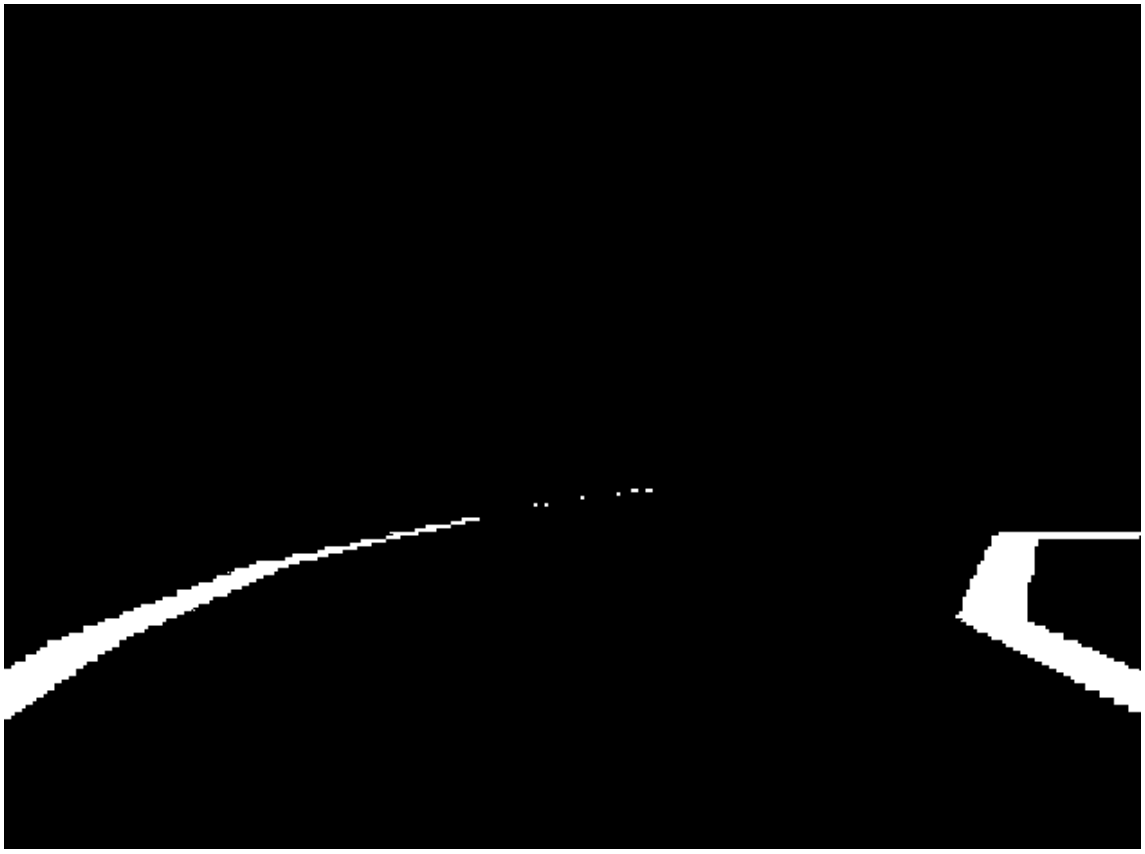


Figure 4.3: Red color masked image of the input image from Figure 4.1. Note that red pixels have been retained and all others rejected

4.2.2 Perspective Transformation

Now that the lane lines have been isolated from the background, the main objective is to navigate the vehicle around the path as close to the center of the lane as possible. To properly navigate a turn, the vehicle must have some way of calculating the proper trajectory to match the turn radius of curvature. However, due to the perspective of the camera mounted to the front of the vehicle, parallel lines in the real world will appear to slope towards one another as the distance from the vehicle increases. To remove this perspective and make a trajectory seen by the camera match the trajectory in the real world, a perspective transformation is performed.

A perspective transformation is simply a linear mapping of image pixels from one space to another. In this case the mapping is from the camera perspective to an overhead perspective that is in the real world, also called a bird's-eye perspective. To perform this linear mapping, four points from the original perspective and the corresponding points in the new mapped space are needed. Of these points, no three may be collinear. To find these points, the vehicle is placed as shown in Figure 4.4. The camera is positioned to lie in the middle of the track, 0.305 m (1 ft) from either edge. The points lie at distances of 0.61 m (2ft) and 0.914 m (3ft) from the camera, respectively. Once the vehicle has been positioned, the points seen in Figure 4.5 are superimposed onto the camera stream, and manually adjusted until they lie exactly upon the selected reference points from the overhead view.

Code 4.4: PC: Perspective transformation of red color masked image to bird's-eye perspective

```
image = Image.open(image_stream)
cv_image = np.array(image)

cv2.circle(cv_image,(10,378),1,(0,0,255),-1) #lower left point
cv2.circle(cv_image,(110,321),1,(0,0,255),-1) #upper left point
cv2.circle(cv_image,(519,324),1,(0,0,255),-1) #upper right point
cv2.circle(cv_image,(616,384),1,(0,0,255),-1) #lower right point

pts1 = np.float32([[10,378],[110,321],[519,324],[616,384]])
pts2 = np.float32([[220,280],[220,180],[420,180],[420,280]])

M = cv2.getPerspectiveTransform(pts1,pts2)

dst = cv2.warpPerspective(mask,M,(640,480))
```

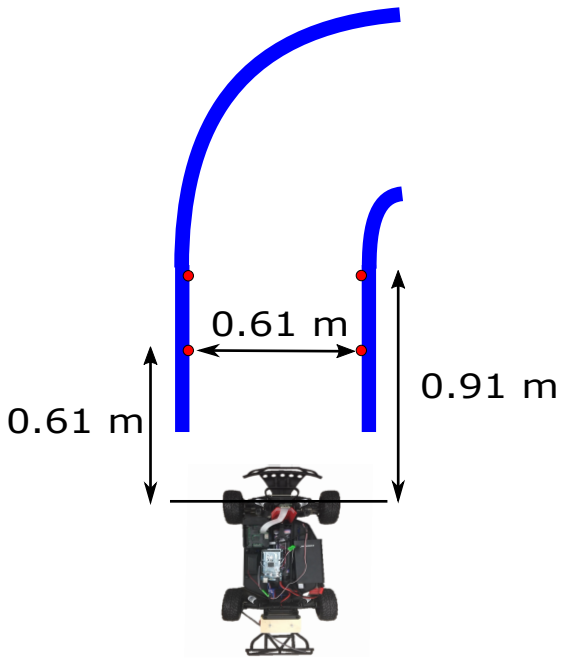


Figure 4.4: Overhead positioning of car with respect to the reference points used for perspective transformation

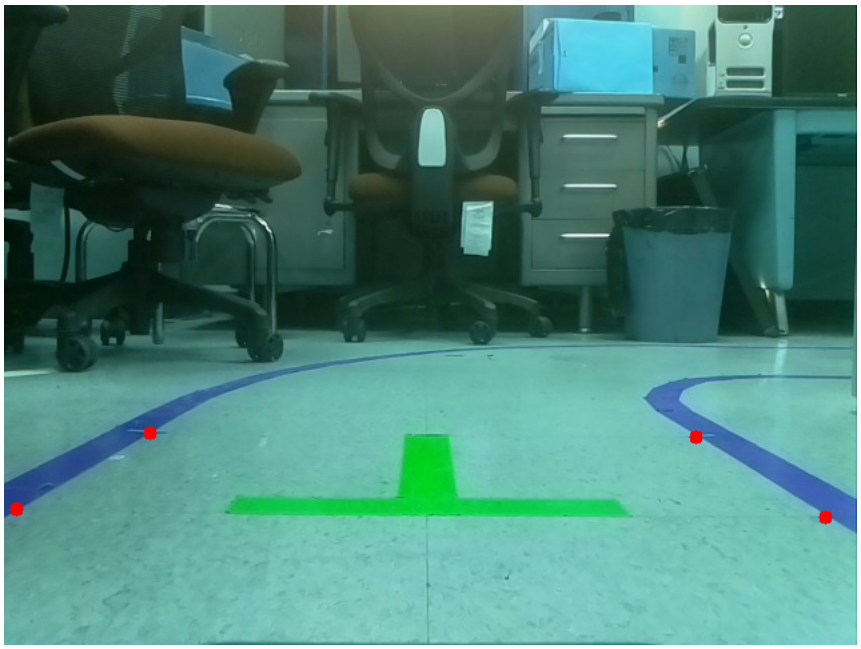


Figure 4.5: Camera perspective from the vehicle positioned as in the image to the left, with the same reference points superimposed over the streamed image

Once the points are placed as desired, the new mapping of points must be created. The resolution of the Raspberry Pi camera is 640×480 pixels, so the new mapping should be of the same

resolution. To take advantage of the whole mapped space, the camera will be assumed to lie at the point (320, 480), which is the bottom point centered exactly in the middle of the new mapped space. Using a scale of 100 pixels/ft, the points from bottom left and proceeding clockwise will be mapped to (220,280), (220,180), (420,180) and (420,280), respectively. Now that the points have been selected, `getPerspectiveTransform()` finds the matrix which transforms the first set of points to the new perspective space. Finally, `warpPerspective()` transforms every pixel from the color masked image of Figure 4.3 to the new perspective space using the matrix solved for in the previous step. The code used to perform the perspective transformation is shown above in Code 4.4. The resulting perspective transformed image gives the overhead view of the track as seen in Figure 4.6.

Figure 4.6: Overhead perspective transformed image of Figure 4.2, showing how the track appears from straight above

4.2.3 Hough Transformation

With the perspective now transformed to the overhead view needed for navigation, the Hough transformation is implemented to group the color masked pixels into line segments. Transforming the pixels into line segments will allow the curvature of the track to be calculated in real time. The process of transforming pixels into line segments is discussed fully in Section 1.4.4. The commands to perform the Hough transformation are shown in Code 4.5.

The python library `cv2` has a built in Hough transformation function `HoughLinesP()` which accepts the arguments `rho`, `theta`, `threshold`, `min_line_len` and `max_line_gap`. The parameters `rho` and `theta` are the resolution of the calculated ρ and θ from the Hough transformation. In this case the resolution is set to 1 pixel and 1°, respectively. The input `threshold` sets the minimum number of intersections which must occur for a point (θ, ρ) to detect a line segment. Also, to be considered a line segment, the line must have a number of pixels greater than the `min_line_len` parameter. Finally, `max_line_gap` sets the maximum gap that can be between two points to be considered part of the same line.

Code 4.5: PC; Hough Transformation of red color masked pixels to line segments

```
rho = 1
theta = np.pi/180
threshold = 20
min_line_len = 10
max_line_gap = 20

lines = cv2.HoughLinesP(img, rho, theta, threshold,
   np.array([]), minLineLength=min_line_len,
   maxLineGap=max_line_gap)
```

The output `lines` is a set of points [[x_1 y_1 x_2 y_2], [x_3 y_3 x_4 y_4], ...] corresponding to the endpoints of every line segment of the Hough transformation. With the pixels transformed into line segments, a trajectory for the vehicle can be calculated.

4.2.4 Trajectory Calculation

Not every detected line segment from the Hough transformation is needed to calculate the trajectory of the vehicle. Some lines will correspond to far ahead turns which the vehicle has not yet reached. Since the closest point the vehicle camera can see is 0.76 m (2.5 ft) away, a method

is needed to identify all lines which intersect with an imaginary line 0.76 m ahead of the vehicle. This is accomplished with Code 4.6. The line that will be used to detect intersecting segments 0.76 m ahead is created across the width of the overhead viewing perspective. The line is placed 250 pixels from the camera, recalling that a scale of 100 pixels per foot has been established.

