Building An Autonomous Car Fleet Using the Raspberry Pi

MCS Capstone Report

John Reid

March 24, 2017

Advisors: Prof. Simon Perrault, Prof. Robby Tan

Yale-NUS College

Abstract

Autonomous cars have the potential to transform our current transport systems, by navigating complex road environments more safely and efficiently than human drivers. This project focuses on understanding and implementing many of the same hardware, machine learning and computer vision techniques used in full-scale autonomous car navigation on miniature cars controlled by the Raspberry Pi computer. The outcome is to have two cars travelling in convoy around a track. Techniques implemented include lane detection, filtering, object avoidance, motor control and traffic sign recognition. A set of hardware instructions and code modules is provided in the report.

Author's Note

My personal goal for this project was to explore a wide range of technologies related to driving and navigation. The focus was on application and integration of different components rather than theory. As such, I have not made any strictly new theoretical contributions. Instead, I focused on getting a hardware prototype to work, and developing a reusable code base that others can use to further their own explorations. My primary resources for this project were online guides and tutorials rather than academic papers. I would often start with these, and then extend, repurpose or combine them. For example, I added Kalman filtering to my ultrasound component to get more accurate readings. In the process I have created a more well-rounded and fully-featured solution than any I have found using similar hardware.

Contents

Chapter 1

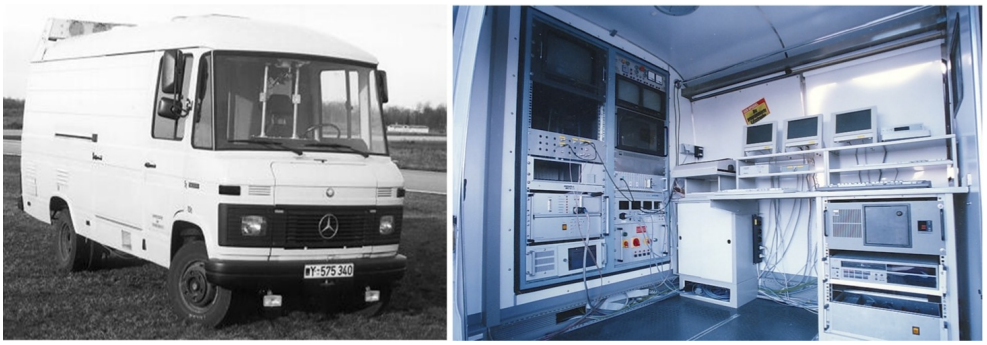
Introduction

1.1 A history of driverless cars

Autonomous vehicles have been around for much of the 20th century. Torpedoes were used in WW1, and German V2 rockets flew themselves over the English Channel during WW2. Yet driverless cars have taken a much longer time to develop, chiefly because they have to interact with a much more complex real-world environment of pedestrians, vehicles and varied terrains.

The start of modern driverless car technology is generally considered to be in 1986, when a white Mercedes-Benz van outfitted by Ernst Dickmanns and his team from Bundeswehr University Munich took to the roads in Bavaria without a driver at the wheel [15]. Previous effors had focused on building specialized tracks for driverless cars, but Dickmanns' team pioneered the use of computer vision for navigation, an approach also followed by this project.

Figure 1.1: Ernst Dickmann's Mercedes Van



Source: *www.dyna-vision.de*

Today's efforts build on Dickmanns' work, with the addition of lidar [1]. Google started its self-driving car program in 2009 [14], and Uber introduced a self-driving service in Pittsburgh in September 2016 [6]. While there are regulatory and safety concerns that still need to be addressed, it seems likely that in the near future our roads will be filled with autonomous vehicles.

There are many challenges to solve in building a driverless car - above all, a car should be able to sense and respond to its environment. With that in mind, most cars today employ a wide range of sensors and inputs, and try to construct as detailed a representation of their environment as possible.

1.2 The Raspberry Pi

This project aimed to explore some of the techniques involved in real-world autonomous driving using cheap, off-the-shelf components. With that in mind, the Raspberry Pi mini-computer was chosen as the platform, for three

[1] the use of lasers to measure distance

main reasons.

1. **Specifications** - With 1GB RAM and a 1.2gHz quad-core CPU, the Pi is about as powerful as a high-end mobile phone. Its operating system is a flavour of Linux, and it can run most general-purpose programming languages including Python and C/C++. This makes it much more capable than similar mini-computers like the Arduino.

2. **Open-source community** - Due to its popularity as a hobby kit, there are many tutorials online resources and libraries available. This project made extensive use of these, allowing much more to be accomplished than if it had begun from first principles.

3. **Hardware connections** - The Raspberry Pi sports 40 GPIO (General Purpose Input/Output) pins, allowing multiple sensors and devices to be attached. It can drive a 5V output through the pins, enough to power most small-scale electronics. In addition, a camera can be attached, which was used as the primary method of navigation.

1.3 Project Scope

The concrete objective of this project is to build two autonomous vehicles based on the Raspberry Pi, and have them navigate themselves around a track of the author's design.

The exact target has changed throughout the development of the project.

Initially, bluetooth beacons were going to be utilized to create an indoor navigation system, and swarm intelligence-based algorithms would be used for governing complex interactions between the cars. This proved to be too ambitious, and much more time was spent on the hardware and computer vision aspects of the project.

Although much of today's commercial software around driverless cars is proprietary, many of the techniques explored in this project (depth perception, neural networks, filtering) are being used in research projects [2]. One of the biggest differences is in the hardware - each of the cars costs around $150 to build, compared to around $8000 for an entry-level lidar kit alone[1].

1.4 Project Status

All the necessary modules have been written and partially tested, although the traffic sign detector is still failing to detect signs accurately. As a result, it is likely that the traffic sign detection and classification modules will not be used in the final implementation. The two cars have been built, however they require some of their motors to be replaced. This has prevented full-system testing and debugging. Thus, the project has not yet achieved its stated goal of having two cars drive around a track in convoy.

A demo will be ready for the final presentation in April. All code changes after the initial report submission will be made on a separate 'dev' branch

on the code repository.

Chapter 2

Hardware

The project required a lot of equipment and time to set up, and most of the first semester was taken up by building the cars. Ingmar Stapel's tutorial on building a remote-control car using the Raspberry Pi was used as a starting point[11].

2.1 Equipment

Most of the equipment was purchased online from two websites: element14, a Singaporean electronics supplier, and banggood.com, a Chinese e-commerce site. In some cases, more than three of each item was bought because it was cheaper to buy in bulk. Below find a list of the items purchased and used.

