



Lane keeping for autonomous vehicles

Implementation and evaluation of a lane detection algorithm for a down-scaled autonomous vehicle

JOHAN EHRENFORS, STANISLAV MINKO

Bachelor's Thesis at ITM
Supervisor: Damir Nesic
Examiner: Nihad Subasic

MDAB 645 MMK 2017:27

Abstract

According to reports from NHTSA, over 90% of all automobile crashes occur due to driver error. The resulting casualties could potentially have been avoided by implementing autonomous features in automobiles. One specific situation that can be automated is lane keeping while driving on the highway.

The technology to allow autonomous driving on highways already exists in consumer vehicles, such as the Tesla Motor's Model S, but is limited to high-end vehicles. This bachelor thesis investigates how a simple control system can be designed in order to keep an autonomous vehicle centered in a highway lane by using a single digital camera. The performance of the demonstrator was evaluated by measuring the error of the calculated vehicle position as well as the maximum lane positioning error for different image sampling frequencies.

A miniature demonstrator vehicle was assembled from off-the-shelf components and custom parts. A digital camera was used to capture the area in front of the vehicle. From this image, the road lane positions were extracted using a combination of Canny edge detection and Hough lines. An approximate vehicle road position was calculated from the placement of the detected lanes in the visual field of the camera. The steering was regulated by a PID-controller which used the lane position error to steer the vehicle.

The final demonstrator software was able to process images captured at up to approximately 15 frames per second, which led to an average lane positional error of roughly 10 percent of the lane width. The calculated vehicle lane position was found to be of acceptable accuracy for this type of demonstrator.

The demonstrator performance was good within the scope of the project, and the implementation can probably be applied to low-risk situations.

Referat

Implementation och utvärdering av fildetektion för ett nedskalat självkörande fordon

Enligt siffror från amerikanska NHTSA, orsakas över 90% av alla bilolyckor av förarfel. De resulterande dödsfallen skulle ha kunnat förhindras genom att implementera autonoma funktioner i bilar. En specifik situation som kan automatiseras är filhållning vid motorvägskörning.

Teknologin för detta finns redan i vissa bilar på marknaden, exempelvis Tesla Motors Model S, men det är än så länge begränsat till toppskiktet. Det här kandidatexamensarbetet undersöker hur man kan konstruera ett enkelt styrsystem för att hålla en bil inom en motorvägsfil med hjälp av en enda digitalkamera. Prestandan av vår demonstrationsprototyp evaluerades genom att mäta felet av bilens placering inom filen, samt det maximala placeringsfelet vid olika bilduppdateringsfrekvenser.

Ett fordon i miniatyr sammanställdes från butiksköpta komponenter och skräddarsydda delar. En digitalkamera användes för att fånga området framför fordonet. Från denna bild kunde körfältslinjernas läge approximeras genom en kombination av Canny kantdetektion och Hough linjer. Bilens ungefärliga körfältsposition beräknades från läget av körfältslinjerna inom kamerans synfält. Styrningen kontrollerades av en PID-kontroller som använde positioneringsfelet som indata.

Den slutgiltiga mjukvaran för fordonet klarade av att behandla uppemot 15 bilder per sekund, vilket ledde till ett placeringsfel på cirka 10% av körfältets bredd. Den beräknade körfältsplaceringen ansågs vara tillräckligt noggrann för denna typ av demonstrator.

Fordonsprototypens prestanda inom projektets ramar var god, och den framtagna lösningen kan förmodligen tillämpas rakt av för situationer med låg risk.

Acknowledgements

We would like to thank our examiner Nihad Subasic for valuable lectures and supportive assistance. Further we would like to thank our supervisor Damir Nesic for his input on this report, but also the entire Mechatronics group, both assistants and students, for their valuable help on how to solve the problems we encountered. This project would not have turned out as well as it did without them! Thank you ITM for the economic assistance when purchasing components and providing access to the necessary prototyping tools.

Contents

1	Introduction	1
1.1	Background	1
1.2	Purpose	1
1.3	Scope	2
1.4	Method	2
2	Theoretical background	4
2.1	Ultrasonic distance measurement	4
2.2	PID-controller Algorithm	4
2.3	Image processing	5
2.3.1	Canny edge detection	5
2.3.2	Hough line detection	6
2.4	DC-motor control	7
2.4.1	Pulse Width Modulation	8
2.4.2	H-bridge	8
2.4.3	Servo motor	9
3	Demonstrator	10
3.1	Problem formulation	10
3.2	Hardware	10
3.2.1	Raspberry Pi 3	11
3.2.2	Ultrasonic distance sensor HC-SR04	11
3.2.3	Camera Module v2	12
3.2.4	Futaba S3003 Servo Motor	12
3.2.5	DG02S DC-motors	12
3.2.6	L298N-based custom H-bridge	13
3.2.7	Power regulation using LM2596	13
3.3	Software	14
3.3.1	Operating system	14
3.3.2	Image processing	14
3.3.3	Lane position approximation	17
3.3.4	Control system	18

4	Experiments	19
4.1	Experiment 1: Error of calculated lane position	19
4.1.1	Experiment description	19
4.1.2	Experiment results	19
4.1.3	Experiment discussion	21
4.1.4	Experiment conclusion	21
4.2	Experiment 2: Positioning error depending on camera frame rate . .	21
4.2.1	Experiment description	21
4.2.2	Experiment result	22
4.2.3	Experiment discussion	23
4.2.4	Experiment conclusions	23
5	Discussion and conclusions	24
5.1	Discussion	24
5.2	Conclusion	25
6	Recommendations and future work	26
6.1	Recommendations	26
6.2	Future work	27
	Bibliography	28
	Appendices	29
A	Python 2.7 Code	30
B	Experiment description	47
C	Raw Data from Experiment 1	51
D	Raw Data from Experiment 2	52