Code 4.6: PC: Filtering of detected Hough lines with search line intersection

```
scan_line = (0,230,640,230) # X1, Y1, X2, Y2

def intersect(line_one,line_two):
   X1,Y1,X2,Y2 = line_one
   X3,Y3,X4,Y4 = line_two


   A1 = (Y1-Y2)/(X1-X2)
   A2 = (Y3-Y4)/(X3-X4)
   b1 = Y1-A1*X1
   b2 = Y3-A2*X3

   if (A1 == A2):
      return False;

   Xa = (b2 - b1) / (A1 - A2)

   if ( (Xa < max( min(X1,X2), min(X3,X4) )) or (Xa > min( max(X1,X2),
      max(X3,X4) )) ):
      return False;
   else:
      return True;
```

The function `intersect` takes two line segments as the input arguments, and breaks the lines down into their endpoint coordinates. Line one will always be the imaginary line 0.76 m ahead of the vehicle, and line two will be one of the line segments detected from the Hough transformation. With these endpoints, the slopes of the line segments `A1` and `A2` are calculated, along with their corresponding y-intercepts `b1` and `b2`. If `A1` is equal to `A2`, the lines must be parallel, and can therefore not intersect. The next check is to see if the intersection point of the two line segments falls within the bounds of the the maximum and minimum endpoints. The variable `Xa` is the x-coordinate of the intersection point, if it in fact exists. If this calculated intersection point is less than the smallest x-coordinate or greater than the largest x-coordinate of either of the line segments, the line segments do not in fact intersect. If all of these conditions are false, the lines must intersect, and the Hough line will be used for calculation of the vehicle trajectory.

With the lines that have been retained, the trajectory may now be calculated. The accepted line

segments are sorted by whether they define the left or right lane line by seeing if the x-coordinate of the first endpoint is greater or less than the midpoint at 320 pixels. Once the line segments are sorted to the left and right, the average `X1`, `Y1`, `X2` and `Y2` are calculated for each group to establish the definitive left and right lane boundaries. The average is calculated to smooth any large variations that might come from a single outlier line segment. Now that the average left and right boundaries have been calculated, the trajectory that navigates most closely to the middle of the lane is the average of the left and right boundaries. To again smooth any large oscillations that might occur from an outlier frame, the navigation trajectory is the average of the previous 10 frames of trajectory data. The trajectory path for the vehicle is then drawn onto the Hough lines image, as shown below in Figure 4.7.

Figure 4.7: Overhead perspective of the Hough line transformed track from Figure 3.5, with search line (horizontal line) and navigation trajectory (vertical middle line) projected onto the image

With the trajectory line calculated, the final step is to calculate the angle that must be sent to the steering servomotor. Using the endpoints of the navigation trajectory line, the path slope angle is the inverse tangent of the difference between the x-coordinates divided by the difference between the y-coordinates.

4.2.5 Bluetooth Connection

Once the angle of the path has been calculated, the program must pass that angle to an on board Arduino so that the turn can be executed. This is accomplished with Bluetooth. The Bluetooth communication code is shown below in Code 4.7.

Code 4.7: PC: Bluetooth connection with Uno 1 and transmission of path slope input

```
import serial

port="/dev/rfcomm0"
bluetooth=serial.Serial(port, 9600)
bluetooth.flushInput()
bluetooth.write(str.encode(str('<')))
bluetooth.write(str.encode(str((int(turn_angle)))))
bluetooth.write(str.encdoe(str('>')))
```

The data set is transferred to the Bluetooth module with serial communication. Serial communication is simply sending the data one bit at a time in sequential order from the sender to receiver. The program is directed to the location of the HC-06 communication port, which in this case is located in /dev/rfcomm0. If the module has previously been synced with the computer the connection is established. The serial communication is then initialized on this port at a baud rate of 9600 bits per second. After flushing the input to assure that no random bytes are still in the serial pipeline, an initialization character '<' is sent across the serial port, followed by the path slope angle and then a closing character '>'. The opening and closing characters make it easier for Uno 1 to know when the number has been completely sent so that no bytes are lost.

4.2.6 Receipt and Transmission of Path Slope Input from Uno 1 to Uno 2

With the path slope input sent to the HC-06 Bluetooth module, Uno 1 must receive the bytes from the serial port and pass them to Uno 2 for processing. This is accomplished with the Arduino sketch shown below in Code 4.8. This section of code does not include the transmission of speed

input to Uno 3, which will be discussed later. The full script is included in Appendix B.

Code 4.8: Uno 1: Serial receipt and transmission of path slope input to Uno 2

```
SoftwareSerial steering_connection(10, 11);// 10 = RX, 11 = TX

void setup ()
{
  Serial.begin (9600);
  steering_connection.begin(9600);

}

void writeInput ()
{
  byte c = Serial.read ();
  steering_connection.write(c);
}

void loop ()
{
  while(Serial.available ())
  {
    writeInput ();
  }
}
```

Since Uno 1 needs to receive serial information from one source and transmit serial information to another, a second serial port is needed in addition to the built in serial port on pins 0 and 1. The library `SoftwareSerial` allows this to be accomplished with code. This software serial port is initialized to pins 10 and 11, with pin 10 acting as the receiving pin and pin 11 the transmission pin. Both serial ports are initialized in the Uno `setup` function to a baud rate of 9600 bits per second, with the built-in serial port denoted as `Serial` and the software serial called `serial_connection`. It is important that the bit rate of these serial ports matches the bit rate of the Bluetooth transmission, or bytes of information will be lost. In the main Uno loop, the function `writeInput()` is called whenever any bytes are available in the `Serial` buffer. When `writeInput()` is called, the first available byte in the serial buffer is read from the buffer and then immediately written across the software serial port to Uno 2. This process is repeated as long as more bytes are in the buffer to assure that all path slope inputs are passed to Uno 2.

4.2.7 Writing Path Slope Input to Steering Servomotor

Uno 2 must perform the functions of receiving the incoming path input, decoding the 1 or 2-digit integer input, and passing that input to the steering servomotor. To ease in the decoding of the incoming integers, start and ending characters `'<'` and `'>'` are implemented to assist the decoder in recognizing when an integer is arriving and when it has been completely received.

Code 4.9: Uno 2: Path slope input receipt and decoding

```
const char startCharacter = '<';
const char endCharacter = '>';

void setup ()
 {
  Serial.begin (9600);
 }

void processInput ()
 {
  static long receivedNumber = 0;
  static boolean negative = false;
  byte c = Serial.read ();

  switch (c)
    {
     case endCharacter:
      if (negative)
        processNumber (- receivedNumber);
      else
        processNumber (receivedNumber);

     case startCharacter:
       receivedNumber = 0;
       negative = false;
       break;

     case '0' ... '9':
       receivedNumber *= 10;
       receivedNumber += c - '0';
       break;

     case '-':
       negative = true;
       break;
   }
 }

void loop ()
 {
  if (Serial.available ())
    processInput ();
 }
```

After initializing the serial port, the `processInput()` function is continuously called as

long as bytes are in the serial buffer. Each incoming byte is read, and the corresponding `switch` case is executed. If the start character `'<'` is received, the function variables are set to their default states. In the case that the received byte is an integer, the variable `receivedNumber` is multiplied by 10 to shift any previously received integers one spot to the left, and the current integer is added. If a negative sign is received, the `negative` state is set to true. Finally, once the close character is received, the `receivedNumber` is passed to the function `processNumber()`, which is discussed below.

With the received path input from the computer decoded, Uno 2 must perform the pure-pursuit path tracking algorithm that was discussed in Chapter 3.

Code 4.10: Uno 2: Pure-pursuit steering input calculation

```
void processNumber (const long n)
{
  thetap_rad = (n) * (pi/180);

  Yp = lane_width*sin((pi/2)-thetap_rad);

  Xp = camera_scan_distance - lane_width*cos((pi/2)-thetap_rad) +
     axle_to_camera_length;

  theta_deg = analogRead(feedbackPin);

  theta_mapped = map(theta_deg,0, 179,-90,90);

  theta_rad = theta_mapped*(pi/180);

  lookahead_angle_rad = (thetap_rad - theta_rad)/2;

  lookahead_distance = -(X*cos(thetap_rad) - Xp*cos(thetap_rad) +
     Y*sin(thetap_rad) - Yp*sin(thetap_rad))/(cos(lookahead_angle_rad
     - thetap_rad + theta_rad));

  curvature = (2*sin(lookahead_angle_rad))/lookahead_distance;

  delta_rad = atan(curvature*axle_distance);

  delta_deg = delta_rad*(180/pi);

  writeToServo = map(delta_deg,-90,90,0,180);

  steer.write(writeToServo);

}
```

The received path slope input is first converted from degrees to radians and then used to calculate the pursued point (X_p, Y_p). The current angular position of the servo is then read from the feedback

pin attached to pin A0 of Uno 2. Since this angle ranges from 0° to 180°, the angle is mapped to the -90° to 90° and then converted to radians for the calculation of the look ahead angle. The look ahead distance is then calculated from the previously calculated parameters. This in turn is used to calculate the curvature and subsequently the steering input to the servo, δ. Finally, this steering angle is converted to degrees, mapped back to the 0° to 180° space, and then written to the steering servo.

4.3 Sign Detection

The speed of the vehicle is controlled by the identification of speed limit signs and stop signs. As discussed in Section 1.4.2, the identification is accomplished with HOG that train an SVM . To perform these tasks, the SVM must first be trained, then the classifier must examine images to determine if the trained stop sign or speed-limit sign appears. Finally, the vehicle must adjust to the appropriate speed if a sign is detected.

4.3.1 Support-Vector-Machine Training

The first step in identifying street signs is training the SVM that makes the decision of whether or not a particular sign is present. In this case, the training is performed with a tool from the dlib open-source library called imglab. The imglab tool requires the acquisition of training images containing either a stop sign or speed-limit sign. Using more training images will help improve the accuracy of the classifier, but for this project, only 12 images of stop signs and 12 images of 40 mph speed limit signs are used. After placing the 10 images into a folder on the PC, the imglab tool is opened and the user interface appears as shown in Figure 4.8.

The files on the left are the images that will be used for training. To perform the training, the user draws boxes around the stop signs or speed-limit signs as is seen in Figure 4.8. It is important that the user consistently labels the objects in the image so that they will be properly classified for the SVM. After cycling through every training image, the training data will be saved as an extensible markup language (XML) file, which is used to create the SVM.

In Python, the SVM may now be created using the XML training data. This is performed with

Figure 4.8: User interface for the imglab tool used for training stop sign and speed limit sign SVM

the dlib function `train_simple_object_detector`, which takes the input XML file and outputs an SVM file that ultimately determines what constitutes an object as being present or not.

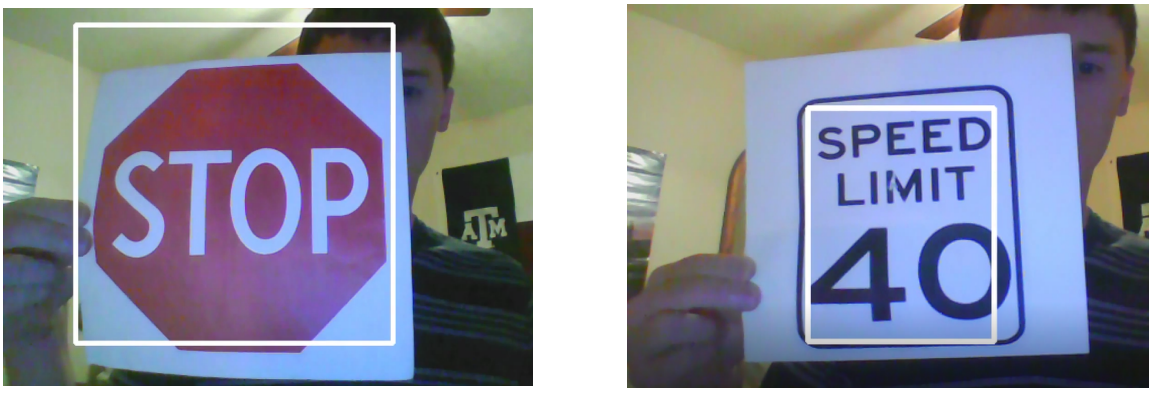
Code 4.11: PC: Training the simple object detector

```
import dlib

dlib.train_simple_object_detector(stop_sign_training.xml,
    "stop_sign_detector.svm")

dlib.train_simple_object_detector(speed_limit_40_training.xml,
    "speed_limit_40_detector.svm")
```

The training will take several minutes to perform depending on the number of training images used and the power of the computer processor. After the training is completed with the above script, these lines of code do not need to be run again unless the user wishes to add new training images to the original XML file. Otherwise, the output SVM file will be sufficient to perform the object detections. With the training performed, the accuracy of the trainer should be proven by providing the SVM file to the dlib `simple_object_detector` and demonstrating that the detector can detect the stop sign or speed limit sign from an image that was not used as training data as in Figure 4.9.