- 3 X Raspberry Pi Model 3B
- 3 X Raspberry Pi Camera v2
- 3 X Ni-MH AA battery pack

- 3 X Battery holder case
- 1 X 8-slot AA battery recharger
- 2 X 4WD Chassis with motors
- 1 X Female-Female Jumper Wire pack
- 2 X Xiaomi 10000 mAh Powerbank
- 3 X L298 H-Bridge Motor Driver Board
- 10 X SRF05 Ultrasound Sensor

The following items were also bought, but not used, due to changes in the objectives of the project:

- 3 X Bluetooth Low Energy beacon
- 10 X I2C Logic Level Converter
- 1 X Tamiya Gearbox

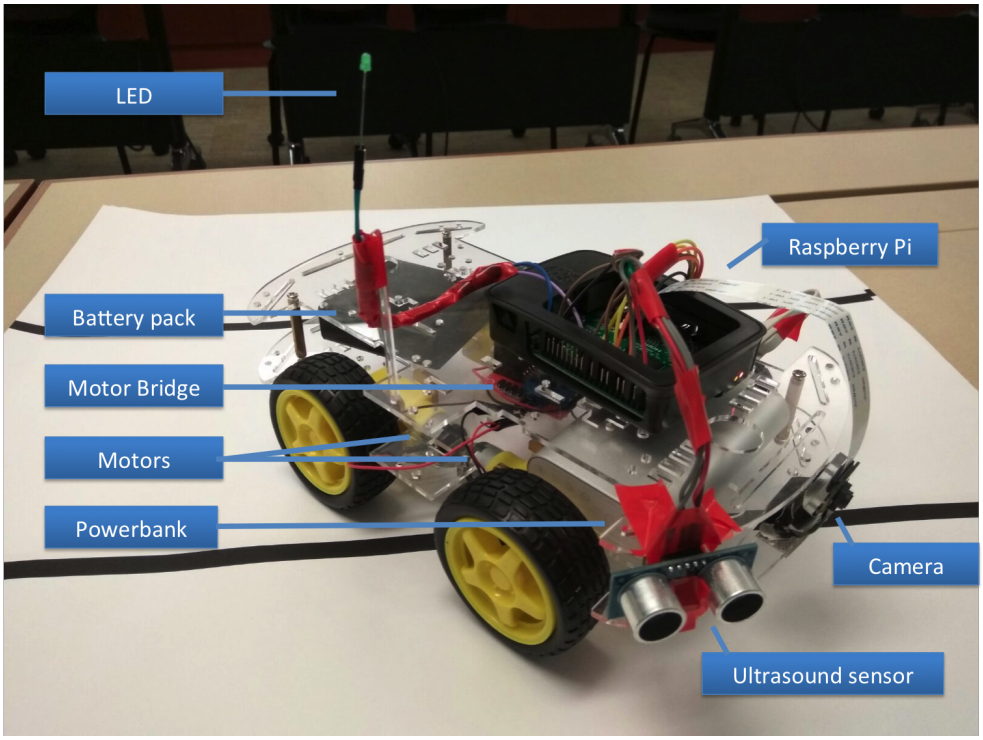
LEDs, resistors and soldering equipment were used from the Yale-NUS Fabrication Studio, where most of the work took place.

2.2 Setup

2.2.1 The Pi

The Pi can control external devices and sensors through its 40 GPIO pins. Through these pins, it can send and receive either 'high' or 'low' signals -

Figure 2.1: Setup of the car



essentially a binary communication protocol. To communicate intensity, the Pi can control the percentage of time an output is 'high' or 'low', otherwise known as the duty cycle [3]. This allows it to set the speed of the motors or the brightness of an LED, for example.

The Pi is powered by a 5V powerbank, of the type commonly used to recharge mobile phones. The motors require a separate power source, since they draw too much current away, and would cause the Pi to crash otherwise.

2.2.2 LED

An LED was included in each car, as it was a useful way to learn about physical computing, and could be used for debugging purposes. Each LED was connected and then to a ground and a 5V pin of the Pi. The resistance needed can be calculated as follows. The Raspberry Pi can output 5V through its GPIO pins, and a standard LED has a drop of about 1.7V. The current should be limited to around $10 \sim 20$ mAh so that it doesn't burn out the Pi. Therefore, Ohm's law can be used to calculate the resistor needed.

$$I = \frac{V}{R}$$

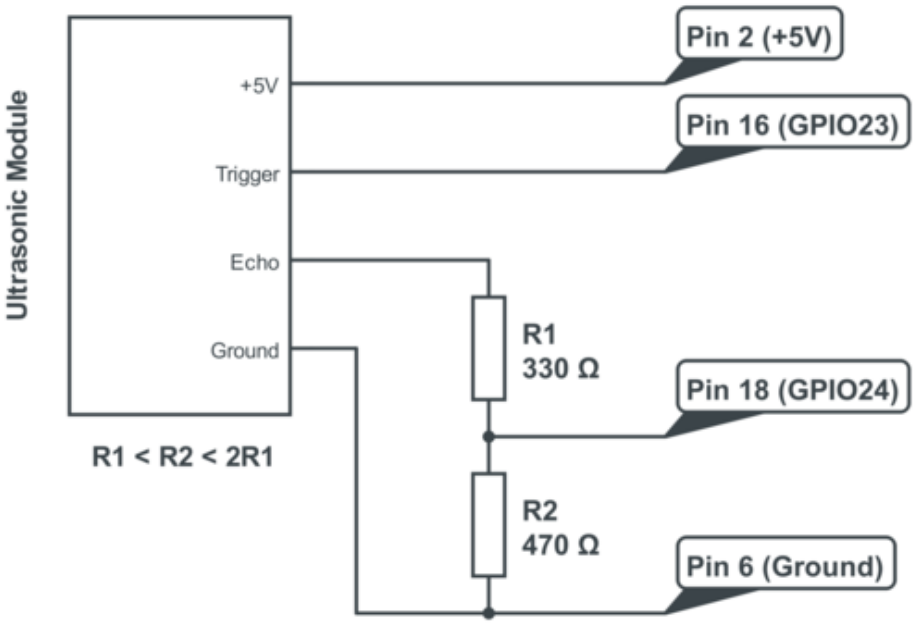
$$0.1 = \frac{3.3}{R}$$

$$R = 330\Omega$$

2.2.3 Ultrasound

SRF05 ultrasound sensors require some computation on the user's end to obtain actual distance measurements. As seen in Figure 2.2, apart from the ground and 5V pins, the sensor requires an input (trigger) pin, and an output (echo) pin. When the Pi takes a reading, it sends a signal through the input pin to the sensor. The sensor then sends out an ultrasonic pulse, and measures the time it takes for the pulse to return. The sensor then transmits a signal back to the Pi, and the duration of the signal is proportional to the distance measured.

Figure 2.2: Ultrasonic circuit



Source: *raspberrypi-spy.co.uk*

The two resistors act to reduce the input voltage to the ultrasound sensor, which can only handle $3.3V$ and below. The following rule for voltage dividers, based on Ohm's law, can be used to calculate the resistors needed.

$$V_{out} = V_{in}\frac{R_2}{R_1 + R_2}$$

$$3 = 5\frac{R_2}{R_1 + R_2}$$

$$R_2 = \frac{3}{2}R_1$$

Note that this gives a ratio, rather than an absolute value. For convenience however, this project used $R_1 = 330\Omega, R_2 = 470\Omega$.