List of Figures

2.1	Depiction of gradients in a grayscale image.	6
2.2	(left) Parametrization of line in xy-space (right) Sinusoidal curves in Hough parameter space. Modified from [1].	7
2.3	Visual representation of PWM. [2]	8
2.4	A simplified H-bridge schematic, using switches instead of transistors. Modified from [3].	8
3.1	Diagram showing the connections between hardware components.	10
3.2	Photograph of the completed demonstrator, with components marked. .	11
3.3	Timing diagram for interfacing with the HC-SR04. Image from [4]. . . .	12
3.4	Pin out diagram for USB-A [5].	13
3.5	Image captured by Camera Module v2.	15
3.6	Canny Edge Detection algorithms output.	15
3.7	Hough lines on top of cropped original image.	15
3.8	Detected road edges after image processing.	16
3.9	(left) Lane fitting using linear interpolation (right) Lane fitting focusing on average slope of detected lines.	16
3.10	Captured image with center of image (dashed) and lane midpoint (dot) marked.	17
4.1	The vehicle's calculated lane position compared to its actual position. .	20
4.2	Error of calculated vehicle position compared to its actual position. . .	20
4.3	Average and median errors from experiment 2.	22
5.1	The field of view of the Camera Module v2.	24

List of Tables

2.1	Effect of increasing PID gain values (Adapted from[6]).	5
-----	---	---

Chapter 1

Introduction

Autonomous vehicles have the potential to save thousands of lives per year, and are currently being developed by many companies, such as Tesla [7] and Ford [8]. This report explains how a particular prototype platform for autonomous lane keeping was assembled, including the necessary components, algorithms, and complete software implementation, as well as considerations regarding the performance of the vehicle under controlled circumstances similar to highway conditions.

1.1 Background

Transportation has vastly increased in complexity throughout the last century, from Henry Ford's mass produced Model T automobile that began selling to the masses in 1908 [8], to today's modern electric Tesla Model S featuring semi-autonomous driving [7]. The reason for such development is the advancements in technology that have led to cheaper and more reliable sensors, more efficient manufacturing techniques, and software controlled systems, consequently resulting in increased performance, reliability, and safety of modern vehicles.

According to the United State's National Highway Traffic Safety Administration [9], it is estimated that over 90% of all crashes between 2005 to 2007, across the entire United States, occurred due to driver error. Out of the roughly three million reported accidents, 11.3% of the drivers were either killed or suffered incapacitating injuries. One option to reduce the risks associated with driving is to replace the human factor with software that mimics human driver behavior.

1.2 Purpose

The main goal of this bachelor thesis project is to investigate how to build a system capable of staying in the middle of the lane during autonomous driving, as well as implementing a set of algorithms to demonstrate and evaluate this behavior with a down-sized prototype vehicle. The research question for the project is as follows:

How can a control system be designed to keep an autonomous vehicle in the middle of a highway lane using a digital camera?

To be able to evaluate the performance of the project the question was further split into two subquestions:

1. What is the error of the calculated vehicle position for different positions on the road with implemented lane detection algorithm?
2. For a fixed vehicle speed, what is the maximum lane positioning error for different image sampling frequencies?

1.3 Scope

This project was limited to building a down-scaled demonstrator to represent a full-scale autonomous vehicle. The focus was on investigating the research questions, not to create a fully functional vehicle. As such, the demonstrator is not intended to be as mechanically complex or robust as a commercial vehicle, and will only be operated under controlled environments. Meeting traffic and multiple lanes will not be investigated.

This demonstrator must be built from off-the-shelf hardware, or manufactured using the available machinery. The algorithms to be used have to run in real time on an available micro-controller.

In order to answer the research questions, the vehicle must be able to stay in the middle of model road built for experiment purposes. The track to be used for answering the research questions is defined by two parallel lane markings set at a constant distance apart.

By the end of the project, the demonstrator should be able to navigate along a realistically curved highway environment without going off track with any of the wheels. The vehicle should also be able to keep from colliding with a frontal obstacle, as well as handling temporarily obscured or missing lane markings.

1.4 Method

Initially, the authors listed the minimum requirements for the demonstrator, without going into the implementation or precise hardware:

- Similar design to regular cars: four wheels and front wheel steering
- Able to propel and steer the vehicle
- Able to operate without physical tethers
- Able to capture digital imagery at refresh rates exceeding 20 frames per second

1.4. METHOD

- Able to detect obstacles directly in front of the vehicle
- Maximum length, width, and height of 30, 25, and 20 centimeters respectively

To research potential solutions for building a demonstrator fulfilling the mentioned requirements, scientific literature, popular science articles, and online forums were reviewed. The primary focus was on finding similar projects, and reading about topics such as image processing, computer vision, lane detection algorithms, and obstacle detection.

Several sketches were made to plan the component layout and assembly. This stage culminated with creating a 3D-model to visualize the placement of the components, and to determine the total size of the demonstrator. After this digital mock-up had been created, a list of specific components was submitted for purchase through the course administrators.

The demonstrator was assembled and underwent minor iterative changes to improve component fit. Test tracks to test the impact of different software solutions were marked out using white masking tape to represent road markings.

Once the demonstrator was capable of staying within the lane markings, experiments to answer the research questions were carried out. The exact experiment design, results, discussion, and conclusions are presented later in this report.

Chapter 2

Theoretical background

This project includes the use of various sensors, actuators, and algorithms for which the general theoretical background is presented in this chapter.

2.1 Ultrasonic distance measurement

Ultrasonic distance sensors emit a pulse of ultrasonic sound waves above the human audible frequency range, and senses when the pulse is reflected off of a solid object. The distance s to the object can be calculated using the known speed of sound v and the time t it takes for the reflected ultrasonic pulse to be detected by the ultrasonic distance sensor. The relationship in equation (2.1) below is used to determine the distance.

$$s = \frac{vt}{2} \tag{2.1}$$

The speed of sound in air primarily depends on air pressure and temperature. Close to the sea level, and at around room temperature (25°C), the speed of sound is approximately 343 m/s. For the purposes of developing a demonstrator, assuming that these conditions apply is considered satisfactory, so no temperature sensor was included.

2.2 PID-controller Algorithm

Control systems are used for a variety of systems, e.g. in order to regulate heating systems, balance robots, stabilize water levels in dams, etc.

PID-controllers are digital control systems that use a combination of weighted Proportional, Integral, and Derivative components depending on the error $e(t)$ between the current measured value and a specified desired value, in order to calculate the required change $u(t)$ to a control signal. Coupled with modern micro-controllers, mathematical calculations can be carried out quickly enough to control time-critical systems, such as emergency braking.

2.3. IMAGE PROCESSING

The operation of a PID-controller is captured by equation (2.2), and the behavior of the system depends on the configuration of the three values: proportional gain (K_P), integral gain (K_I), and derivative gain (K_D) [10].