(a) (b)

Figure 4.9: Successful stop sign (a) and 40 mph speed limit (b) detection using an image that was not provided from the training data set

4.3.2 Street-Sign Recognition from Video Stream

Once the stream begins and the image has been captured by the PC, several important boolean conditions are set that determine whether or not the algorithm is in scanning mode and actively looking for street signs, or if an object has already been detected and is being tracked. The default state is setting `no_object_found = True` and `tracked_object = False`, meaning the algorithm scans for new detections until a sign has been identified. The tracker object is initialized for when an object has finally been detected. The sign recognition portion of the code begins scanning for both of the signs. The `toc` variable is updated for every iteration of the loop, and when there is a 1 s difference between the `tic` and `toc`, the sign detection is performed on whatever image arrives on that iteration. One second was chosen as the time between searches because it takes approximately 0.4 s for the SVM to scan the image for detections, and searching more frequently would cause delayed responsiveness for the navigation algorithm.

Code 4.12: PC: Street Sign Detection Algorithm

```
tracker = dlib.correlation_tracker()
no_object_found = True
tracked_object = False

tic = time.time()
try:
 while True:
      if no_object_found == True:

        toc = time.time()
```

```
if toc-tic > 1:
   tic = toc
   sl40dets = speed_limit_40_detector(cv_image)
   ssdets = stop_sign_detector(cv_image)

   for k, d in enumerate(sl40dets):
      print("Detection {}: Left: {} Top: {} Right: {} Bottom:
         {}".format(
         k, d.left(), d.top(), d.right(), d.bottom()))
      rectop = d.top()
      recleft = d.left()
      recbot = d.bottom()
      recright = d.right()
      no_object_found = False
      tracked_object = True
      tracker.start_track(cv_image,
         dlib.rectangle(recleft,rectop,recright,recbot))
      bluetooth.write(str.encode(str('b')))
      bluetooth.write(str.encode(str(40)))
      bluetooth.write(str.encode(str('e')))
      break

   for k, d in enumerate(ssdets):
      print("Detection {}: Left: {} Top: {} Right: {} Bottom:
         {}".format(
         k, d.left(), d.top(), d.right(), d.bottom()))
      rectop = d.top()
      recleft = d.left()
      recbot = d.bottom()
      recright = d.right()
      no_object_found = False
      tracked_object = True
      break
```

For each sign that is detected, a bounding rectangle is created in the area where the detection was made from the upper, lower, left, and right bounds. Now that a sign has been detected, `no_object_found` and `tracked_object` switch to `False` and `True`, respectively, so that the tracking portion of the code may begin. The bounding rectangle information is stored and later passed to the tracker so that it knows what object to begin tracking. The appropriate speed or slow signal is then passed to Uno 1 for eventual communication to Uno 3. The characters 'b' and 'e' for beginning and end are used to differentiate the speed-control information passed to Bluetooth from the '<' and '>' used to signal navigational information. The `while` loop is then broken and the tracking code begins.

4.3.3 Street-Sign Tracking

In order to begin tracking an object, the dlib tracker must be given the starting bounding rectangle of whatever object is to be tracked. Fortunately, this is provided whenever one of the signs is detected.

Code 4.13: PC: Street-Sign Tracking Algorithm

```
   while tracked_object == True:
      tracker.update(img)
      rect = tracker.get_position()
      pt1 = (int(rect.left()), int(rect.top()))
      pt2 = (int(rect.right()), int(rect.bottom()))
      cv2.rectangle(img, pt1, pt2, (255, 255, 255), 3)
      if rect.left() > 480 or rect.right() < 115 or rect.bottom() <
         100 or rect.top() > 390:
         tracked_object = False
         no_object_found = True
```

While the object remains on the screen, the tracker will continuously update the bounding boxes to follow the sign. This has the benefit of not incurring latency in the stream like the sign-detection algorithm causes, but has the downside of not allowing a previously unseen sign to be detected while another is on screen. The tracker will continue following the currently tracked sign or signs until the sign begins to move too far off screen on either the top, bottom, left, or right. At that point, even if some of the sign is still on the screen, the sign is considered to have passed and the code will revert to the sign scanning mode.

4.3.4 Receipt and Transmission of Speed Input from Uno 1 to Uno 3

After a sign has been detected and the proper speed-control signal is sent to Uno 1 via Bluetooth, that signal must be passed to Uno 3. This is accomplished in a very similar manner to the way path information is passed to Uno 2 for steering control.

Code 4.14: Uno 1: Serial receipt and transmission of speed input to Uno 3

```
#include "SoftwareSerial.h"

SoftwareSerial speed_connection(A0,A1); //A0 = RX, A1 = TX

int speed_index = 0;
char speed_buffer[4];

void setup ()
{
  speed_connection.begin(9600);
```

```
  pinMode(7,OUTPUT);
    digitalWrite(7,LOW);
}

void writeInput ()
{
  byte c = Serial.read ();

  if (c == 'b' || speed_index != 0)
  {
       speed_buffer[speed_index] = c;

       if (speed_index < 3)
       {
         speed_index += 1;
       }
       else
       {
         speed_connection.write(speed_buffer,4);
         speed_index = 0;
       }
  }
}

void loop ()
{
  if (Serial.available() > 0)
  {
    digitalWrite(7,HIGH);
    while(Serial.available ())
    {
      writeInput ();
    }
  }
}
```

A software serial connection is established on pins A0 and A1 of Uno 1 and initialized. One output pin is also declared on pin 7. In the main loop of the program, this pin writes `HIGH` as soon as any serial communication is made between the computer and the Bluetooth module to let Uno 3 know that navigation commands have been received and the vehicle may begin rolling forward. The `writeInput` is also called when serial bits are available. The major difference between the earlier steering control code is that this code first looks to determine if the first transmitted character is a 'b' for beginning. If so, the communication is intended for Uno 3, and the new speed is passed on. Since the speed is always a two digit integer, the program simply waits for the next three bits to arrive after the 'b' and then transmits all at once.

4.3.5 Controlling the Vehicle Speed

The vehicle speed is determined by one of three possible speed states. After serial communication is established between Uno 1 and the computer, pin 7 is written `HIGH` and the vehicle begins rolling at the default speed state of 0.45 m/s. Since the vehicle is controlled by an electronic speed controller, there is no use for any other feedback mechanism such as Hall effect or encoder to maintain drive speed. If a speed limit sign is detected, the vehicle enters the high speed state and the speed is increased to 0.67 m/s. Alternatively, if the stop sign is detected, the speed is set to the stop state of 0 m/s. The new PWM value is then transmitted to the ESC corresponding to the particular speed state.

Code 4.15: Uno 3: Controlling vehicle speed

```
#include <Servo.h>
Servo drive;

const char startOfNumberDelimiter = 'b';
const char endOfNumberDelimiter = 'e';

void setup()
{
  Serial.begin(9600);
  drive.attach(5);
}

int current_speed = 100;
int new_speed;


void loop()
{
  while(Serial.available ())
  {
    processInput ();
  }

  if(digitalRead(7) == HIGH)
  {
    drive.write(current_speed);
    delay(20);
  }

  Serial.println(current_speed);
}

void processNumber (const long n)
{
  new_speed = 2.6*n;
  current_speed = current_speed + (new_speed-current_speed);
}
```

5. EXPERIMENTAL SETUP AND RESULTS

This chapter presents the experimental setup and results. The first section presents the experimental environment and test track parameters. The methods for testing the navigational and sign detection accuracy are then presented, along with the results from each experiment.

5.1 Testing Environment

Figure 5.1 show a closed course create to test the ability of the vehicle to navigate autonomously. The course shown below is created with red duct tape, and measures 0.61 m across at all points on the track. The ratio between the vehicle width and the track is the same ratio as the average car and the width of American roadways. The two straightaway sections of the track each measure 2.7 m in length. The outer lane boundary for the 90° right-hand turns is made from a quarter-circle of radius 0.91 m, and the inner boundary a quarter-circle of radius 0.30 m. One speed limit sign is placed halfway down the first straightaway to increase the vehicle speed, and a stop sign is placed at the end to inform the vehicle that the track ends.

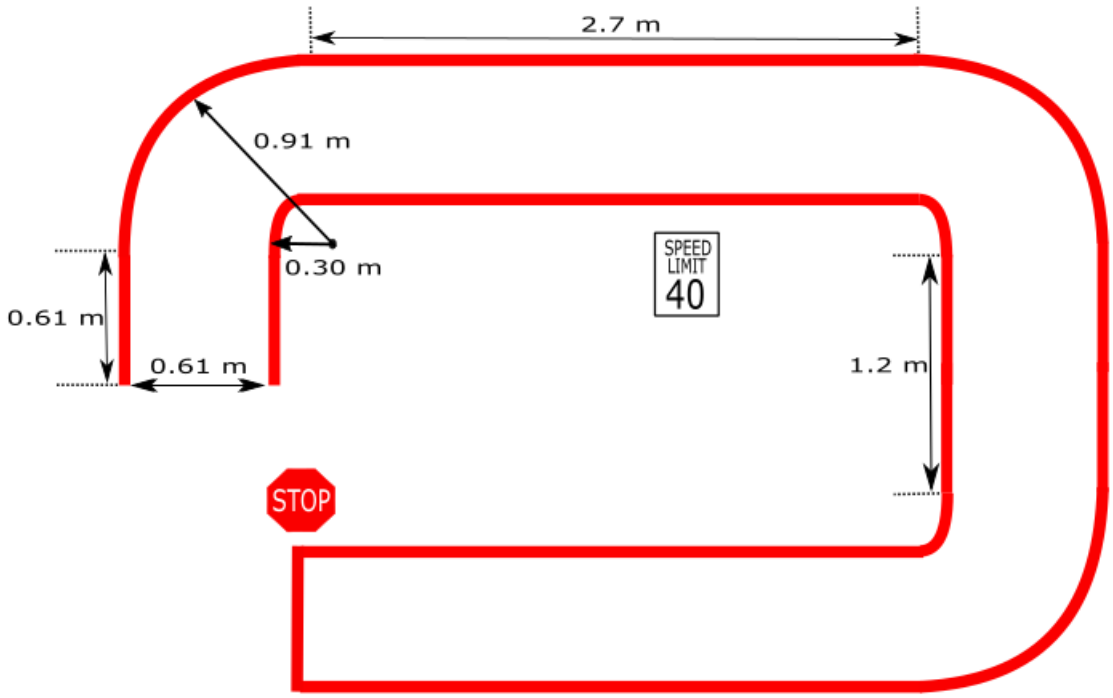


Figure 5.1: Closed track for vehicle navigation and speed testing

The sharp turns of the track are necessary to fit within the confines of the laboratory where the experiment is performed. The laptop that receives the video stream from the Raspberry Pi is located near the entrance of the track. At the start of testing, the vehicle is situated in the center of lane at the entrance to the track.