2.2.4 Motors

There are four motors, two for each side. An L298 H-bridge connects the motors, a power source for the motors and the Pi. The H-bridge is essentially a complex switch that allows current to be driven across the motor in both directions [17].

Firstly, the motors on each side are connected to each other in parallel, and then to the H-Bridge. Thus, the motors on each side get the same amount of current, and behave in the same manner.

Secondly, the H-bridge is connected with a power source for the motors, and the Raspberry Pi for receiving signals. The H-bridge needs three inputs per side - two inputs to set the direction (forward/backwards), and one to set the speed of the motors.

Figure 2.3: H-bridge with motors

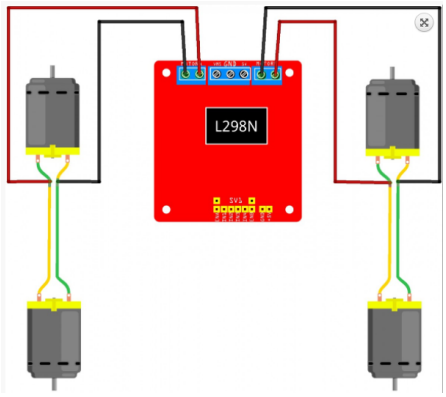
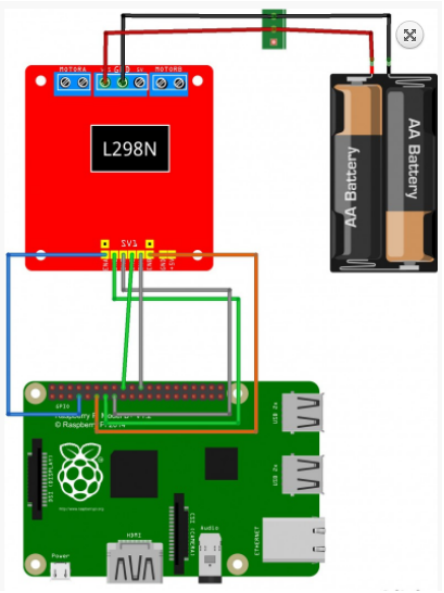


Figure 2.4: H-bridge with power source and Pi



Source: *custom-build-robots.com*

Chapter 3

Software

The software component took up most of the second semester. All of the code is available online as a github repo at https://github.com/johnmicahreid/capstone, and a significant goal for the project is for others to be able to build on the work contained there, as this project built on the work of many others.

3.1 Development Environment

3.1.1 Tools and libraries

Below find a list of the software tools, libraries and programs used in this project.

- Environment
  - Python 2.7 and Python 3.5 (programming language)
  - brew, apt-get, conda and pip (package managers)

  - Raspbian Jessie and macOS Sierra (operating systems)
  - Git and Github (version control system)
  - Jupyter Notebook (development environment)
  - Nano (text editor)
  - SSH and VNC (communication protocols)
- Libraries
  - OpenCV (computer vision library)
  - Scikit-learn (machine learning library)
  - Tensorflow (machine learning library)

3.1.2 Design choices

Python was chosen as the development language as the author was already familiar with it, it is fast to program in, and has many libraries available for the tasks needed. It runs slower than compiled languages like C or C++, and this performance bottleneck significantly impacted the functionality of the car. However, considering that this was mostly an exploratory and relatively small-scale project, the choice can be justified.

The code and development time was split roughly evenly between a MacBook Air and the Pi. Testing, debugging and hardware-specific code was

mostly done on the Pi, while machine learning, computer vision and data processing was done on the Air. Neither the Pi or the Air is able to run all of the available code. The code follows the Object-Oriented Programming paradigm, as each component/sensor is a separate module which can be loaded and used separately.

There are three types of scripts in the software repo.

- **Test and debugging scripts** - These are generally small, standalone scripts intended to test specific components e.g. the camera, the ultrasound sensor. They are run from the command line of the Pi.
- **Modules** - These are Python classes that are loaded into the main script, and contain functions which the main loop will invoke.
- **Jupyter notebooks** - With a mixture of code, images/outputs and commentary, these notebooks are intended to help people understand how and why calculations are being performed. They are generally viewed in a desktop browser.

In addition, there is an images directory, for use in testing, as well as a cascade directory containing a Haar cascade classifier (see Traffic Sign Detection section).

3.2 Navigation

3.2.1 Ultrasound Distance Measurement

The ultrasound module is used to detect obstacles. As mentioned previously, the ultrasound sensor required some user computation. The input to the sensor was a trigger signal telling it to begin measuring. The output from the sensor was a long signal, the length of which was correlated to the distance measured. See the pseudocode implementation below.

```
SEND trigger signal to sensor


WAIT until echo signal from sensor starts
MEASURE start time
WAIT until echo signal from sensor stops
MEASURE stop time


SUBTRACT start from stop time
DIVIDE result by two
MULTIPLY result by scaling factor
```

The division by two accounts for the fact that the pulse travels to the object and back. The scaling factor converts the pulse duration (in milliseconds) to a distance (in centimetres).

The problem with this method is due to the hardware - since the Pi runs a full operating system with a scheduler, the time measurements are not guaranteed to be accurate. This means that the measurements tended to be quite noisy, a problem which will be tackled later using filtering.

3.2.2 Lane Detection

The primary method of navigation in this project is lane detection. The input is an RGB image from the Pi's camera and the output is an estimate of the car's position relative to the centre of the lane - the navigational 'error'. This information can be passed to the motor control module to determine how the car steers.

A low-level process involving line detection was used in this project. Alternative methods such as neural networks were ruled out due to lack of good training data and computational requirements.

There are multiple stages involved, as follows:

1. **Convert the image into an edge map** - This is a standard preprocessing step in Computer Vision, and a built-in method from OpenCV called Canny Edge Detection was used. In brief, Canny Edge Detection involves blurring the image to reduce noise, then computing the image gradients (changes in pixel intensities), which signal edges, and finally applying thinning and thresholding operations. It results in a black and white image of the edges in the image.