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{d}{dt} e(t) \quad (2.2)$$

Behavior that is impacted by the choices of these values are *overshoot*, *static error*, *settling time* and *rise time*. Overshoot is caused by over-regulating and "shooting past" the desired value. Static error (steady state error) is a constant error that remains over time. Settling time is the time it takes for a system's step response to continuously stay within a certain margin of error from the desired value. Rise time is how long it takes to reach the desired value from a previous steady state. For braking and steering applications, too much of an overshoot or too long of a rise time might lead to a scenario where an accident occurs despite the system initially working in the right direction. If a static error occurs, the vehicle may consistently stay off center. *Integral windup* occurs when the error continuously increases the integral part of the PID-controller output, until there is a risk that the controller output leads to significant overshoot upon a sudden change in the input data. For a basic understanding of how increase of the the values K_P , K_I , and K_D affect a control system, see table 2.1.

Table 2.1. Effect of increasing PID gain values (Adapted from[6]).

Gain value	Rise time	Overshoot	Settling time	Static error
K_P	Decrease	Increase	Small change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	Increase	Decrease	Decrease	No change

2.3 Image processing

The images captured by a digital camera need to be processed in order to be able to detect the lane lines and calculate an approximate lane position error.

2.3.1 Canny edge detection

For many image recognition applications, it is necessary to detect edges in an image, containing one or more color channels. The *Canny edge detection algorithm* returns a monochromatic image (containing only black and white pixels) describing whether each pixel is part of an edge or not [1]. To the human eye, the outlines of the original image will appear to be marked in the output image, as shown in figure 3.6 in chapter 3.3.2.

An edge is defined as a *set of connected pixels that lie on the boundary between two regions* [1], which are separated by a gradient in the color values. Ideally, these two regions are distinguished by two entirely different colors, but realistically there is a transitional gradient between the two regions. This is illustrated below in figure 2.1.

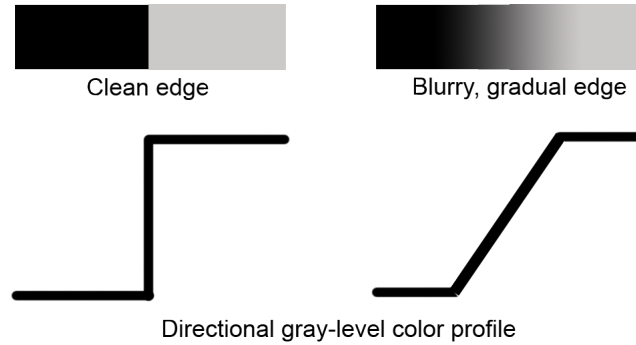


Figure 2.1. Depiction of gradients in a grayscale image.

The value of the gradient is used to determine where edges occur. If the gradient exceeds a threshold value, this is classified as an edge. This process is complicated if noise is introduced to the edge, since the gradient will vary greatly. One solution to this issue is using a *smoothing algorithm*, such as a *Gaussian filter*, to remove noise from the image [1]. Additionally, by decreasing the image resolution, noisy areas can be averaged into a single pixel value, while also decreasing the execution time required for the edge detection algorithm, since there is less data to handle.

This approach allows the detection of edges, or gradient changes, in a one-dimensional array of values. This process can be expanded to handle two-dimensional images, which may also contain several color channels, to what is known as the Canny edge detection algorithm. When analyzing edges in an image, several directional gradients can be superimposed to generate an output image containing the edge pixels. More information on Canny edge detection can be found in Gonzalez and Woods' *Digital Imaging Processing* [1].

2.3.2 Hough line detection

In practice, edge detection algorithms cannot be guaranteed to detect all edges present in an image, due to noise or gradients below the threshold. In order to maximize the quality of edge detection, a linking algorithm is usually designed to assemble edge pixels into meaningful edges or region boundaries. A common way to accomplish this is by using Hough transform to link the edges into lines [1]. It is a global approach which works with the entire image.

The idea of edge linking using Hough transform is to represent each possible line passing through two or more edge pixels in a Hough parameter space($\rho\theta$ -plane), see

2.4. DC-MOTOR CONTROL

figure 2.2 below.

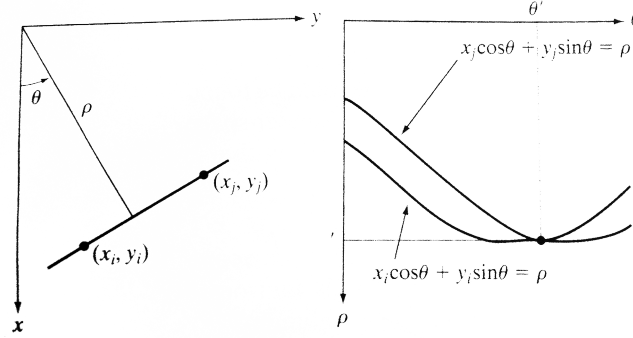


Figure 2.2. (left) Parametrization of line in xy-space (right) Sinusoidal curves in Hough parameter space. Modified from [1].

Each curve in Hough parameter space represents all the pixels on that specific line in the xy-plane. The points where the number of intersections exceeds a set threshold value are classified as lines. More information on this topic can be found in *Digital Image Processing* [1].

2.4 DC-motor control

DC-motors are rotary electrical devices that are actuated by the flow of direct current, ergo the term DC-motor. It is often necessary to control these motors' rotational speed, torque, and position.

A model of a DC-motor is captured by the equations (2.3), (2.4), and (2.5) [11]. The values used are the voltage over the motor U_A , internal resistance R_A , device constant $K_2\phi$, voltage drop E over the rotor's winding, motor torque M , rotational speed ω , and current draw I_A .

$$U_A = R_A I_A + E \quad (2.3)$$

$$E = K_2 \phi \omega \quad (2.4)$$

$$M = K_2 \phi I_A \quad (2.5)$$

In order to control the rotational speed ω and torque M of a regular rotational DC-motor, the voltage U_A can be regulated, as R_A is a device constant. Propelling a vehicle forwards requires that the torque M exceeds the moment produced by frictional forces. If either the torque or rotational speed is known, the other value can be calculated.

2.4.1 Pulse Width Modulation

The voltage supplied to the leads of the DC-motor can be varied by modulating the supply voltage using Pulse Width Modulation (PWM). The principle behind PWM is that the voltage is switched on and off, resulting in an average power supply value lower than the nominal power supply value, for example when controlling a DC-motor's performance. The percentage of the time that the power is on is called the *duty cycle*. The time between pulses is referred to as the *period time*. The basic idea of PWM is shown in figure 2.3.