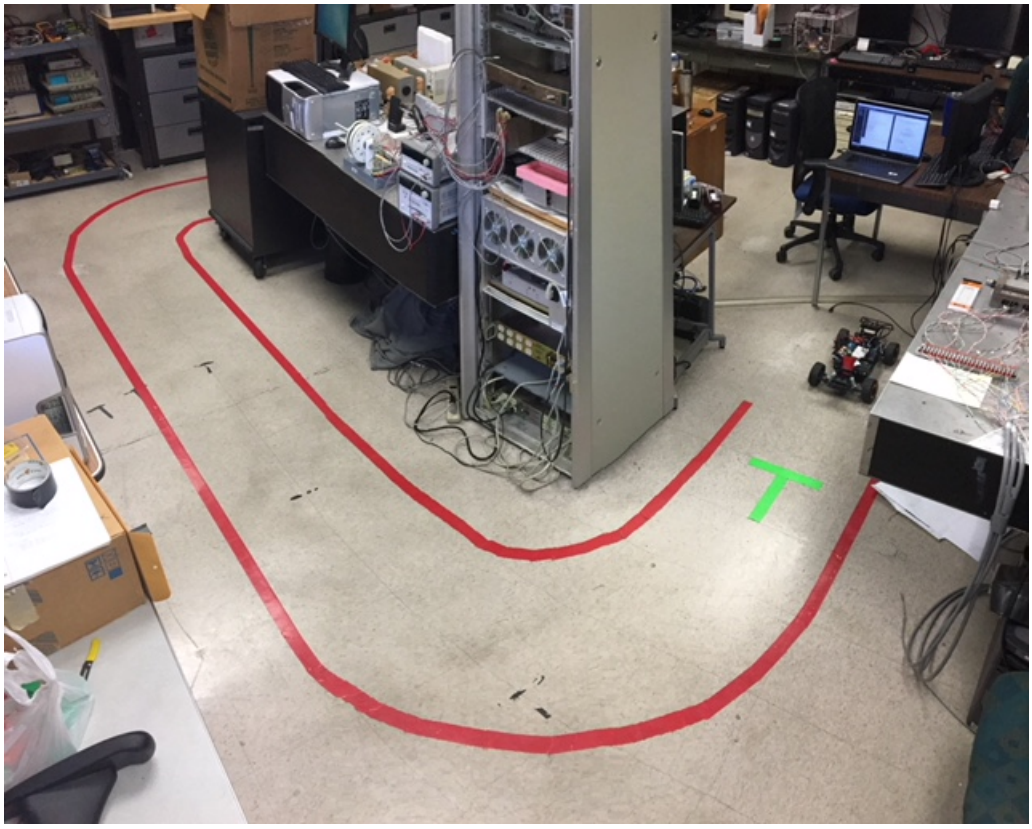


Figure 5.2: Testing track positioned within the laboratory environment, with laptop shown near the entrance to the track

5.2 Lane Navigation Testing Methods

After the vehicle is positioned at the entrance to the track, the Raspberry Pi and computer scripts are executed to commence the vehicle navigation test. Due to the stochastic nature of the WiFi system used for communication between the Raspberry Pi and computer, a test only qualifies as a proper test if the video stream averages at least 20 fps with no major interruptions during the test run. The HOG detection for signs is set to occur once per second during the navigation testing.

The accuracy of the navigation system is determined by how close the center of the rear axle

of the vehicle travels to the center line of the lane, since that is the point used for calculation of the pure pursuit steering angle. The run is recorded by four cameras positioned above the testing track, one at each turn and one along the first straightaway. The closeness to the center line is determined by noting the rear axle position every 10 frames of recorded film, which at 30 fps equates to 0.33 s per measurement, and then taking the normal from the reference path to the recorded point. The distance from the path is given in pixels, which is then converted to centimeters by measuring the number of pixels in a floor tile. In this case, each tile is exactly 30.48 cm.

5.3 Lane Navigation Testing Results

For the recorded run, the performance of the car throughout the first turn can be seen below in Figure 5.3. The footage was recorded with a GoPro camera attached to the 3.05 m ceiling and centered over the lane approximately 30 cm from the start.

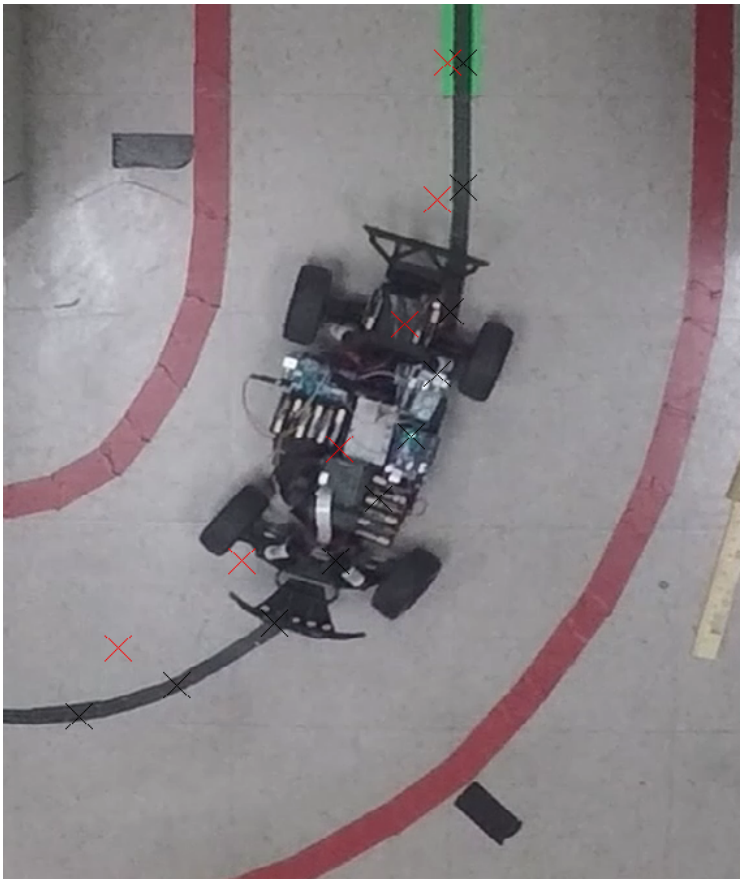


Figure 5.3: Vehicle navigating first turn with center line reference points (black) and actual travel points (red)

The vehicle begins tracking slightly inward from the beginning, which could be due to some misalignment in the starting position. The turn begins slightly too early, and as a result the vehicle deviates towards the inside lane line. The vehicle turning radius matches the radius of curvature of the turn as can be seen in the last three points in Figure 5.3. The greatest deviation the vehicle takes from the center line during this turn is 9.14 cm (3.6 in).

The second segment of footage is shown in Figure 5.4 from the straightaway, which was recorded with an iPhone 7 camera centered over the lane. The camera is again mounted at 3.05 m, and is 0.91 m from the beginning of the straightaway, denoted by the faint vertical line seen right in front of the vehicle in the image below.

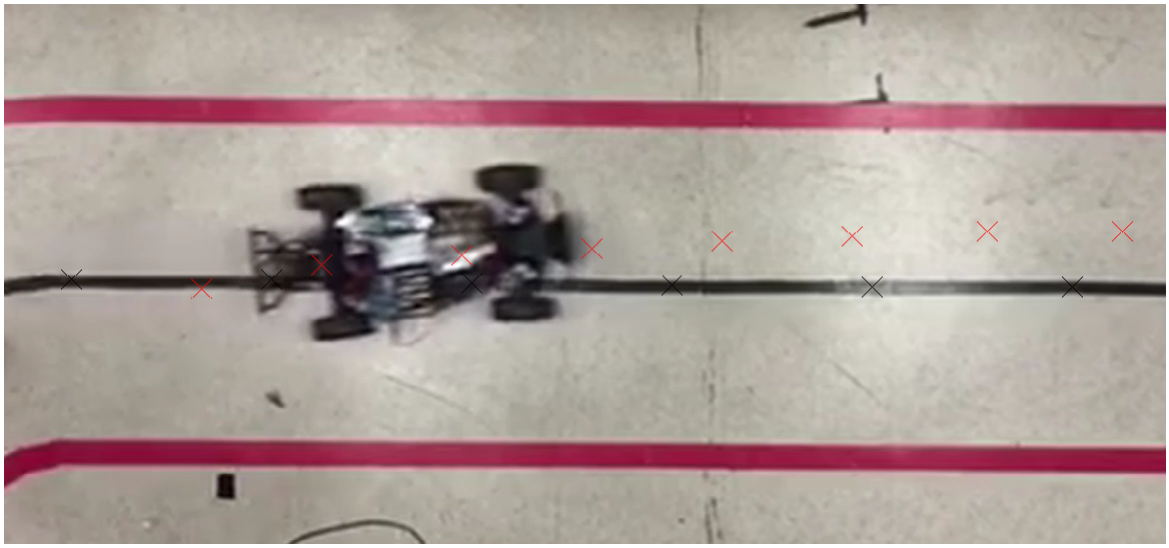


Figure 5.4: Vehicle navigating through the straightaway potion of the track with center line reference points (black) and actual travel points (red)

As the vehicle finishes the first turn and enters the straightaway, it quickly shallows the steering angle from the sharp first turn. The shallowing appears to occur slightly too quickly, resulting in the vehicle not being perfectly centered in the lane down the straightaway. As the steering angle reaches 0° in the straightaway, the center of the vehicle's rear axle is offset from the center line by 11.2 cm (4.4 in).

The second turn shown in Figure 5.5 is recorded from an iPhone 4 camera mounted 0.61 m

down the straightaway of the second turn and over the outside lane line. Like the cameras, it is also positioned 3.05 m from the floor.

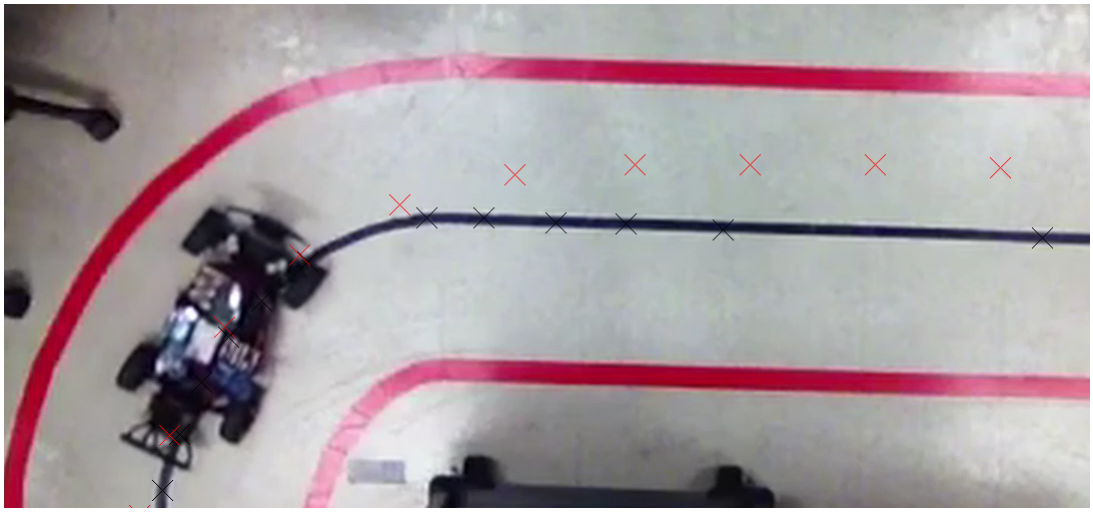


Figure 5.5: Vehicle navigating through the second turn and small straightaway of the track with center line reference points (black) and actual travel points (red)

The vehicle turning radius matches the second turn radius of curvature very well, deviating only 2.3 cm (0.87 in) for the first 4 points of Figure 5.5. As the vehicle comes out of the turn, it once again shallows from the sharp turn too quickly, giving a similar result on the short straightaway as was seen the the long straightaway. The offset from center as the steering angle goes to 0° is slightly larger for this turn, and the vehicle is offset from the center line by 15 cm (5.9 in).

The final turn is recorded by an iPhone 4 camera placed 3.05 meters above the floor and 0.61 m down the final straightaway. The positioning of the vehicle as it completes the final turn is shown in Figure 5.6. The vehicle recovers well from this offset as it enters the final turn. As in the first turn, the actual slope the vehicle takes relative to the path matches well, but once again the turn is taken slightly too early, resulting in deviation towards the inner lane boundary. The maximum deviation during the turn is 8 cm (3.15 in). Finally, as the vehicle comes out of the turn, it once again shallows too quickly and has an offset down the straightaway of 10 cm (3.9 in).