Figure 3.1: Input: Raw image

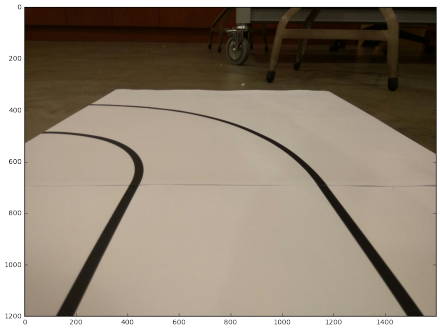


Figure 3.2: Canny Edge detector

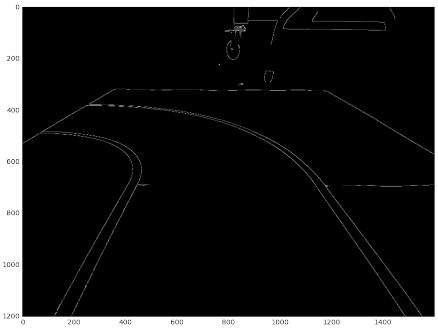


Figure 3.3: Hough Transform for Line Detection

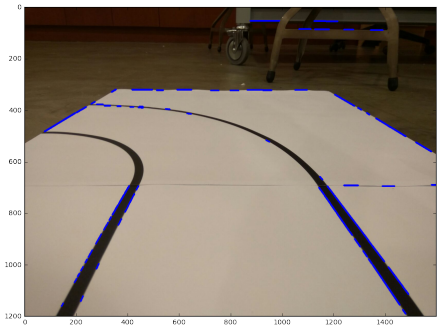


Figure 3.4: Clustering and Thresholding

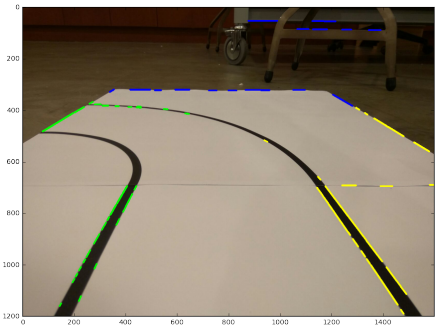


Figure 3.5: RANSAC

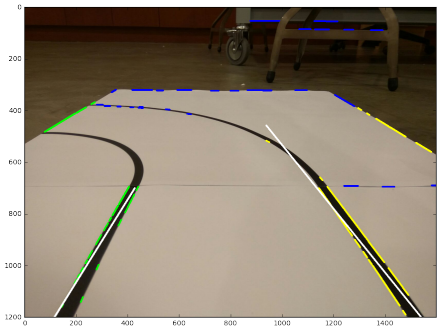
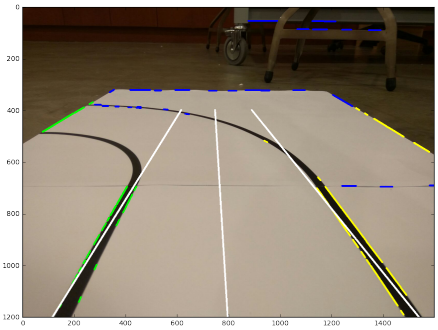


Figure 3.6: Output: Mid lane



2. **Detect the lines in the image** - Similar to the first step, this is a standard feature extraction operation which can be accomplished using the Hough Transform technique. This involves converting an image into parameter space (such that points become lines, and lines become points), and then finding local maxima in the parameter space [18]. It returns a list of line segments, which can be seen in Figure 3.13.

3. **Clustering and thresholding** - Before estimating the two lanes, the line segments need to be separated into left and right groups, depending on which side of the midline they are on. Next, the lines above the 'horizon' can be excluded, since the lanes will not appear above a certain threshold. This will make the subsequent estimation more accurate.

4. **Detecting the two lanes** - This is essentially a regression function, except with line segments rather than points. The problem is that there may be outliers in the data which would significantly skew the results for a simple linear regression. Instead, this project uses an algorithm called RANSAC, which stands for RANdom SAmple Consensus. This is an iterative method for calculating the parameters of a line that is robust against outliers. It works by repeatedly sampling different lines and calculating the proportion of 'inliers' for each - data points that lie sufficiently close to the sample line. While it is not guaranteed to converge, a good fit can usually be obtained after just a few samples

Figure 3.7: RANSAC sample - bad estimate

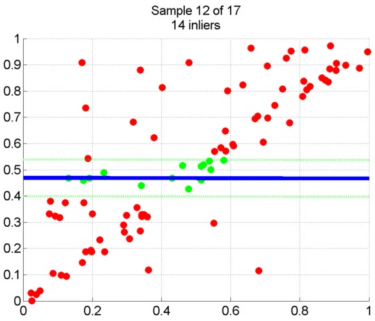
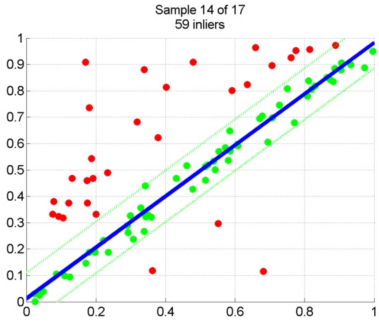


Figure 3.8: RANSAC sample - good estimate



Source: *danielwedge.com*

[19]. See Figures 3.7 and ransac2 for examples of good and bad fits.

5. **Compute the mid lane** - This is a fairly simple geometrical operation. Given the equations of two lines $y = m_l \cdot x + c_l, y = m_r \cdot x + c_r$, find the mid-line $y = m_m \cdot x + c_m$.

   This is done by calculating the intersection point of the two lines x_p, y_p, then calculating the gradient m_m as an average of m_l and m_r. Finally, the intercept of the mid-line c_m is calculated by substituting in x_p, y_p into the equation $y = m_m \cdot x + c_m$, as shown below

$$x_p = \frac{c_r - c_l}{m_l - m_r}$$

$$y_p = m_l \cdot x_p + c_l$$

$$m_m = \tan\left(\frac{\tan^{-1}(m_l) + \tan^{-1}(m_r)}{2} - \frac{\pi}{2}\right)$$

$$c_m = y_p - m_m \cdot x_p$$

   In the third equation, the averaging must be done in trigonometric form, and due to a quirk in the co-ordinate system of images (the

origin is in the top left, rather than the bottom left as in mathematics), the correct solution is at right angles to the simple average.

Now the offset or error from the car to the midline can be calculated. The position of the car is assumed to be in the horizontal center of the frame - half of the image width or $I_w/2$. On the bottom of the frame I_h, the distance from the midline to the car position gives the error at time t, which can be written as

$$e(t) = I_w/2 - (m_m * I_h + c_m)$$

This method works quite well at obtaining the error when the lines are straight, but breaks down somewhat for curves. It still needs some filtering to remove noisy measurements.

3.2.3 Filtering

Both the ultrasound measurement and the mid-lane computation involve noisy, imperfect estimates. This is a problem, since this information will be used to make navigational decisions. A solution is needed to smooth out the noise while still accurately estimating the underlying quantities.

This project uses Kalman Filtering to tackle these problems. Kalman Filtering is an iterative mathematical method that uses a set of equations and consecutive data inputs to estimate the true value of the quantity being measured, when the measured values contain errors and uncertainties [12]. It is

a special case of Bayesian filtering which assumes variables are linear and noise is normally distributed, which is reasonable in our case. Kalman filters are preferable to more general methods such as particle filtering, since they run faster and are easier to debug.