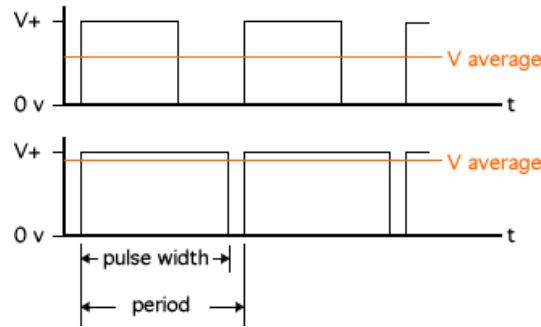


Figure 2.3. Visual representation of PWM. [2]

2.4.2 H-bridge

In some instances, it is necessary for an electric motor's rotation be reversed. A simple solution is to reverse the polarity of the motor, but doing this mechanically is highly impractical. Instead, four transistors, which can be used as digital switches in circuits, can be connected in the arrangement shown in figure 2.4. This arrangement is known as an *H-bridge*.

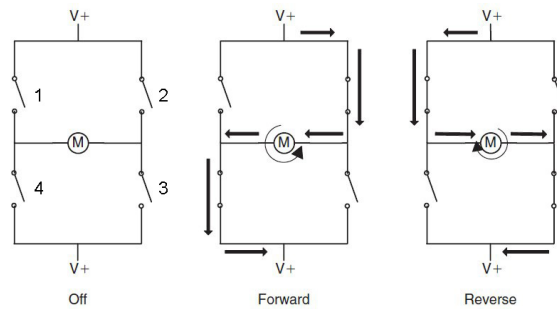


Figure 2.4. A simplified H-bridge schematic, using switches instead of transistors. Modified from [3].

2.4. DC-MOTOR CONTROL

By activating transistors 1 and 3, current is allowed to pass through the motor from left to right. If instead transistors 2 and 4 are activated, the current will pass from right to left, which will turn the motor in the opposite direction.

Another positive aspect of using an H-bridge, is that a lower voltage can be used to trigger the transistors in order to control high-powered motors or other hardware. This allows for remote digital control of heavy machinery. By using a PWM-signal as the trigger for the H-bridge transistors, the motor speed can be varied in the desired direction.

2.4.3 Servo motor

Servo motors are rotational or linear actuators that allow precise angular or linear position control. It requires a sensor for positioning feedback to keep desired position value and a motor to turn. Three common types of servo motors are *positional rotation*, *continuous rotation* and *linear* servos.

Any variations in the input signal, or the reading thereof, can lead to *jitter*, which is unintended behavior where the servo motor tries to jump from one position or speed to another, resulting in a twitching rotor.

Chapter 3

Demonstrator

3.1 Problem formulation

In accordance to the purpose established in the beginning of this project, a down-scaled autonomous vehicle was constructed in order to answer the two research questions. This chapter presents how the demonstrator was designed and programmed to stay centered between two lane markings.

3.2 Hardware

The complete hardware overview is shown in figure 3.1, where the bold arrows signify power being supplied, and the lighter arrows represent control signals being sent or received.

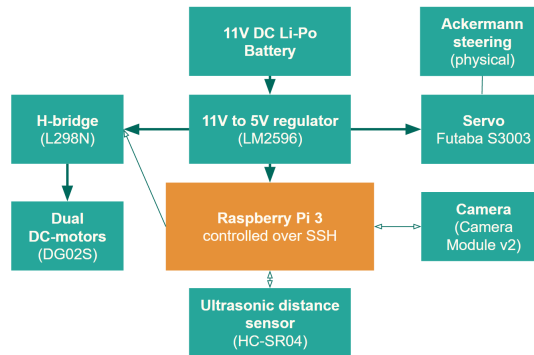


Figure 3.1. Diagram showing the connections between hardware components.

The completed demonstrator is shown in figure 3.2, with all the components from figure 3.1 marked.

3.2. HARDWARE

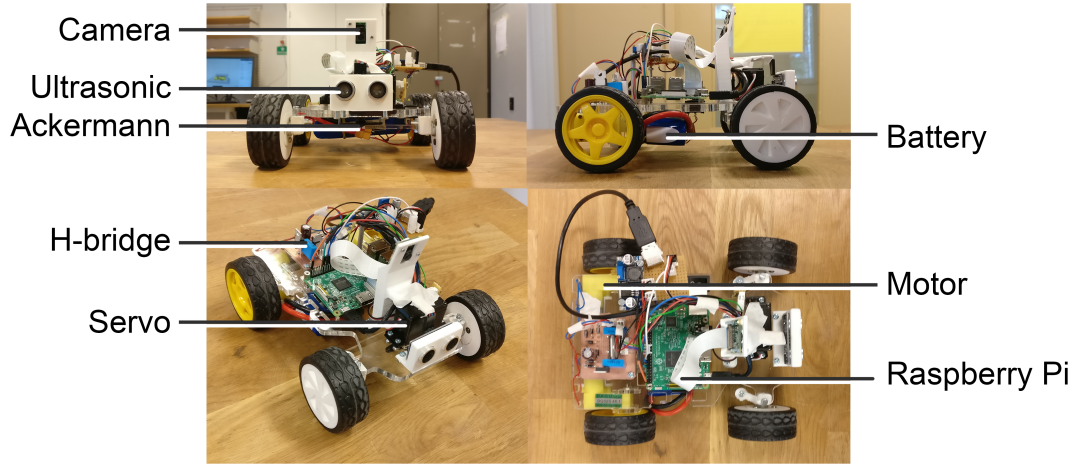


Figure 3.2. Photograph of the completed demonstrator, with components marked.

The demonstrator has a final length, width, and height of approximately 21, 18, and 14 cm respectively. This is within the size limitations specified in the demonstrator requirements.

3.2.1 Raspberry Pi 3

The demonstrator is controlled by a Raspberry Pi 3 [12], which is a full ARM-based quad-core computer. This processing power was deemed necessary to handle the image processing methods used. The Raspberry Pi board contains 40 general purpose input/output (GPIO) pins which can be used to control and/or power various sensors and other components. The board is also capable of wireless connections, allowing for the system to be controlled over SSH [13].

3.2.2 Ultrasonic distance sensor HC-SR04

The ultrasonic distance sensor HC-SR04 [4] was chosen for the demonstrator. The chip features the pulse emitter and receiver, and is interfaced through four pins. They are labelled VCC (5V), GND (0V), ECHO, and TRIG.