When all 37 collected data points are plotted along the vehicle track path along with the reference center line, the result is shown below in Figure 5.7. Overall, the vehicle achieved the goal

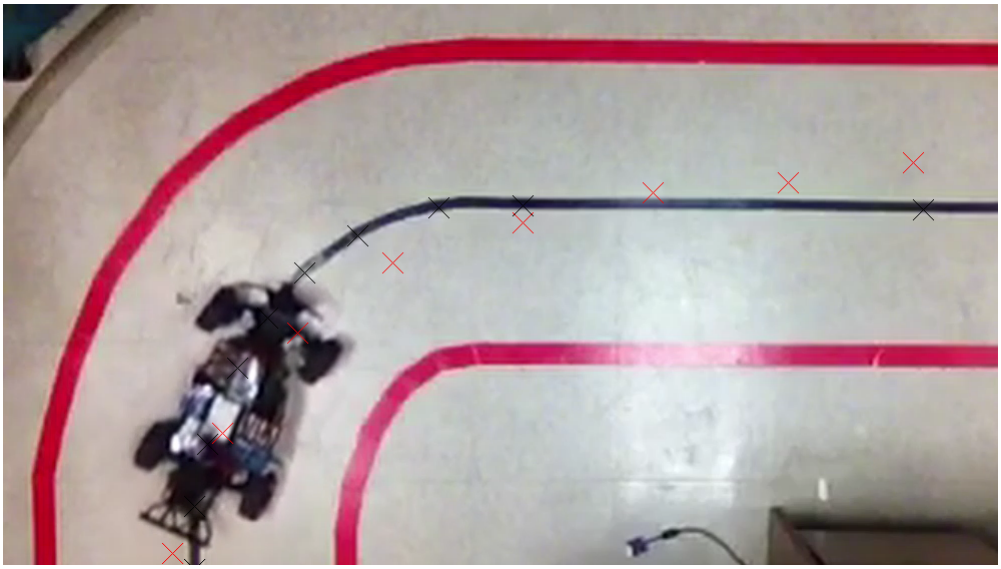


Figure 5.6: Vehicle navigating through the final turn and finishing straightaway of the track with center line reference points (black) and actual travel points (red)

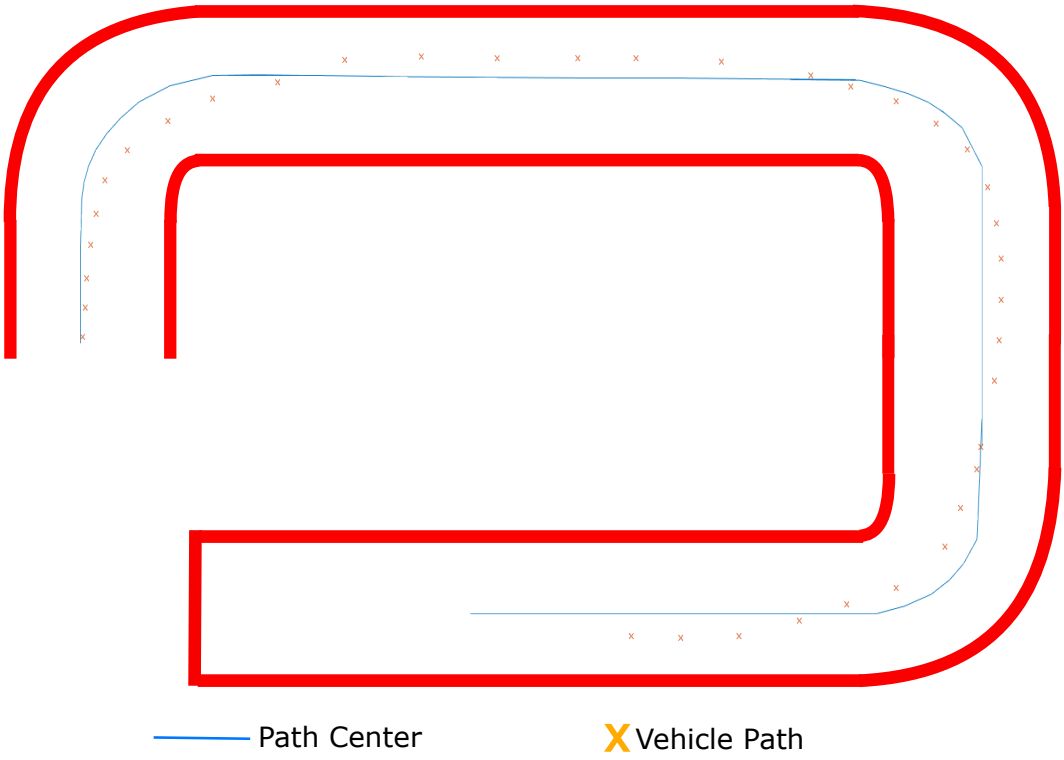


Figure 5.7: Complete vehicle plotted trajectory with reference centerline (blue) and collected data points (orange x)

of navigating the track successfully without accidentally departing the lane, which is the the first priority of the navigation controller. As for navigating as closely to the center line of the track as possible, the positional error of the vehicle relative to the center line is shown below in Figure 5.8.

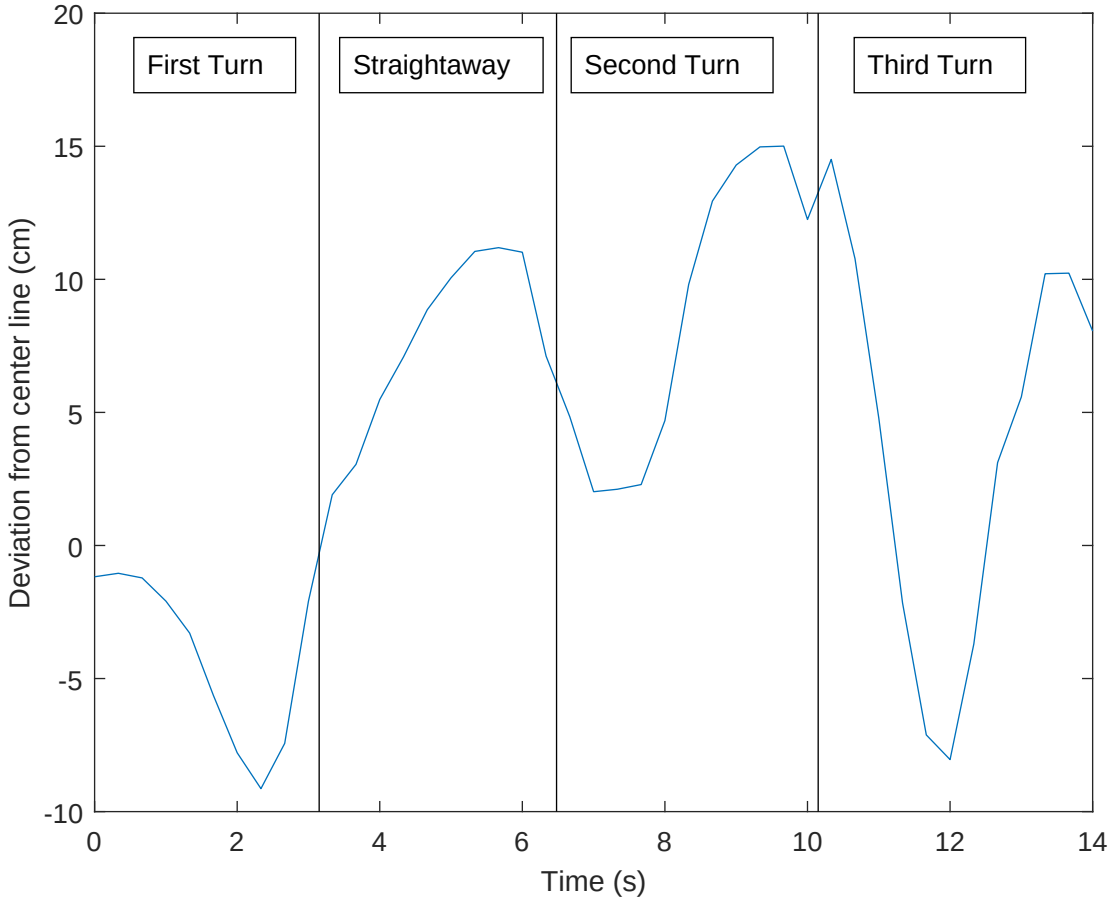


Figure 5.8: Vehicle navigational error as a function of time, with the four track segments noted.

The negative error indicates deviation towards the inner lane boundary, while positive error shows deviation towards the outer boundary. As can be seen from the error plot, the vehicle navigated through the turns very well, with the maximum error coming from the 9.14 cm deviation in the first turn. Coming out out the turns into the straightaway sections the vehicle did not do as well, as can be seen in the straightaway section of the plot and the second turn section between 8 and 10 seconds. In these sections the deviation was as much as 15 cm.

The larger error in the straightaways is likely due to the sharp difference in steering angle between the turn and straightaway. Because the vehicle looks 0.76 m ahead, the vehicle can see the straightaway while still in the middle of the sharp turn. The large difference in slope likely lead to the vehicle consistently shallowing out the steering angle too quickly so that the vehicle ended the turn not perfectly centered into the straightaway.

The implemented computer vision processes in this program drastically improve the rate of navigation inputs. With the navigation method presented in [1], it takes 200 ms to process one frame of video, meaning the program only processes 5 fps. The processes in this project decrease the processing time to 33 ms, meaning that it averaged 30 fps. The longest process of each frame iteration is in receiving the frame, which takes on average 17 ms. The masking and perspective transformations take the smallest amount of time, averaging 3 ms and 2 ms, respectively. Finally, the Hough transformation takes an average of 11 ms, bringing the total processing time to 33 ms.

5.4 Street-Sign Recognition Testing Methods

For the recognition of the street signs, the success of the system is determined first by whether the sign is successfully recognized, and second by how long the detection time takes. While the work in [1] performed recognition tasks while stationary and moving, the tests are all performed while moving since this was the area where the other method failed 40% of the time. Since attempting to perform the detection too many times per second can lead to latency in the navigation system, the detection is performed only once per second.

The vehicle begins with the sign out of view, and then rounds one of the track turns towards the sign. After the Raspberry Pi and laptop scripts are executed, the stream from the Raspberry Pi is recorded with a screen-recording software called OBS Studio. The video is later reviewed frame by frame, and the detection time is recorded as the time from when the street sign fully appears in frame as shown in Figure 5.9 to the time that the vehicle notes the detection and draws the bounding rectangle as in Figure 5.10.

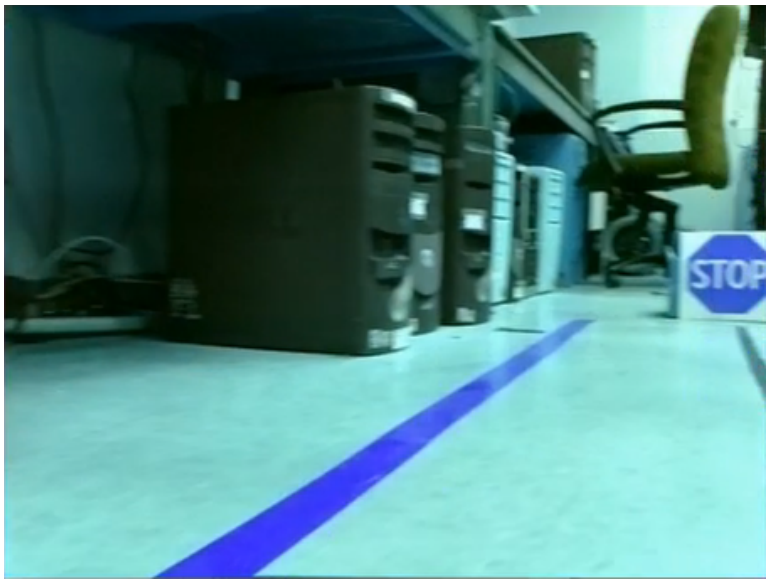


Figure 5.9: The frame where the stop sign first fully appears

Figure 5.10: The frame where the stop sign successful stop sign detection occurs and the bounding rectangle is drawn, similar to Figure 4.9

5.5 Street Sign Recognition Testing Results

The street-sign recognition test is performed 10 times for each sign. The results for the stop sign detection test are shown below in Figure 5.11. The stop sign detector achieves 100% accuracy