While a full treatment of Kalman filters is outside the scope of this report, the implementation used is worth explaining. Similar to a Hidden Markov Model, the state of the system x, at a time t depends only on the previous state and a measured data value. In this case, the state is a 2D data value representing position and velocity of the object relative the ultrasound sensor (or position and velocity of the car relative to the mid-lane). There are two main stages, as represented by the equations below.

$$\hat{x}_{t|t-1} = F\hat{x}_{t-1|t-1} + w \tag{3.1}$$

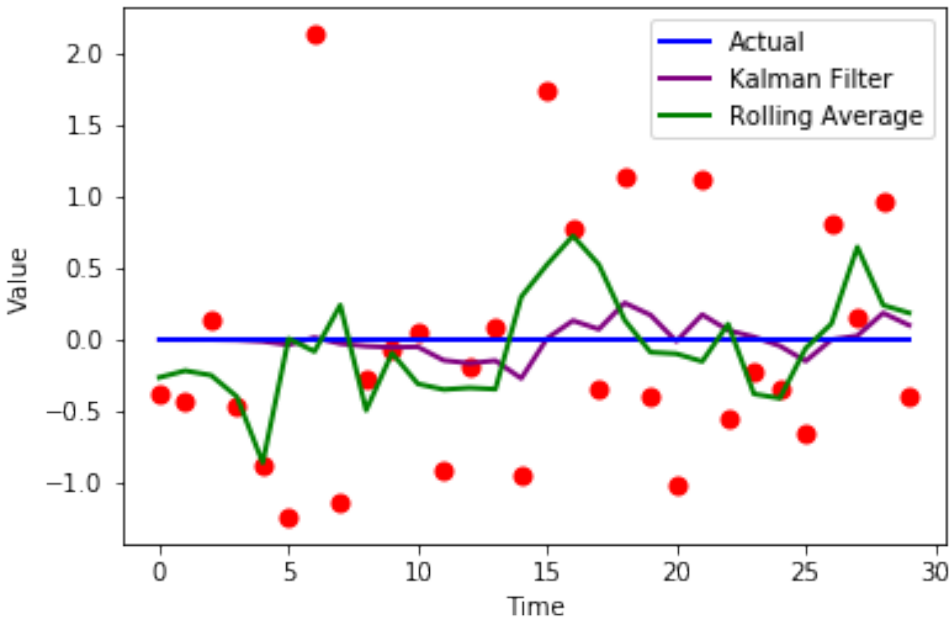
In this stage, a prediction of the state at time t is made, based on the previous state. Since this is a conditional probability, the state can be written as $\hat{x}_{t|t-1}$. F is known as the transition matrix, and it specifies the physical laws by which we expect the state to change. Since the variables are assumed to be changing linearly, a linear transition matrix is used. w is the process noise covariance matrix, and it is a measure of how much noise to expect in the process. This is an important parameter which governs how closely the model will stick to the data versus following the physical laws.

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t(z_t - \hat{x}_{t|t-1}) \tag{3.2}$$

In the second stage, the predicted state is updated by a new data value z_t. K_t is known as the Kalman Gain, and it can be thought of as a weighting factor, deciding how much importance to give to the prediction or the measurement. It is calculated automatically from the errors in the measurement and prediction.

As seen in Figure 3.2.3, a Kalman filter can be much more accurate at estimating the true line given data with Gaussian noise than a simple rolling average.

Figure 3.9: Kalman Filter to smooth out Gaussian noise



3.2.4 Motor Control

The Raspberry Pi can control the speed and direction of each set of motors independently. This makes it simple to define functions for moving left, right, forward, backward, accelerating and decelerating.

For the main navigational task, a proportional steering module was implemented. This is a simplified version of a proportional–integral–derivative controller (PID controller), which is a control loop feedback mechanism used in many industrial applications.

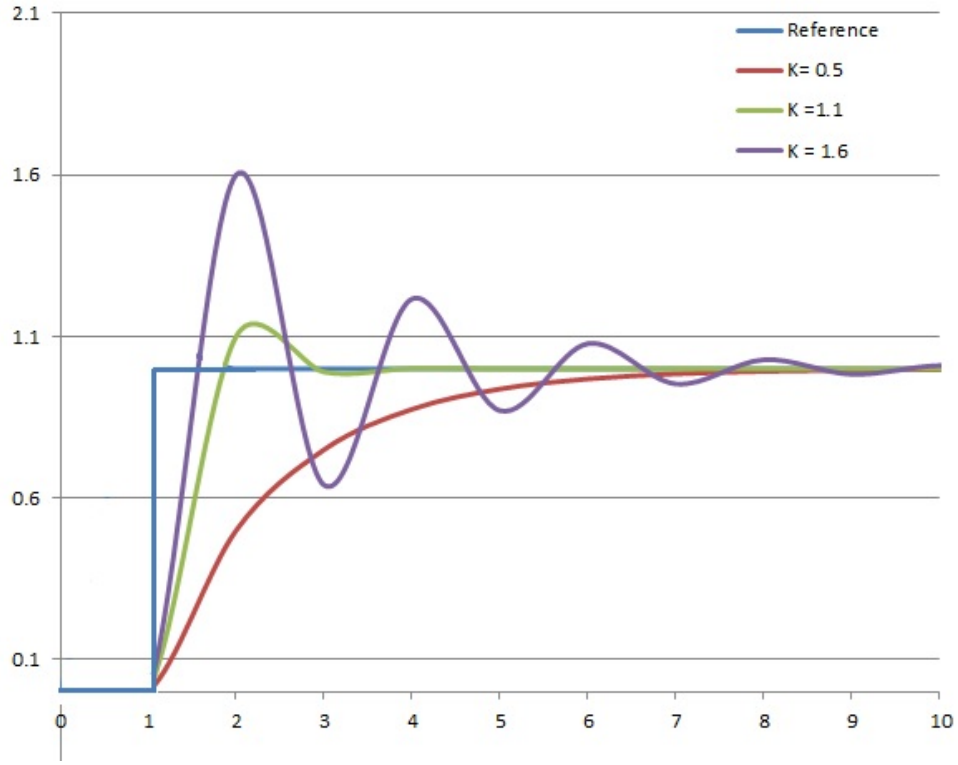
The equation governing the controller is as follows:

$$P_{out} = K_p e(t)$$

Here, P_out is the desired system response (steering angle in this case). It is a result of the error term $e(t)$, which is calculated as the offset of the car from the desired position, multiplied by the proportional gain constant K_p. The effect of this is that if the car is far out of position, it will steer more aggressively back towards the correct position, and taper off as it gets closer. The proportional gain constant must be chosen carefully - too high a value will cause the car to oscillate around the desired position, and too low a value will make the convergence very slow. In Figure 3.2.4, the straight line is the true value, while $K_p = 1.1$ causes the controller to converge the quickest.

Preliminary tests with the motor controller worked well, however hardware problems have prevented more extensive testing and tuning of the parameter.

Figure 3.10: Effect of different proportional gain constants on convergence



Source: *wikipedia.org*

3.2.5 Object Tracking

Given two cars in convoy, how should the second car navigate? The simplest method is to implement a kind of object tracking, where the second car tries to keep the first car in the centre of its field of view. To do this, the second car needs to calculate its error relative to the first car. This is exactly the same output as the lane tracking problem, so the motor control code can be

reused here.