The pulse is emitted from the sensor when TRIG pin receives a high signal. Sensor's field of view is approximately 15 degrees in a forward-facing cone shape. The ECHO pin switches to a high-voltage state for the time it took for the pulse to be returned, as shown in 3.3. The duration of the high signal can be measured using a micro-controller, which is then used to calculate the distance to the reflecting object.

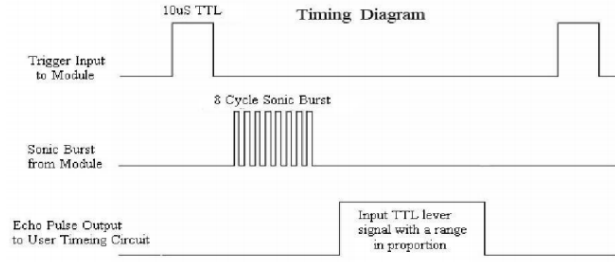


Figure 3.3. Timing diagram for interfacing with the HC-SR04. Image from [4].

3.2.3 Camera Module v2

In order to capture the world in front of the vehicle a Camera Module v2 [14] is used, which is designed specifically for the Raspberry Pi. The camera can capture images up to a resolution of 8 megapixel. Note that capturing images at a lower resolution is preferable for decreasing the processing time for the lane recognition algorithm. The camera module features a fixed focus lens.

3.2.4 Futaba S3003 Servo Motor

A positional rotation servo motor was deemed to be the best solution for steering the wheels of the demonstrator vehicle through Ackermann steering, as accurate positioning is necessary. The chosen model was a *Futaba S3003* [15], because it is a high-quality component that was readily available from local retailers.

The S3003 allows a 180 degree range of motion. The data sheet specifies a no-load rotational speed of 0.23 seconds per 60 degrees of rotation, at 4.8V input. Assuming small adjustments, this provides fast enough performance for the demonstrator. Its interface is comprised of three wires; 4.8-6.0V supply voltage (red), GND (black), and the control signal (white).

The positional rotation servo motor's position can be set by varying the pulse width of the control signal. For every cycle of 20ms, a high signal should be applied to the input pin between 1 to 2 ms, depending on the desired position. This signal can be generated through PWM with a period of 20ms (50Hz). When a position is set, the motor will attempt to hold that position using an internal potentiometer to detect the position as part of a closed-loop control system.

3.2.5 DG02S DC-motors

The Dagu DG02S motors that were used have accompanying wheels with a diameter of 65 millimeters. A supply voltage of 3V is recommended, and the model has a no-load speed of roughly 65 rotations per minute [16]. This should result in a top speed of roughly 0.22 m/s, disregarding any friction. Tests where the full voltage of the battery was supplied to the motors, were determined to work well without

3.2. HARDWARE

excessive heat production, which allowed for a simpler power supply system to be used, as well as a higher top speed.

3.2.6 L298N-based custom H-bridge

For the demonstrator prototype, PWM is used to control the speed and direction of the vehicle's movement.


The custom H-bridge circuit board controlled by the common chip L298N, allows for two separate motors to be run independently, in either direction. For the demonstrator, only one output was used to power both motors together. The steering algorithm is able to handle external errors, such as uneven motor speeds, in case the motors do not perform equally. The circuit board was designed and machined by the course assistants, and then soldered by hand.

3.2.7 Power regulation using LM2596

An rechargeable 11.1V Li-Po battery was chosen as a power source because of its availability in the mechatronics lab and ability to supply high currents. Also the battery pack capacity of 2200 mAh allows the demonstrator vehicle to be run for longer periods of time.

Initially a own-designed linear voltage regulator were implemented. However due to overheating, a decision to use manufactured switching voltage regulator was taken. It is based on the LM2596 chip, set to output a steady 5.1V voltage. A screw potentiometer is used to set the output, independent of the higher voltage input. The output connectors on the regulator were connected to the hardware requiring approximately 5V.

The Raspberry Pi's power is supplied through a standard micro-USB cable, so the regulator output was also connected to a USB-A female port, which according to the USB specification should deliver close to 5V on pin number 1, and ground (0V) on pin number 4, as shown in image 3.4.



Pin	Signal	Color	Description
1	VCC	Red	+5V
2	D-	White	Data -
3	D+	Green	Data +
4	GND	Black	Ground

Figure 3.4. Pin out diagram for USB-A [5].

3.3 Software

Once the hardware assembly was completed, a script to detect lanes and steer accordingly in real time had to be programmed.

3.3.1 Operating system

The Raspberry Pi 3 is a system-on-a-chip (SoC) that can run the Linux-based operating system *Raspbian* [17] from a microSD memory card. Raspbian is a derivative of *Debian* [18] that has been tailored specifically for the Raspberry Pi computers and includes several useful packages right from the start. The operating system has primarily been ported by the efforts of Mike Thompson and Peter Green [17].

3.3.2 Image processing

Every captured image undergoes several different image processing steps in order to determine the current location of the vehicle in the lane. The steps that each image goes through are described below, and are inspired by George Sung's script [19] to detect lane markings in video footage. Lanes are detected based on contrasting colors.

Images are captured continuously by the Camera Module v2 at a resolution of 102 by 77 pixels, see figure 3.5, at a maximum of 40 frames per second, which is a hardware limitation in order to use the full area of the image sensor [20]. This low resolution was chosen to decrease the necessary processing, while still providing enough detail to detect the lane markings. The image is also only processed in black and white for the same reason.

In order to obtain a black and white image from the camera with minimal processing, the image is captured in the YUV-format [21], where Y is the *luma* lightness color information. This color coding standard was used during the transition from monochrome to color television, where monochrome televisions only utilized the Y-channel to represent imagery. The additional data from the U and V channels was discarded.

3.3. SOFTWARE



Figure 3.5. Image captured by Camera Module v2.

Further, the image was cropped to avoid including unnecessary background visuals, which further decreases the processing time. Using Canny Edge detection through the OpenCV package, binary edge images were generated from the camera input, as shown in figure 3.6.



Figure 3.6. Canny Edge Detection algorithms output.

Through global processing with Hough transform, the edge pixels were linked into meaningful lines, see figure 3.7 below. These lines are defined by the two endpoints, which are used in subsequent calculations.