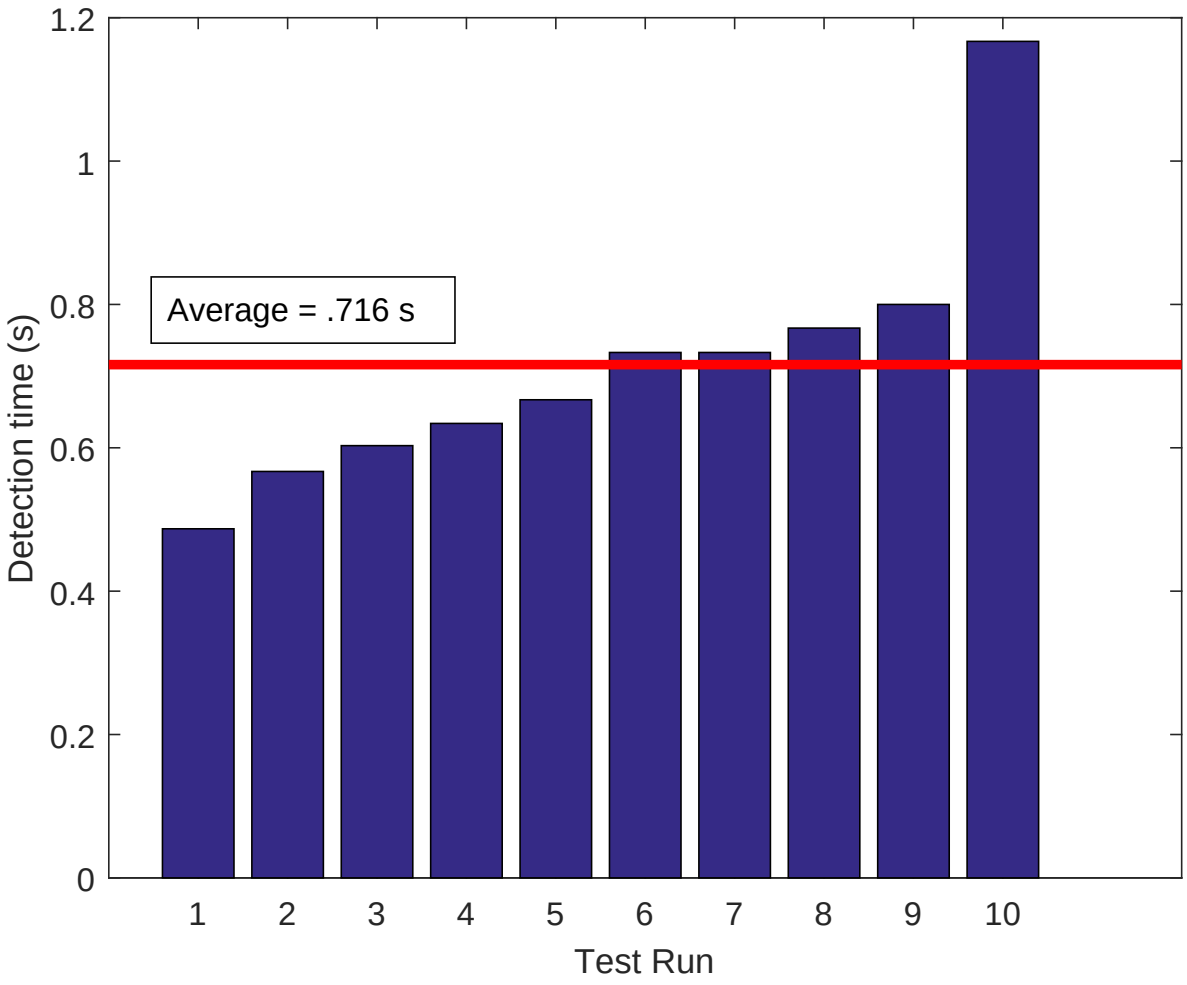


Figure 5.11: Stop sign detection time for 10 test runs

in identifying the stop sign from the streamed video while the vehicle was in motion. Additionally, the average detection time of 0.716 s is considerably faster than the 1.1 s for the 6 successful detections with the methods in [1]. However, it is slower than the 0.423 s that the detection would take if not for the 1 second delay between detections. The calculated sample standard distribution for the 10 tests is 0.1856 s, so assuming that these 10 samples are representative of the vehicle detection time if the test were conducted many times, it is expected that 95% of the detection times will fall within 0.716 ± 0.37 s.

The speed-limit sign detection test is performed similarly to the stop-sign detection test in that

the sign began out of camera view, but differs in that the sign was placed further down the track. The results are shown below in Figure 5.12. While the speed-limit sign detector also works 100%

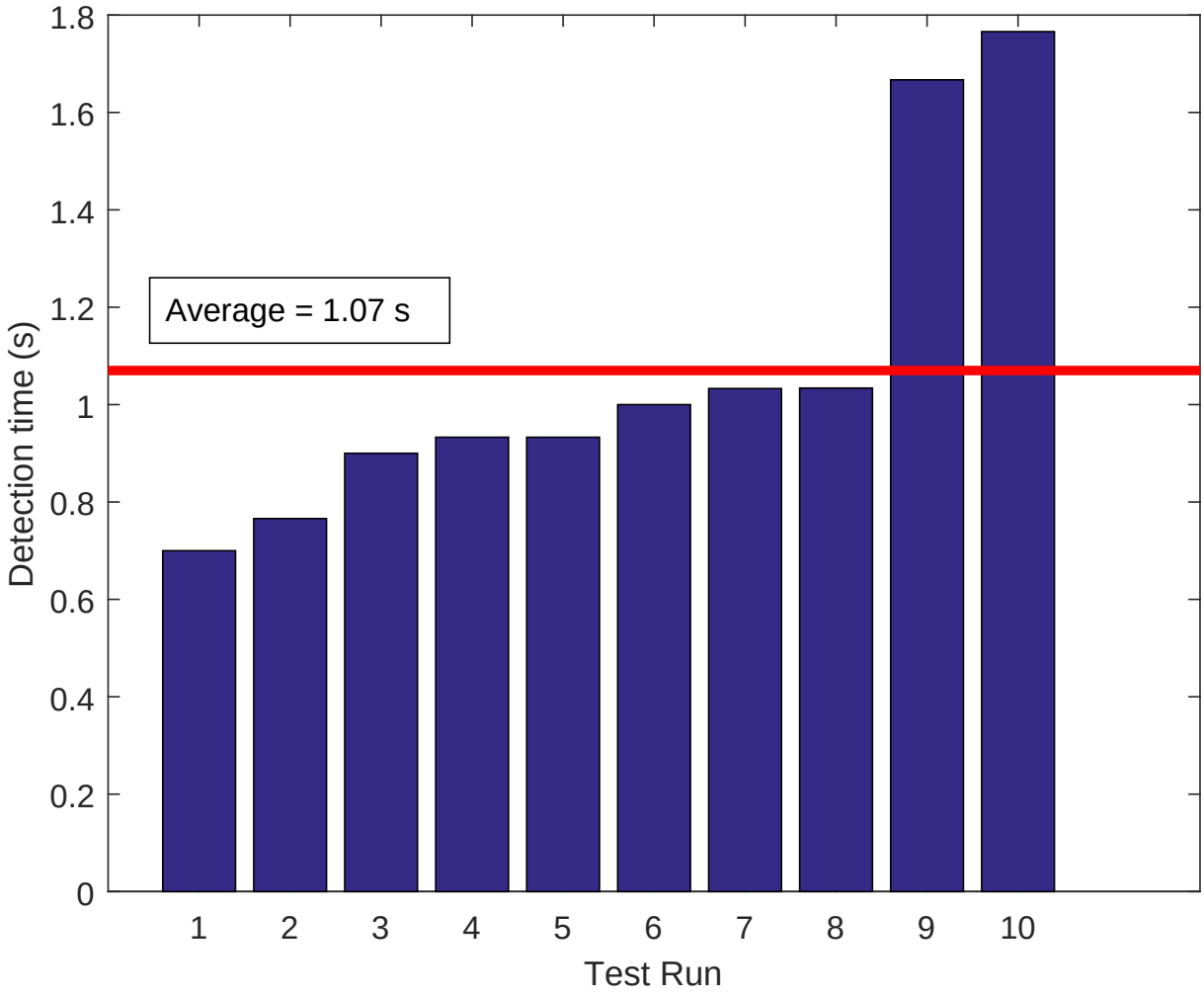


Figure 5.12: 40 mph speed-limit sign detection time for 10 test runs

in detecting the signs, the detection time is noticeably longer than that of the stop sign. The average detection time is 1.07 s, which is nearly identical to the detection time [1]. It is clear though from looking at the detection time chart that two of the tests took considerably longer than the other 8. The calculated sample standard deviation is 0.3565 for the 10 tests, meaning that 95% of the population tests should fall within 1.07 ± 0.71 s, which the two above average tests barely fall within. One possible explanation for the longer detection time might come from examination of

the trained HOG images themselves.

The HOG image created from the training stop sign images has very clear gradients outlining the sign and all of the letters as seen in Figure 5.13. This makes its vector field very distinctive, and therefore likely makes it easier for the SVM to detect when a stop sign appears in the streamed image. The HOG image created from the training speed limit signs is not as well defined as

Figure 5.13: Generated HOG image from stop sign training images

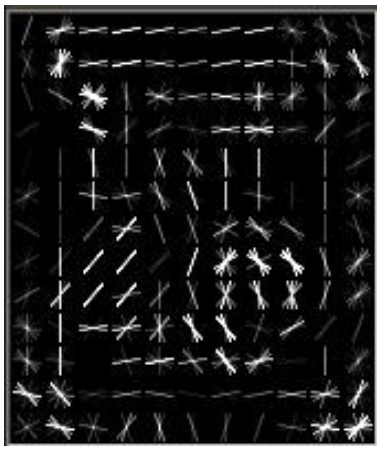


Figure 5.14: Generated HOG image from speed limit sign training images

the stop sign as seen in Figure 5.14. While the rectangular outline of the speed-limit sign and the number 40 can be seen from the gradient vectors, the size of the boxes used for finding the strongest gradients appears to be too large to clearly distinguish the words SPEED LIMIT. So while there are enough features for the detector to eventually pick out the sign, it seems like the vehicle needed to be closer to the sign before the image became clear enough to pick out the sign.

While the method of using a HOG detector coupled with SVM proved faster than the methods in [1], it was slower than some other methods. In [31], SVM coupled with HOG was also explored, and achieved a detection time of 0.54 seconds. The faster response of this HOG-SVM detector is likely due to 1 s delay needed to allow for navigation. However, this paper also presented methods of histogram projection coupled with SVM, histogram projection with multi-layer perception, and HOG with multi-layer perception which achieved detection times of 0.305 s, 0.495 s, and 0.475 s, respectively. If not for the delay to allow for the navigation algorithm, the normal detection time of 0.423 would be much closer to these values, so a method which avoids this delay is much more desirable.

6. CONCLUSIONS AND FUTURE WORK

The vehicle successfully achieved the project goals of adjusting speed and steering trajectory completely autonomously with computer vision and machine-learning. The HOG method of identifying street signs to control the speed of the vehicle proved to be very successful, and increased identification accuracy to 100% with a lower detection time of 0.716 s compared to the 1.1 s detection time of [1]. The steering controller provided a method of creating new trajectory points in real time so that pure-pursuit control could be implemented without the need of a manually pre-mapped route. While the controller was capable steering the vehicle well through turns, the shallowing of the steering angle exiting the turns caused offset down the straightaway sections as high as 15 cm. Implementation on more gradual curves such as those seen on highways and typical streets might provide more success than the sharp turns of the track.

The navigation methods proposed in this work have potential use as a backup to the navigation methods currently employed by autonomous vehicles. With most autonomous vehicles, road environments must be mapped with lidar by having human drivers first manually drive the road so the vehicle can learn the layout of the road. The navigation is then later conducted autonomously by having the vehicle use GPS to verify its relation to the previously mapped points. These methods can struggle when the road undergoes significant changes in layout, such as construction or road closure. The proposed method may be helpful in guiding the vehicle thorough this unexpected road change until the vehicle arrives back on road where the environment has been previously mapped.

The HOG method of identifying objects would be most applicable in processing route information a vehicle might encounter on the road. As has been demonstrated in this project, the HOG detector has near-perfect detection accuracy, and is very accurate when given proper training images. The HOG detector is probably most robust in identifying objects that have a very consistent shape, so route signs and warning signs that are nearly uniform nationwide would be most easily identified using the methods proposed in this work.

6.1 Future Work and Improvements

Even with the successful completion of the project goals, there are plenty of opportunities for future investigations and improvements. For the HOG detector, it would be interesting to see further development of the learning capabilities of the computer. Exploring the use of neural networks or other machine leaning algorithms could open up a host of interesting new applications. Future work in this area could also include adding more detected objects to the capabilities of the current HOG detector. One such example would be a HOG detector which would recognize the light status of an approaching traffic signal. Another future improvement is speeding the detection time. The largest improvement would come from eliminating the 1 s delay, which would reduce the detection time to 0.423 s. However, while 0.423 s would be acceptable for sign detection, in some applications like the detection of pedestrians this is still too long of a delay.