To simplify the problem, a yellow ball was attached to the centre of the back of the first car. The problem then reduces to detecting the centre of the ball, and calculating the offset of the ball in the x-direction from the current car. This can be done frame-by-frame, and doesn't require any past information.

OpenCV was used to to solve this problem, based on a tutorial by Adrian Rosebrock [7]. The steps are as follows:

1. **Preprocessing** - First, apply a Gaussian blur to reduce noise, and then convert the image to HSV color space. This is an alternative representation of colors in a cylindrical color space, and allows for easier manipulation and segmentation of colors.

2. **Detect the yellow regions** - Apply a binary mask based on yellow color thresholds. All yellow objects now appear as white, and everything else appears black. In addition, do some dilations and erosions to remove small amounts of noise.

3. **Find the centre of the ball** - Use the OpenCV function FindContours to get the shapes of the white regions, then select the largest shape. Use the OpenCV function MinEnclosingCircle to draw a circle that completely covers the shape. After that, extract the x and y co-ordinates of the centre, and subtract the x-coordinate from the midline to get the offset.

Figure 3.11: Input: Raw image

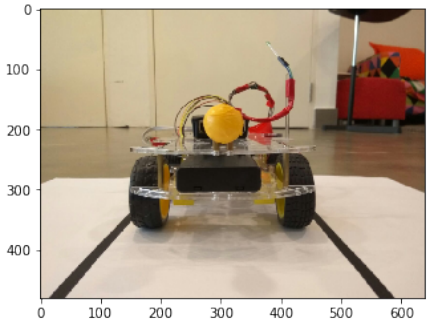


Figure 3.12: Binary Mask based on HSV yellow threshold

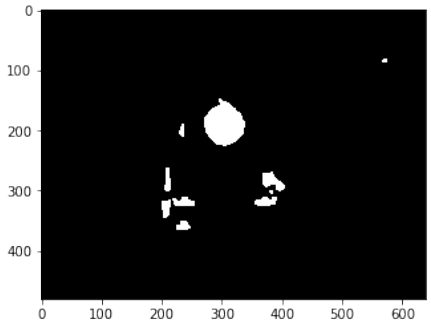


Figure 3.13: Output: small offset

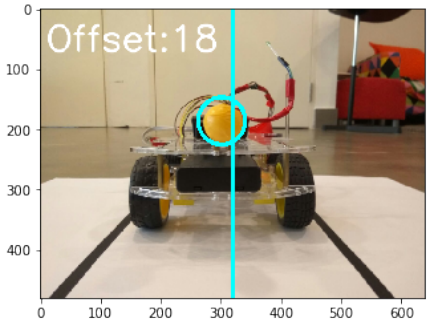
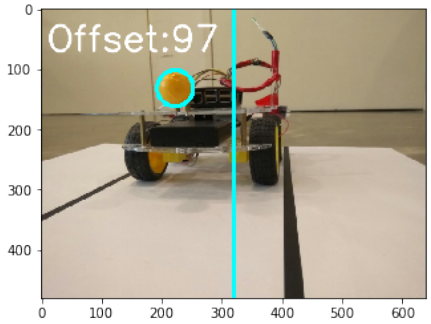


Figure 3.14: Output: large offset



There is one more useful piece of information obtainable from the circle - the radius of the circle will serve as an approximation for how far away the object is. If the radius is small, then the leading car is far away and the trailing car can afford to accelerate, and vice versa. This technique worked reasonably well in testing, however it is vulnerable to error due to other yellow blotches. A better technique based on this approach would be to have a uniquely identifying shape and train a specialized classifier for it, which is partially what the next section does.

3.2.6 Traffic Sign Detection

Given an image containing street signs, the first task is to locate the street signs within the image, i.e. return a bounding box of coordinates around each streetsign.

This project used the Viola-Jones object detection framework [13] originally designed for face detection, and adapted it to streetsigns. It was chosen because it has high accuracy, and runs in real time, at least for faces.

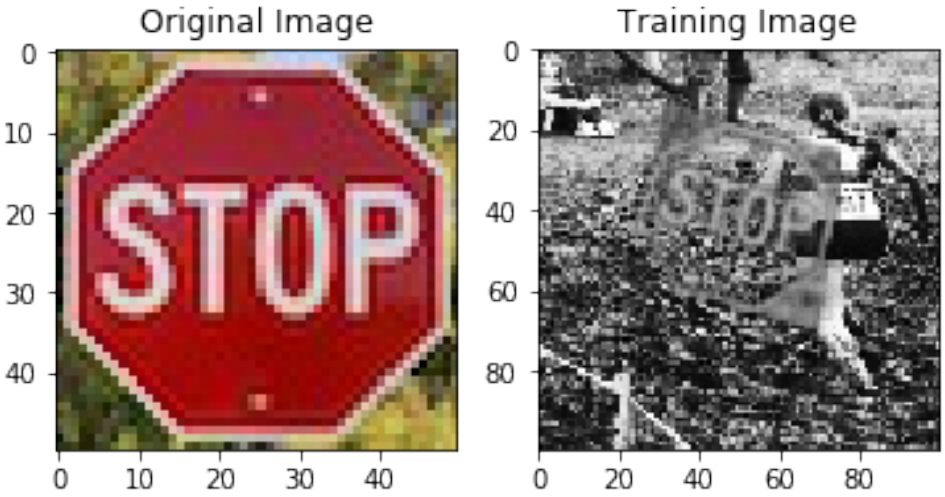
The training stage involves selecting a series of features that can discriminate, with high accuracy, between the presence or absence of the target.

1. **Haar-like features** - Four types of rectangular features are used to match patterns in the image. They are applied to patches of varying size in the image, and the value obtained is the sum of pixel intensities in the white areas minus the sum of pixel intensities in the dark areas.

2. **Integral Image** - A useful technique to speed up the process is to pre-compute the integral image, a cumulative sum of the pixel intensities from top left to bottom right of an image. This allows any Haar-like feature to be computed in a constant amount of time.