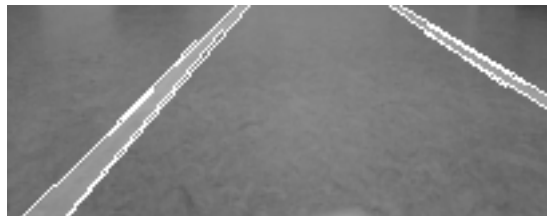


Figure 3.7. Hough lines on top of cropped original image.

The next step was to filter out unlikely lane lines based on their slope and position within the image. Lines with slopes above or below the threshold gradients of 12 or below 0.35 were disregarded, as they were unlikely to be actual lane markings. Furthermore lines positioned too far from the edges of the picture are not classified as lane markings. The leftmost point of potential right lines should be positioned within the rightmost 5/8ths of the image, and vice versa.

Finally after fitting the remaining lines, the lanes could be detected. The final output is shown in figure 3.8 below. If no lane lines could be detected, the last previous found lane was assumed to still be in place, which is comparable to when snow or other debris covers up the lane markings temporarily.



Figure 3.8. Detected road edges after image processing.

During the early stages of the project, each lane was approximated through linear interpolation of the end points of the lines. However, after realizing that short lines detected in the image could cause unexpected behavior, as seen in figure 3.9, a revised method was proposed.

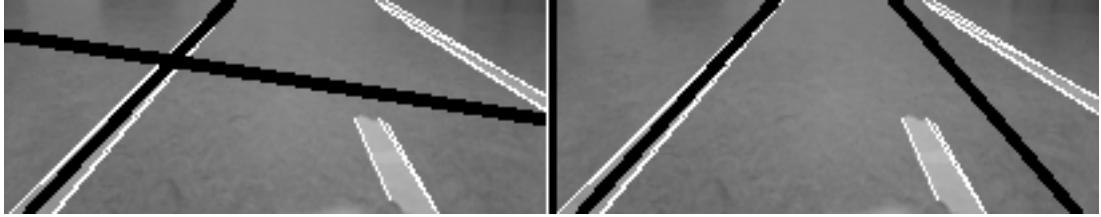


Figure 3.9. (left) Lane fitting using linear interpolation (right) Lane fitting focusing on average slope of detected lines.

The new method relies on using the detected lines' average slope and average intersect with the top of the image to approximate the lane position. A line which passes through two points can be expressed in the form of equation (3.1). The slope m of this line is calculated through equation (3.2). The y-intercept b is easily obtained from (3.1). From there, the x-intersect x_{int} can be calculated by setting $y = 0$ in (3.1), which yields equation (3.3). Since more than a single left or right

3.3. SOFTWARE

line may be detected in an image, an average line is determined by calculating the average slope m_{avg} (3.5) and average x-intercept $x_{int,avg}$ (3.4), where N is the number of lines. These two numbers define the black line markings shown in figure 3.9.

While the end result is still not quite accurate when stray lines are detected, the improvement is substantial compared to the results from the previous method.

$$y = mx + b \quad (3.1)$$

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad (3.2)$$

$$x_{int} = \frac{b}{m} \quad (3.3)$$

$$x_{int,avg} = \frac{1}{N} \sum_{i=1}^N x_{int,i} \quad (3.4)$$

$$m_{avg} = \frac{1}{N} \sum_{i=1}^N m_i \quad (3.5)$$

3.3.3 Lane position approximation

The goal was to keep the vehicle centered in the middle of the lane. This was achieved by finding where the lanes intersect with the bottom of the image by setting y in (3.1) to the correct y-coordinate, and then determining the midpoint x_{mid} between these intersections by using equation (3.6). The offset Δx (3.7), expressed in pixels, of this point from the middle of the image (where w_{img} is the image width) is used as error input for the PID-controller.

$$x_{mid} = \frac{x_{left} + x_{right}}{2} \quad (3.6)$$

$$\Delta x = x_{mid} - \frac{w_{img}}{2} \quad (3.7)$$

These points are visualized in figure 3.10.

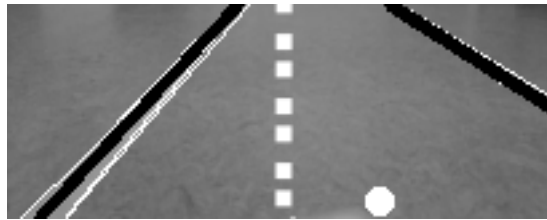


Figure 3.10. Captured image with center of image (dashed) and lane midpoint (dot) marked.

3.3.4 Control system

Based on the offset between the lane middle and the image's horizontal center, the car's positional error is logged in a vector, together with the time at which the image was taken. A PID-controller was iteratively tuned to maximize performance and to stay centered in the lane. Only the last 500 milliseconds of error values were used for the integral part of the controller, which also works to counteract integral windup behavior during long stretches of continuous error.

Suitable PID-values were chosen by adjusting the values so that the output from the PID-controller represented reasonable steering angles, as well as attempting to drive the vehicle through a test track. The controller's P-value was finally set to 0.8, the D-value to 0.1 to dampen oscillations, and the I-value to 0.05 for smoother steering.

The output of the PID-controller was used to set the angle of the wheels relative to the body of the vehicle. This steering angle was capped to $\pm 15^\circ$ through an approximate wheel-to-servo-angle conversion, as implemented with the code found in appendix A.

If the steering angle is sufficiently high, the speed of the vehicle can also be adjusted by lowering the PWM-signal to the H-bridge. This results in increased maneuverability, and regaining control of the vehicle when going through a sharp turn.

Chapter 4

Experiments

4.1 Experiment 1: Error of calculated lane position

The first experiment was designed to find an answer to the research question *What is the error of the calculated vehicle position for different positions on the road with the project's lane detection algorithm?*

4.1.1 Experiment description

The experiment should examine the error of calculated vehicle position depending on its actual position. The algorithm for detecting lane position ran several times while the car prototype was moved between equally spaced positions within the lane. Lane's width were set to 250 mm measured between the middle of tape strips. Road lanes were marked using 20 mm wide masking tape. Data was recorded at intervals of 62.5 mm from the left lane marking. The output data from the algorithm was compared to the demonstrator's actual position. A detailed experiment description and the raw data from three trials is available in appendix Band C respectively.

4.1.2 Experiment results

Results from the first experiment are presented in figures 4.1 and 4.2 below.