Additional improvements include reducing the latency and connection issues between the WiFi connection, computer, and Bluetooth receiver. While the project goals could have been accomplished with hardwire connections, an ideal autonomous vehicle would be capable of carrying all the necessary hardware on board. A larger vehicle or more powerful processor than the Raspberry Pi might solve these issues and make the use of wireless communication unnecessary. Other additions such as multiple Raspberry Pi controllers or multiple cameras could reduce the latency of the HOG detector on the navigation controller and perhaps provide additional visual feedback for improved steering capability.

REFERENCES

[1] F. B. Berg, "Lane keeping and pedestrian avoidance for a vision-based autonomous test vehicle," Master's thesis, Texas A&M University, College Station, TX, 2016.

[2] "Highway of the future," *Electronic Age*, vol. 17, no. 1, pp. 12–14, January 1958.

[3] D. A. Pomerleau, *Advances in neural information processing systems 1*. San Fransisco, CA: Morgan Kaufmann, 1989.

[4] *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. SAE Std. J3016, 2014.

[5] National Highway Traffic Safety Administration, "National motor vehicle crash causation survey," July 2008.

[6] J. He, H. Rong, J. Gong, and W. Huang, "A lane detection method for lane departure warning system," in *Proceedings of the 2010 International Conference on Optoelectronics and Image Processing*, vol. 1, November 2010, pp. 28–31.

[7] P. N. Bhujbal and S. P. Narote, "Lane departure warning system based on hough transform and euclidean distance," in *Proceedings of the 2015 Third International Conference on Image Information Processing (ICIIP)*, December 2015, pp. 370–373.

[8] Insurance Institute for Highway Safety, "Stay within the lines," *Status Report*, vol. 52, no. 6, pp. 2–4, August 2017.

[9] W. Pananurak, S. Thanok, and M. Parnichkun, "Adaptive cruise control for an intelligent vehicle," in *Proceedings of the 2008 IEEE International Conference on Robotics and Biomimetics*, February 2009, pp. 1794–1799.

[10] I. Fletcher, B. J. B. Arden, and C. S. Cox, "Automatic braking system control," in *Proceedings of the 2003 IEEE International Symposium on Intelligent Control*, October 2003, pp. 411–414.

[11] L. Xin, D. Bin, and H. Hangen, "Vision-based real-time pedestrian detection for autonomous vehicle," in *Proceedings of the 2007 IEEE International Conference on Vehicular Electronics and Safety*, December 2007, pp. 1–5.

[12] Wahyono, L. Kurnianggoro, J. Hariyono, and K. H. Jo, "Traffic sign recognition system for autonomous vehicle using cascade svm classifier," in *Proceedings of the IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, October 2014, pp. 4081–4086.

[13] M. Pontil and A. Verri, "Support vector machines for 3d object recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 6, pp. 637–646, June 1998.

[14] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, June 2005, pp. 886–893, vol. 1.

[15] H. Cho, P. E. Rybski, and W. Zhang, "Vision-based bicyclist detection and tracking for intelligent vehicles," in *Proceedings of the 2010 IEEE Intelligent Vehicles Symposium*, June 2010, pp. 454–461.

[16] L. Mao, M. Xie, Y. Huang, and Y. Zhang, "Preceding vehicle detection using histograms of oriented gradients," in *Proceedings of the 2010 International Conference on Communications, Circuits and Systems (ICCCAS)*, July 2010, pp. 354–358.

[17] Ntozis. (2010, September 23) Rgb cube. Licensed under CC BY-SA 3.0. [Online]. Available: https://commons.wikimedia.org/wiki/File:RGB_cube.jpg

[18] M. Horvath. (2015, December 28) Hsv color solid cylinder. Licensed under CC BY-SA 3.0. [Online]. Available: https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png

[19] R. Duda and P. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, January 1972.

[20] Arduino. Arduino Uno. [Online]. Available: https://store.arduino.cc/usa/arduino-uno-rev3

[21] Raspberry Pi Foundation. Raspberry Pi 3 Model B. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[22] Amazon. Arducam 5 Megapixels 1080p Sensor OV5647 Mini Camera Video Module for Raspberry Pi Model A/B/B+, Pi 2 and Raspberry Pi 3. [Online]. Available: https://www.amazon.com/gp/product/B012V1HEP4/ref=oh_aui_detailpage_o05_s03?ie=UTF8&psc=1

[23] ——. HC-06 Bluetooth Serial Pass-Through Module Wireless Serial Communication Compatible With Arduino. [Online]. Available: https://www.amazon.com/Pass-Through-Communication-Compatible-Atomic-Market/dp/B00TNOO438

[24] OpenCV release-3.0.0. [Online]. Available: https://opencv.org/

[25] Dlib release-19.4.99. [Online]. Available: http://dlib.net/

[26] J. M. Snider, "Automatic steering methods for autonomous automobile path tracking," April 2011.

[27] S. Mammar and D. Koenig, "Vehicle handling improvement by active steering," *Vehicle System Dynamics*, vol. 38, no. 3, pp. 211–242, 2002.

[28] H. Andersen, Z. J. Chong, Y. H. Eng, S. Pendleton, and M. H. Ang, "Geometric path tracking algorithm for autonomous driving in pedestrian environment," in *Proceedings of 2016 IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*, July 2016, pp. 1669–1674.

[29] D. Jones. (2017, February 25) Capturing to a network stream. PiCamera release-1.13. [Online]. Available: http://picamera.readthedocs.io/en/release-1.13/recipes1.html#streaming-capture

[30] ——. (2017, February 25) Rapid Capture and Streaming. PiCamera release-1.13. [Online]. Available: http://picamera.readthedocs.io/en/release-1.13/recipes2.html#rapid-capture-and-streaming

[31] A. Salhi, B. Minaoui, M. Fakir, H. Chakib, and H. Grimech, “Traffic signs recognition using hp and hog descriptors combined to mlp and svm classifiers,” *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 11, pp. 526–530, 2017.

APPENDIX A

COMPLETE PC CODE

```
import io
import socket
from socket import *
import struct
from PIL import Image
import numpy as np

import sys
import glob


import dlib
import cv2


import serial
import time

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import math
from skimage.transform import (hough_line, hough_line_peaks,
   probabilistic_hough_line)


server_socket = socket()
server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

options = dlib.simple_object_detector_training_options()

options.add_left_right_image_flips = False

options.C = 5

options.num_threads = 4
options.be_verbose = True


def hough_lines(img, rho, theta, threshold, min_line_len,
   max_line_gap):
  lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]),
     minLineLength=min_line_len, maxLineGap=max_line_gap)

  X1, Y1, X2, Y2 = create_navigation_path(scan_line,lines)

  global last_10_X1
  last_10_X1.append(X1)

  global last_10_Y1
  last_10_Y1.append(Y1)
```

```
    global last_10_X2
    last_10_X2.append(X2)

    global last_10_Y2
    last_10_Y2.append(Y2)

    img_shape = img.shape
    height, width = img_shape

    line_img = np.zeros(img_shape)

    draw_lines(line_img, lines)
    return line_img


def draw_lines(img, lines, color=[255, 0, 0], thickness=2):
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1,y1), (x2,y2), color, thickness)

def create_navigation_path(scan_line,lines):
    left_X1_sum = 0
    left_Y1_sum = 0
    left_X2_sum = 0
    left_Y2_sum = 0
    left_lines_detected = 0

    right_X1_sum = 0
    right_Y1_sum = 0
    right_X2_sum = 0
    right_Y2_sum = 0
    right_lines_detected = 0

    scan_line = (0,230,640,230) # X1, Y1, X2, Y2


    for line in lines:
        for x1,y1,x2,y2 in line:
            line = (x1,y1,x2,y2)
            if intersect(scan_line,line):
                if x1 <= 320:
                    left_X1_sum = left_X1_sum + x1
                    left_Y1_sum = left_Y1_sum + y1
                    left_X2_sum = left_X2_sum + x2
                    left_Y2_sum = left_Y2_sum + y2
                    left_lines_detected = left_lines_detected + 1
                else:
                    right_X1_sum = right_X1_sum + x1
                    right_Y1_sum = right_Y1_sum + y1
                    right_X2_sum = right_X2_sum + x2
                    right_Y2_sum = right_Y2_sum + y2
                    right_lines_detected = right_lines_detected + 1

    if left_lines_detected == 0 and right_lines_detected == 0:
        scan_line = (0,300,640,300)
        for line in lines:
            for x1,y1,x2,y2 in line:
                line = (x1,y1,x2,y2)
```

```
        if intersect(scan_line,line):
          if x1 <= 320:
            left_X1_sum = left_X1_sum + x1
            left_Y1_sum = left_Y1_sum + y1
            left_X2_sum = left_X2_sum + x2
            left_Y2_sum = left_Y2_sum + y2
            left_lines_detected = left_lines_detected + 1
          else:
            right_X1_sum = right_X1_sum + x1
            right_Y1_sum = right_Y1_sum + y1
            right_X2_sum = right_X2_sum + x2
            right_Y2_sum = right_Y2_sum + y2
            right_lines_detected = right_lines_detected + 1


if left_lines_detected > 0:
    left_avgX1 = left_X1_sum/left_lines_detected
    left_avgY1 = left_Y1_sum/left_lines_detected
    left_avgX2 = left_X2_sum/left_lines_detected
    left_avgY2 = left_Y2_sum/left_lines_detected

if right_lines_detected > 0:
    right_avgX1 = right_X1_sum/right_lines_detected
    right_avgY1 = right_Y1_sum/right_lines_detected
    right_avgX2 = right_X2_sum/right_lines_detected
    right_avgY2 = right_Y2_sum/right_lines_detected

if left_lines_detected == 0 and right_lines_detected == 0:
   left_avgX1 = 220
   left_avgY1 = 260
   left_avgX2 = 220
   left_avgY2 = 360

   right_avgX1 = 420
   right_avgY1 = 220
   right_avgX2 = 420
   right_avgY2 = 320


if left_lines_detected == 0 and right_lines_detected != 0:

   left_avgX1 = (right_X1_sum/right_lines_detected)-200
   left_avgY1 = right_Y1_sum/right_lines_detected
   left_avgX2 = (right_X2_sum/right_lines_detected)-200
   left_avgY2 = right_Y2_sum/right_lines_detected

if right_lines_detected == 0 and left_lines_detected != 0:

   right_avgX1 = (left_X1_sum/left_lines_detected) + 200
   right_avgY1 = left_Y1_sum/left_lines_detected
   right_avgX2 = (left_X2_sum/left_lines_detected) + 200
   right_avgY2 = left_Y2_sum/left_lines_detected

if left_avgY1 < left_avgY2:
   left_avgY1, left_avgY2 = left_avgY2, left_avgY1
   left_avgX1, left_avgX2 = left_avgX2, left_avgX1

if right_avgY1 < right_avgY2:
   right_avgY1, right_avgY2 = right_avgY2, right_avgY1
```

```
    right_avgX1, right_avgX2 = right_avgX2, right_avgX1

  avgX1 = (left_avgX1 + right_avgX1)/2
  avgY1 = (left_avgY1 + right_avgY1)/2
  avgX2 = (left_avgX2 + right_avgX2)/2
  avgY2 = (left_avgY2 + right_avgY2)/2

  return (avgX1,avgY1,avgX2,avgY2)

def intersect(line_one,line_two):
   X1,Y1,X2,Y2 = line_one
   X3,Y3,X4,Y4 = line_two

   I1 = [min(X1,X2), max(X1,X2)]
   I2 = [min(X3,X4), max(X3,X4)]

   Ia = [max( min(X1,X2), min(X3,X4) ),
    min( max(X1,X2), max(X3,X4) )]

   A1 = (Y1-Y2)/(X1-X2)
   A2 = (Y3-Y4)/(X3-X4)
   b1 = Y1-A1*X1
   b2 = Y3-A2*X3

   if (A1 == A2):
      return False;

   Xa = (b2 - b1) / (A1 - A2)

   if ( (Xa < max( min(X1,X2), min(X3,X4) )) or (Xa > min( max(X1,X2),
      max(X3,X4) )) ):
      return False;
   else:
      return True;

def average_10_frames(X1_list,Y1_list,X2_list,Y2_list):

#### Find average of last 10 X1 points
  if len(X1_list) > 10:  ### When we get to 11 points, delete the
     first point
    del X1_list[0]

  if len(X1_list) == 10:
    X1_list_average = sum(X1_list)/10

#### Find average of last 10 Y1 points
  if len(Y1_list) > 10:  ### When we get to 11 points, delete the
     first point
    del Y1_list[0]

  if len(Y1_list) == 10:
    Y1_list_average = sum(Y1_list)/10

#### Find average of last 10 X2 points
  if len(X2_list) > 10:  ### When we get to 11 points, delete the
     first point
    del X2_list[0]

  if len(X2_list) == 10:
```

```
    X2_list_average = sum(X2_list)/10

#### Find average of last 10 Y2 points
  if len(Y2_list) > 10:  ### When we get to 11 points, delete the
     first point
    del Y2_list[0]

  if len(Y2_list) == 10:
    Y2_list_average = sum(Y2_list)/10

  return (X1_list_average,Y1_list_average,
     X2_list_average,Y2_list_average)

def find_steering_angle(X1,Y1,X2,Y2):
  opposite = (X2-X1)
  adjacent = (Y1-Y2)
  rad_steering_angle = math.atan2(opposite,adjacent)
  deg_steering_angle = math.ceil(math.degrees(rad_steering_angle))
  return deg_steering_angle



speed_limit_40_detector =
   dlib.simple_object_detector("speed_limit_40_detector.svm")
stop_sign_detector =
   dlib.simple_object_detector("stop_sign_detector.svm")
tracker = dlib.correlation_tracker()
no_object_found = True
tracked_object = False
run = True
rectop = 0
recleft = 0
recbot = 0
recright = 0
i = 0

scan_line = (0,230,640,230) # X1, Y1, X2, Y2

last_10_X1 = []
last_10_Y1 = []
last_10_X2 = []
last_10_Y2 = []

connection = server_socket.accept()[0].makefile('rb')
port="/dev/rfcomm0"
bluetooth=serial.Serial(port, 9600)
bluetooth.flushInput()

tic = time.time()
try:
 while True:
   image_len = struct.unpack('<L',
      connection.read(struct.calcsize('<L')))[0]
   if not image_len:
      break

   image_stream = io.BytesIO()
   image_stream.write(connection.read(image_len))
```

```
image_stream.seek(0)
image = Image.open(image_stream)
cv_image = np.array(image)

hsv = cv2.cvtColor(cv_image, cv2.COLOR_RGB2HSV)

lower_red = np.array([0,50,50]) #0,100,100
upper_red = np.array([10,255,255])

mask = cv2.inRange(hsv, lower_red, upper_red)

cv2.imshow("mask",mask)

rho = 1
theta = np.pi/180
threshold = 20
min_line_len = 10
max_line_gap = 20

if i > 5:

   pts1 = np.float32([[15,395],[116,334],[529,331],[631,392]])
   pts2 = np.float32([[220,280],[220,180],[420,180],[420,280]])

   M = cv2.getPerspectiveTransform(pts1,pts2)

   dst = cv2.warpPerspective(mask,M,(640,480))
   cv2.imshow("Birdseye",dst)

   houged =
      hough_lines(dst,rho,theta,threshold,min_line_len,max_line_gap)

   if i >= 15:
    final_X1, final_Y1, final_X2, final_Y2 =
       average_10_frames(last_10_X1,last_10_Y1, last_10_X2,
       last_10_Y2)
    cv2.line(houged,(final_X1,final_Y1),(final_X2,final_Y2),(255,0,0),3)
    turn_angle =
       find_steering_angle(final_X1,final_Y1,final_X2,final_Y2)
    print(turn_angle)
    if i%1 == 0:
      bluetooth.write(str.encode(str('<')))
      bluetooth.write(str.encode(str((int(turn_angle)))))
      bluetooth.write(str.encode(str('>')))


   cv2.line(houged,(0,230),(640,230),(255,0,0),3) #search line

   cv2.circle(houged,(320,480),10,(255,0,0),-1) #Camera location
   cv2.imshow("Hough",houged)



i = i+1


if no_object_found == True:
   cv2.imshow('Steam',cv_image)
   cv2.waitKey(1)
```

```
        toc = time.time()
        if toc-tic > 1:
            tic = toc
            sl40dets = speed_limit_40_detector(cv_image)
            ssdets = stop_sign_detector(cv_image)

            for k, d in enumerate(sl40dets):
                print("Detection {}: Left: {} Top: {} Right: {} Bottom:
                    {}".format(
                    k, d.left(), d.top(), d.right(), d.bottom()))
                rectop = d.top()
                recleft = d.left()
                recbot = d.bottom()
                recright = d.right()
                no_object_found = False
                tracked_object = True
                tracker.start_track(cv_image,
                    dlib.rectangle(recleft,rectop,recright,recbot))
                bluetooth.write(str.encode(str('b')))
                bluetooth.write(str.encode(str(40)))
                bluetooth.write(str.encode(str('e')))
                break

            for k, d in enumerate(ssdets):
                print("Detection {}: Left: {} Top: {} Right: {} Bottom:
                    {}".format(
                    k, d.left(), d.top(), d.right(), d.bottom()))
                rectop = d.top()
                recleft = d.left()
                recbot = d.bottom()
                recright = d.right()
                no_object_found = False
                tracked_object = True
                tracker.start_track(cv_image,
                    dlib.rectangle(recleft,rectop,recright,recbot))
                bluetooth.write(str.encode(str('b')))
                bluetooth.write(str.encode(str(0)))
                bluetooth.write(str.encode(str(0)))
                bluetooth.write(str.encode(str('e')))
                break
    else:
        tracker.update(cv_image)

        rect = tracker.get_position()
        pt1 = (int(rect.left()), int(rect.top()))
        pt2 = (int(rect.right()), int(rect.bottom()))
        cv2.rectangle(cv_image, pt1, pt2, (255, 255, 255), 3)
        cv2.imshow('Steam',cv_image)
        cv2.waitKey(1)
        if rect.left() > 480 or rect.right() < 115 or rect.bottom() <
            100 or rect.top() > 390:
            tracked_object = False
            no_object_found = True
        if cv2.waitKey(1) == 27:
            break
finally:
    connection.close()
    server_socket.close()
```