3. **AdaBoost and Cascade** - Adaboost is a method for constructing a strong classifier with high accuracy from a composite of stacked weak classifiers. Each Haar-like feature can be considered a weak classifier in that it gives a prediction, with low accuracy, of whether the target is

Figure 3.15: Training data used for the single-class classifier



Source: *wikipedia.org*

Figure 3.16: *Haar-like feature types*

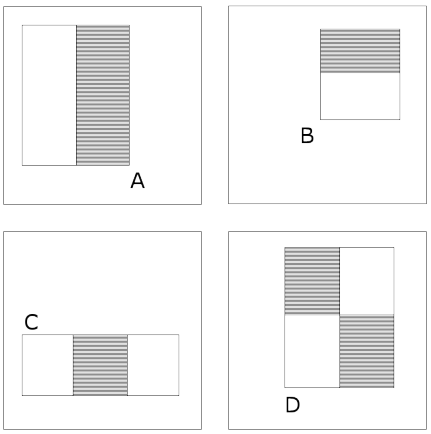
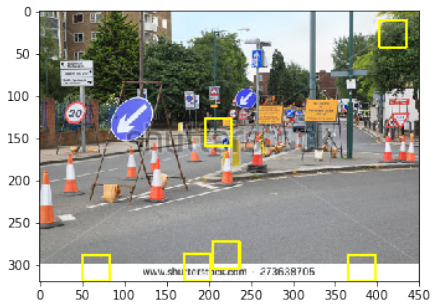


Figure 3.17: *Failed attempt at multi-class classification*



present or not. Since there are many possible features - in a 24x24 window alone there are 162336 features - the job of AdaBoost is to select only the most relevant features based on the positive and negative examples provided, and weight them accordingly. Finally, the classifiers are organized in a hierarchical cascade so that unpromising candidates can be quickly rejected.

This project did not successfully solve the problem in the time given. The first attempt was a multi-class traffic sign detector, trained using a set of OpenCV functions and custom positive/negative datasets. Unfortunately, there was neither the time nor computational resources to achieve significant accuracy.

The second attempt used a single stopsign as the detection target. A positive dataset was created by embedding this image, with various transformations, into negative images, a common technique which can be seen in figure 3.2.6. This should have improved accuracy, but did not, potentially due to the particular positive image selected, or the general unsuitability of this method for traffic signs.

3.2.7 Traffic Sign Classification

This is a well-defined multi-class image-classification problem with a high quality dataset - the German Traffic Sign Recognition Benchmark (GTSRB) dataset, consisting of more than 50 000 images in 43 classes [10]. The current best result of the GTSRB is 99.46% accuracy, achieved by a committee of convolutional neural networks (CNNs). This result is substantially better than human performance (98.84%) and other machine learning techniques such as random forests (96.14%) or Linear Discriminant Analysis (95.68%) [4].

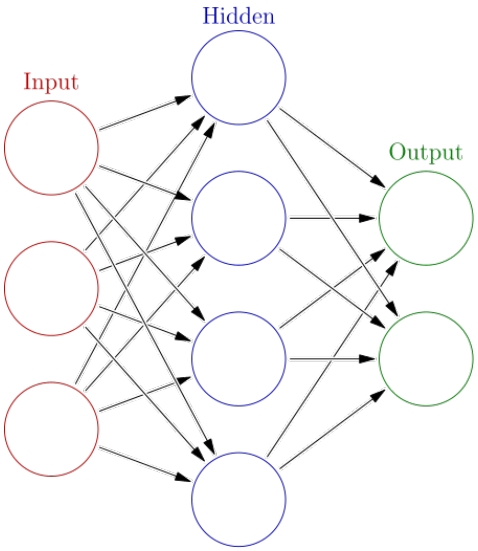
For this project, the model complexity had to be weighted against the Pi's

computational constraints. In the end, the best solution was to use a simple convolutional neural network, based on Yann LeCun's groundbreaking LeNet 5 architecture. [5]. Jeremy Shannon's tutorial [9] was used as a starting point..

A neural network is a computational model based on a large collection of simple neural units (neurons) that loosely mimics a biological brain's functions [16]. A neural network is often represented as a directed graph with nodes as neurons arranged in layers, and edges as connections to neurons in subsequent layers.

The output of a neuron can be written as $f(\sum_i (w_i \cdot x_i) + b)$ where x_i is the output (or activation) of a neuron in a previous layer, w_i is a weighting for that activation, b is the "bias" or threshold value of the neuron in question, and f is an activation function. The parameters $\{w\}, \{b\}$ govern the behaviour of the network. During the training stage, an input is fed through the network, and the final result is compared to a ground truth. The error

Figure 3.18: A simple neural network with a single hidden layer



Source: *Wikipedia.org*

is then computed, and propagated back through the network, where it is used to update the values of the parameters $\{w\}, \{b\}$.

While a full discussion of neural networks is beyond the scope of this report, it can be observed that the hierarchical structure of neural networks allows them to learn features of increasing complexity,

A convolutional neural network improves on a plain vanilla NN by introducing the following features:

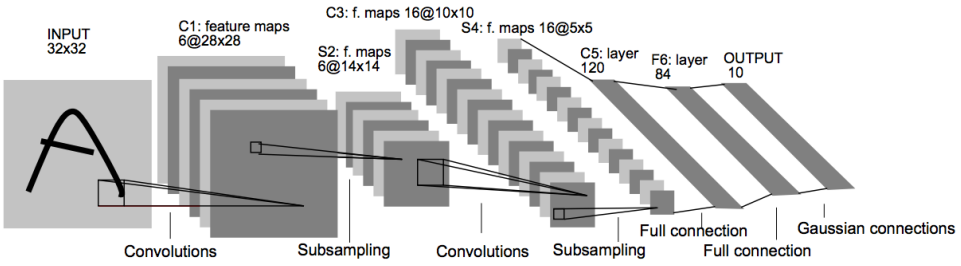
1. **Convolutions with shared weights** - A CNN takes advantage of the correlations between neighbouring pixels or neurons by mapping subregions of the input (e.g a 5x5 grid of neurons) to single neurons in the convolutional layer. These neurons can then learn hierarchical concepts, from raw pixel data, to edges, shapes and objects. Crucially, the same weight value is shared for all neurons in a particular feature map, allowing for a sparse and efficient implementation.

2. **Subsampling** - To reduce dimensionality and provide some degree of spatial invariance, a subsampling layer is often added after a convolution layer. Commonly, a single neuron in the subsampling layer will perform either a maximum or averaging operation on the outputs of a 2x2 region on the previous convolution layer.

This project's architecture is similar to the Lenet architecture seen in Figure 3.2.7, except for the final layers.

Operation	Dimensionality	Output Size
Input		32x32x3
Convolution	6 5x5 maps	28x28x6
Max-pooling	2x2	14x14x6
Convolution	16 5x5 maps	10x10x16
Max-pooling	2x2	5x5x16
Convolution	400 5x5 maps	1x1x400
Full connection	800	800
Dropout	50%	800
Full connection	43	43

Figure 3.19: LeNet 5 Neural Network Architecture



Source: *yann.lecun.com*

There are a number of techniques to improving the performance of CNNs. The ones used in this project can be summarized below:

- **Dropout** - This involves randomly deactivating some proportion of the neurons during the forward pass, thus allowing the network to become less sensitive to any one neuron's contribution. This reduces

overfitting in the model, which is the phenomenon whereby the network's parameters are tuned to the training data very well, but do not generalize well to new examples. In this project's network, a dropout rate of 50% was used, meaning that half of the neurons were inactive during each training step.

- **Dataset augmentation** - The more data a network has to train with, the better it will perform. For this model, the size of the training data was doubled by performing random rotations of 90, 180 or 270°. This also increases the invariance of the network, allowing it to recognize more configurations of the target object.