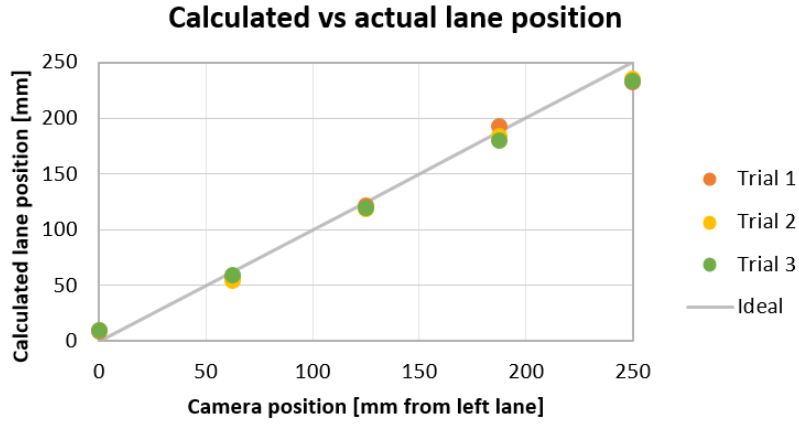


Figure 4.1. The vehicle’s calculated lane position compared to its actual position.

All the three trials show values that are close to the ideal results, i.e. a perfect match between the calculated position and the actual position. The error as a percentage of the lane width is insignificant in comparison to the errors produced by the conversion from lane to position to steering the vehicle in real time.

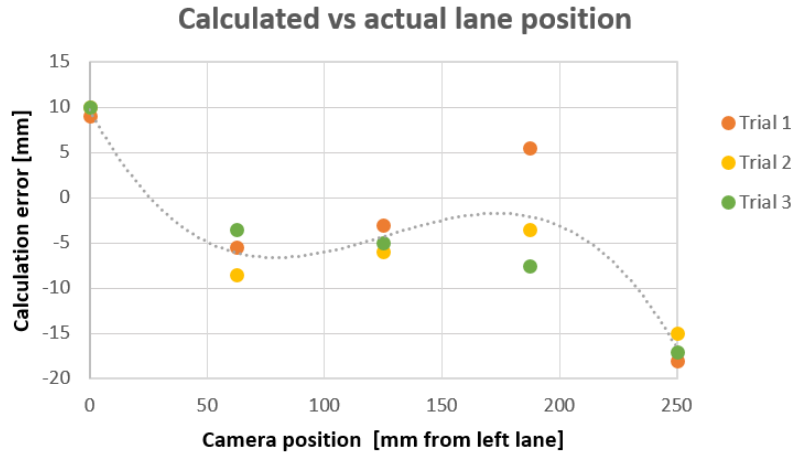


Figure 4.2. Error of calculated vehicle position compared to its actual position.

The maximum errors occur when the demonstrator’s camera is aligned with the lane markings (at 0 and 250 mm). Those are extreme cases which should ideally never occur during successful autonomous highway driving.

4.2. EXPERIMENT 2: POSITIONING ERROR DEPENDING ON CAMERA FRAME RATE

4.1.3 Experiment discussion

The relatively high errors at the edge placements is caused by the camera's field of view and slope filtering algorithm. Placing the car right on the edge would sometimes cause the opposite lane marking to not be detected. Our algorithm filtered out all the slopes that were not between two threshold values, and any line with an excessively high slope would not be recognized as a road edge. When only one side of a tape marking passes the line filter, this may lead to an error in the lane placement of roughly 10 mm, as the tape is 20 mm wide.

Sources of error:

- Human factor while placing the vehicle at specified position. It seems like the data points in figure 4.2 are systematically too low, perhaps due to observing the vehicle placement from a slight angle.
- Light in the room might impact the output because efficiency of Canny Edge detection varies depending on how well gradient difference is seen.
- Limited amount of data points.
- The low camera resolution limits maximal precision.
- Some camera settings are set to automatic mode, which is useful when dealing with diverse real world lighting scenarios.

4.1.4 Experiment conclusion

The maximum car positioning error relative to its actual position on a 250 mm wide road was 18 mm, or approximately 7% off. Error increases when the camera approaches the road edges. The results were better than expected and show that the lane detection algorithm performs well for lane positioning.

4.2 Experiment 2: Positioning error depending on camera frame rate

The second experiment was designed to find an answer to the research question *For a fixed vehicle speed, what is the maximum lane positioning error for different image sampling frequencies?*.

4.2.1 Experiment description

In short, the frame rate was set to a specific value, and the demonstrator's offset from the center of the lane at the end of a straight track was recorded using a video camera. The full experiment description can be found in appendix B. Frame rates from 2.5 Hz and increased in steps of 2.5 Hz, were investigated. This was done in order to see which frame rate was necessary for the demonstrator to reach the

end of the track, and there is a point after which increasing the frame rate has no further positive effect on the positioning error. Five trials were recorded for each frame rate, with an extra trial in case the demonstrator failed to make it to the end of the track. The data collection was terminated when the refresh rate could no longer be sustained for more than ten successive lane detection algorithm cycles, as the cycle time occasionally increased for a moment.

The vehicle speed was fixed by setting the motor speed to a fixed PWM-value. The lanes were separated by 30 cm. In order to achieve the desired frame rate, the script includes a repeated one millisecond delay until the period time has been achieved, assuming that the total necessary cycle time is below the requested period time. Any lane keeping failures were noted, but not used for the graphs.

4.2.2 Experiment result

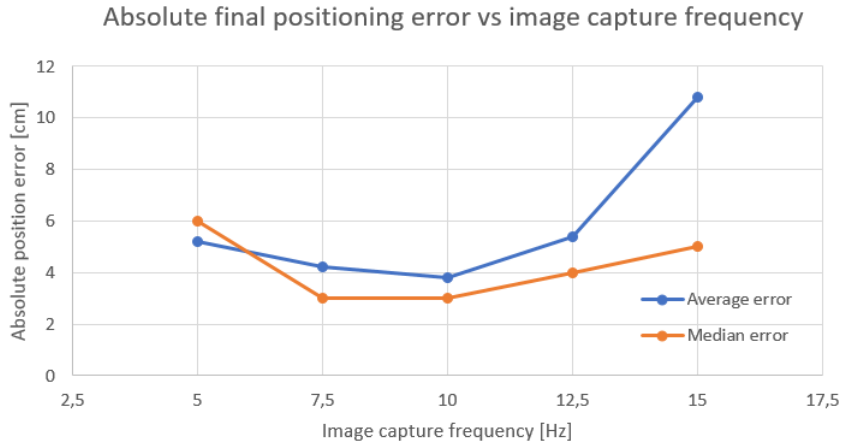


Figure 4.3. Average and median errors from experiment 2.