APPENDIX B

ARDUINO CODE

Code B.1: Arduino 1: Bluetooth Arduino Complete Code

```
#include "SoftwareSerial.h"

SoftwareSerial steering_connection(10, 11);// 10 = RX, 11 = TX

SoftwareSerial speed_connection(A0,A1); //A0 = RX, A1 = TX

int speed_index = 0;

char speed_buffer[4];

void setup ()
{
  Serial.begin (9600);
  steering_connection.begin(9600);
  speed_connection.begin(9600);

  attachInterrupt(0, magnet_detect, RISING);

  pinMode(7,OUTPUT);
  digitalWrite(7,LOW);

}

void writeInput ()
{
  byte c = Serial.read ();

  if (c == 'b' || speed_index != 0)
  {
       speed_buffer[speed_index] = c;

       if (speed_index < 3)
       {
         speed_index += 1;
       }
       else
       {
         speed_connection.write(speed_buffer,4);
         speed_index = 0;
       }
  }

  else
  {
    steering_connection.write(c);
  }
}

void loop ()
```

```
{
  if (Serial.available() > 0)
  {
    digitalWrite(7,HIGH);
    while(Serial.available ())
    {
      writeInput ();
    }
  }

}
```

Code B.2: Arduino 2: Steering Control Arduino Complete Code

```
#include <Servo.h>     //Servo library
#include <math.h>


Servo steer;
int feedbackPin = A0;

const float pi = 3.14159;

const char startCharacter = '<';
const char endCharacter = '>';


float theta_deg;
float theta_rad;
float theta_mapped;

float thetap_deg;
float thetap_rad;

float lookahead_distance;
float axle_distance = .3302; //meters

float lookahead_angle_rad;
float lookahead_angle_deg;

float curvature;

float delta_rad;
float delta_deg;

float Yp;
float Xp;

float X = 0;
float Y = 0;

float lane_width = .3048;
float axle_to_camera_length = .2921;
float camera_scan_distance = .762;


int minDegrees = 60;
int maxDegrees = 120;
int minFeedback = 237;
int maxFeedback = 357;

const int numReadings = 10;

int readings[numReadings];
int readIndex = 0;
int total = 0;
int average = 0;
int currentPosition;

float writeToServo;
```

```
void setup()
{
  Serial.begin(9600);
  steer.attach(5);

  for (int thisReading = 0; thisReading < numReadings; thisReading++)
  {
    readings[thisReading] = 0;
  }

  steer.write(120);
  delay(1000);

  steer.write(90);
  delay(1000);
}

void loop()
{

  while(Serial.available ())
  {
    processInput ();
  }

}

void processNumber (const long n)
{
  Serial.println(n);
  thetap_rad = (n) * (pi/180);

  Yp = lane_width*sin((pi/2)-thetap_rad);

// Serial.print("Yp: ");
// Serial.println(Yp);

  Xp = camera_scan_distance - lane_width*cos((pi/2)-thetap_rad) +
     axle_to_camera_length;

// Serial.print("Xp: ");
// Serial.println(Xp);

  theta_deg = getPos(feedbackPin);

  Serial.print("currentTheta: ");
  Serial.println(theta_deg);

  theta_mapped = map(theta_deg,0, 179,-90,90);

  theta_rad = theta_mapped*(pi/180);

  lookahead_angle_rad = (thetap_rad - theta_rad)/2;

  lookahead_distance = -(X*cos(thetap_rad) - Xp*cos(thetap_rad) +
     Y*sin(thetap_rad) - Yp*sin(thetap_rad))/(cos(lookahead_angle_rad
     - thetap_rad + theta_rad));

  curvature = (2*sin(lookahead_angle_rad))/lookahead_distance;
```

```
  delta_rad = atan(curvature*axle_distance);

  delta_deg = delta_rad*(180/pi);

  writeToServo = map(delta_deg,-90,90,0,180);

  Serial.print("writeToServo: ");
  Serial.println(writeToServo);

  steer.write(writeToServo);

}

void processInput ()
 {
 static long receivedNumber = 0;
 static boolean negative = false;

 byte c = Serial.read ();

 switch (c)
   {

   case endCharacter:
     if (negative)
       processNumber (- receivedNumber);
     else
       processNumber (receivedNumber);

   // fall through to start a new number
   case startCharacter:
     receivedNumber = 0;
     negative = false;
     break;

   case '0' ... '9':
     receivedNumber *= 10;
     receivedNumber += c - '0';
     break;

   case '-':
     negative = true;
     break;

   }
 }

int getPos(int analogPin)
{
  currentPosition = map(analogRead(analogPin), minFeedback,
     maxFeedback, minDegrees, maxDegrees);
  total = total - readings[readIndex];
  readings[readIndex] = currentPosition;
  total = total + readings[readIndex];
  readIndex = readIndex + 1;

  if (readIndex >= numReadings)
  {
```

```
    readIndex = 0;
  }

  average = total / numReadings;
  return average;
}
```

Code B.3: Arduino 3: Speed Control Arduino Complete Code

```
#include <Servo.h>
Servo drive;

const char startCharacter = 'b';
const char endChacter = 'e';

void setup()
{
  Serial.begin(9600);
  drive.attach(5);
}

int current_speed = 100;
int new_speed;


void loop()
{
  while(Serial.available ())
  {
    processInput ();
  }

  if(digitalRead(7) == HIGH)
  {
    drive.write(current_speed);
    delay(20);
  }

  Serial.println(current_speed);
}

void processNumber (const long n)
{
  new_speed = 2.6*n;
  current_speed = current_speed + (new_speed-current_speed);
}

void processInput ()
 {
 static long receivedNumber = 0;
 static boolean negative = false;

 byte c = Serial.read ();

 switch (c)
   {

   case endChacter:
     if (negative)
       processNumber (- receivedNumber);
     else
       processNumber (receivedNumber);

   // fall through to start a new number
   case startCharacter:
```

```
    receivedNumber = 0;
    negative = false;
    break;

  case '0' ... '9':
    receivedNumber *= 10;
    receivedNumber += c - '0';
    break;

  case '-':
    negative = true;
    break;

   }
}
```