- **Learning Rate Optimizer** - This implementation used the Adaptive Moment Estimation (Adam) Optimizer. This is a method of updating the weights and biases of the network through an adaptive learning rate for each parameter, allowing the network to train faster without sacrificing accuracy [8].

- **ReLU activation function** - One problem with classic activation functions such as sigmoid or tanh is that they can become saturated i.e. their gradient falls to almost zero for certain input values, meaning they slow down training massively. By using the Rectified Linear Units function, the gradient of the activation function is always sufficiently

high to backpropagate errors through the network.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

In the end, this project's model achieved an accuracy of 97.12% on the test set, which is substantially lower than the best recorded result, but better than the other machine learning models mentioned in the benchmark.

Chapter 4

Discussion

4.1 Limitations and Optimizations

For anyone wishing to embark on a similar project, the following points are worth keeping in mind.

4.1.1 Hardware

- **The Pi** - Although the Raspberry Pi is fairly powerful for its size and cost, it was quite computationally limited, particularly for the Computer Vision problems. Even for the lane detection problem, the achievable frame rate was around 6 fps, which limited the reaction time of the system. Some of the computational load could have been outsourced to a second Pi or Arduino. Using an Arduino for the ultrasound module, for example, would have improved the accuracy of my readings due to its real-time guarantees. Another option would be to have the Pi stream video to a computer, and outsource all computation to the

computer, but this would reduce the autonomy of each individual car and possibly create a performance bottleneck around the computer.

- **Ultrasound** A better ultrasound sensor could have been found that did not depend on the Pi for computing the distance. I2C circuits would have been useful in order to hook up multiple ultrasound sensors, giving better all-round object avoidance capabilities. In addition, the ultrasound suffered from a sensory bottleneck, as readings could only be taken every 100ms, to avoid picking up on echoes from the previous readings. This limited the reaction time of the car to obstacles.

- **Electronics** The author's general electronics knowledge was quite limited at the start of this project, and he wasted a lot of time figuring out how to do basic things like splicing two wires together. Two of the motors were damaged fairly significantly by resoldering them multiple times. In future, breadboards would be used more often to prototype basic circuits.

- **Car Build** - For one of the cars, the author tried to design a custom chassis using the school's laser cutter. This design proved infeasible due to a lack of integration with other parts (axles, screws etc), and the build of the third car was abandoned in favour of more software development.

4.1.2 Software

- **Multithreading** - The performance effects of multithreading could have been investigated, particularly due to the OO structure. In general, more speed benchmarking could have been done earlier on to improve performance.
- **Language** - Python was great for testing things out, and getting quick results, but has a large computational overhead. For a version 2.0 of this project, a language like C++ would be better, and possibly a different development environment altogether, like ROS (Robot Operating System).
- **Lane Detection** - Curved line detection was never solved. While straight-line RANSAC works well for straight roads, it tended to give inaccurate results for cornering. In future, Hough Transform and RANSAC for curves would be investigated.
- **Motor Control** Currently, the driving module only takes as input the 'instantaneous offset' i.e. how far the car is currently offset from the desired position, rather than accounting for future changes as well. In future, a full-on driving control system could be developed using a simulator.
- **Traffic Sign Recognition** - There were multiple things that could improve the classification performance of the neural network, including

more dataset augmentation, ensembles of convnets, or deeper architectures. In addition, the network has only a fixed 32x32 detection window, but the street signs it encounters could be of different sizes. Investigating feature extraction or dimensionality reduction techniques could be promising.

4.2 Further Work

- **Bluetooth** - The original plan was to use bluetooth as an indoor navigation system, by setting up some bluetooth beacons and having the cars triangulate their position depending on their distance from the beacons. This could be a very interesting field to explore, as well as using bluetooth for the cars to communicate with one another.
- **Co-ordination** - related to the previous point, driving and co-ordination algorithms could be explored that allow for multiple agents to collaborate and solve problems together.
- **Depth perception** - Real-world autonomous vehicles use Lidar for depth-perception, allowing them to rapidly scan their environment for oncoming obstacles. While Lidar is still commercially expensive, perhaps a prototype version could be constructed or obtained.
- **Optical flow** - Having the car take into account how fast the objects in its environment are moving could be useful. This is a field called

Optical Flow, and is very relevant to autonomous navigation.

4.3 Personal Reflections

I have really enjoyed working on this Capstone project. I definitely met my goal of gaining a broad exposure to a range of different techniques, and I particularly enjoyed working with actual hardware and seeing the results.

There were many setbacks and frustrations along the way. Sometimes I would hit a problem that I didn't know how to solve, and I'd spend hours digging around on obscure Linux forums and copy-pasting code in the hopes that something would work. Sometimes I would have to leave certain sections alone, and return to them weeks later when I had enough energy for a fresh attempt. Yet perhaps the most significant takeaway of the capstone was realizing that, with enough patience and effort, I could persevere and figure things out on my own.

Bibliography

[1] Ron Amadeo. Google's Waymo Invests in LIDAR Technology, Cuts Costs by 90 Percent, 2017. [Online; accessed 18-March-2017].

[2] Bryan Clark. How Self-Driving Cars Work: The Nuts and Bolts Behind Google's Autonomous Car Program, 2015. [Online; accessed 21-March-2017].

[3] Jordan Dee. Pulse-width Modulation, 2017. [Online; accessed 18-March-2017].

[4] Institut für Neuroinformatik. IJCNN 2011 Competition Results, 2015. [Online; accessed 18-March-2017].

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[6] Uber Newsroom. Pittsburgh, your Self-Driving Uber is Arriving Now, 2017. [Online; accessed 18-March-2017].

[7] Adrian Rosebrock. OpenCV Track Object Movement, 2017. [Online; accessed 18-March-2017].

[8] Sebastien Ruden. An Overview of Gradient Descent Optimization Algorithms, 2016. [Online; accessed 20-March-2017].

[9] Jeremy Shannon. Project: Build a Traffic Sign Recognition Classifier. https://github.com/jeremy-shannon/CarND-Traffic-Sign-Classifier-Project, 2017.

[10] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[11] Ingmar Stapel. Raspberry Pi WiFi Radio Controlled RC Vehicle - Introduction, 2014. [Online; accessed 21-March-2017].

[12] Michel van Biezen. The Kalman Filter, 2017. [Online; accessed 18-March-2017].

[13] Paul Viola and Michael Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features, 2001.

[14] Waymo. Journey, 2017. [Online; accessed 18-March-2017].

[15] Marc Weber. Where To? A History of Autonomous Vehicles, 2014. [Online; accessed 18-March-2017].

[16] Wikipedia. Artificial Neural Network — wikipedia, The Free Encyclopedia, 2017. [Online; accessed 21-March-2017].

[17] Wikipedia. H Bridge — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 18-March-2017].

[18] Wikipedia. Hough Transform — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 19-March-2017].

[19] Wikipedia. Random Sample Consensus — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 19-March-2017].