At 2.5 Hz, the demonstrator was unable to stay within the lane markings. At 17.5 Hz, the image capture and lane detection algorithm exceeded the specified period time. The general trend, as seen in figure 4.3 is that the optimal image capture frequency is close to 10 Hz, with the error increasing with both higher and lower frequencies. The minimum median absolute positional error was around 3 cm, at which all four wheels are well within the lane markings. The entire collected data can be found in appendix D. The code used for this experiment can be found in appendix A.

4.2. EXPERIMENT 2: POSITIONING ERROR DEPENDING ON CAMERA FRAME RATE

4.2.3 Experiment discussion

The results were often skewed by single trials with large positional errors; which is most noticeable at 15 Hz. This experiment should be re-designed to capture the worst positional error over a specified stretch of track. With this new methodology, going through the video data to find the positional errors without an automated approach would require excessive amounts of time. Another improvement to the experiment would be to record the positional error of the demonstrator while in the middle of a track, instead of where the track ends, as the missing lane markings may cause erratic behavior near the end of the track.

Occasionally, the total cycle time momentarily peaked to much higher values. The cause of this behavior was not determined. Disabling the automatic garbage collector was attempted, without any noticeable difference in performance. An increased frame rate should lead to more frequent adjustments of the steering angle, and thus, better positioning, which was not shown by the experimental results.

Sources of error:

- Momentarily longer period times due to unknown issue.
- The data point should instead be recorded before the end of the track.
- The human factor while measuring the offset. The camera resolution also limits maximal precision, but this is negligible compared to the human error.
- The tripod's legs may impact the image recognition capabilities of the vehicle.
- The steering suffered from some backlash, perhaps impacting the ability to steer.

4.2.4 Experiment conclusions

The autonomous lane positioning algorithm has no problem keeping the demonstrator within the lane, even at frame rates as low as 7.5 Hz. The average and median absolute positioning error were around 4 cm from the center of the 30 cm wide lane, which is acceptable as the demonstrator kept all four wheels within the lane markings, but there is potential to improve this.

Chapter 5

Discussion and conclusions

5.1 Discussion

The demonstrator is made from many different components, including several custom laser-cut, 3D-printed, and machined metal parts. Some of these parts had trouble with fit, leading to some backlash in the Ackermann steering and wheel axles. The overall construction was quite sturdy, except the wheels that exhibited some wobble that may have negatively impacted the steering capabilities of the vehicle. The tire-to-floor friction was quite low, so there is plenty of potential for improving the grip.

The performance on curved roads was not quantitatively recorded, but differing amounts of curvature were used when iteratively testing for suitable PID-values. When the demonstrator was turned too far from the lane direction, or the road curvature was too high, the demonstrator occasionally lost sight of both lanes. The field of view was determined by repeatedly taking photos and placing tape markings until they aligned with the edges of the photo. A top down view shows that the field of view is roughly 60 degrees, as shown in figure 5.1. A wide-angle lens could be used to capture more of the surroundings, should the project be developed further to handle sharper curves or other situations.



Figure 5.1. The field of view of the Camera Module v2.

5.2 Conclusion

The main research question which was investigated through this project was: *How can a control system be designed to keep an autonomous vehicle in the middle of a highway lane using a digital camera?*. The demonstrator's software implementation, exhibits one relatively simple and fast way of first calculating the vehicle position within a lane, and then using this data for a PID-controller which attempts to keep the vehicle centered in the lane. The demonstrator was able to stay within the lane, but had uneven performance.

Experiment 1 answered the first research subquestion: *What is the error of the calculated vehicle position for different positions on the road with the implemented lane detection algorithm?*. It was found that the developed algorithm calculated the lane positioning with a maximum error of 7% of the lane width, which occurs at extreme placements within the lane.

The second experiment was designed to answer the second research subquestion: *For a fixed vehicle speed, what is the maximum lane positioning error for different image sampling frequencies?*. The offsets determined are acceptable for low-risk use as they showed that the demonstrator was able to stay centered in the lane. For any high-risk use it would be recommended to use a sturdier demonstrator capable of capturing data at a higher frequency. Improved lane placement would be beneficial.

Chapter 6

Recommendations and future work

This bachelor thesis project has resulted in successfully constructing and programming a demonstrator vehicle capable of staying within the lane markings. While the results are a good starting point for other projects, further investigations are recommended in order to construct a vehicle with better performance. As a result, this may require more expensive hardware or more computationally complex algorithms.

6.1 Recommendations

Perhaps rapid changes in image data could lead to a measurement being ignored by the control system, as the wheels would occasionally twitch to excessively turned positions, which may also be caused by jitter.

The Camera Module v2 used for detecting the lane markings was not able to capture wide-angle images, which give a better view of the surroundings. Due to this limitation, the demonstrator had trouble detecting the lane upon heavy road curvature, for example for use in residential neighborhoods. This may require changes to the lane position calculation demonstrated in Experiment 1, as the so called fish-eye effect may warp the image at the edges of the visual field. Autonomous vehicle for use on public roads often include multiple wide-angle cameras, occasionally pointed in different directions. Alternatively, the camera could be mounted on a rotational platform that would be turned to always face the current direction of overall vehicle movement.

In order to increase overall performance and consistency, switching to the C programming language might be beneficial, as it is less abstracted than Python. OpenCV is available for this language as well. Increasing the raw performance of the microcontroller should also lead to more reliable driving. As an example, Tesla Motors uses top-of-the-line graphics processors to handle the heavy calculations, which offer much better raw number crunching performance compared to the Raspberry Pi 3.

6.2. FUTURE WORK

6.2 Future work

The project could be expanded by adding distance keeping, based on the ultrasonic distance sensor, in order to create a safer demonstrator. The demonstrator script described in this report ended the program if an obstacle was found to be too close to the front of the vehicle.

The lane position calculation can also be improved to better handle situations where several potential lane markings are detected. Some kind of filter to handle sudden changes in input data due to faulty measurement data, such as the measured distance suddenly decreasing to near zero or the calculated lane position shifting unexpectedly, might be helpful when building a more robust system.

If one were to drastically advance the capabilities of the vehicle, tracking the movements of nearby obstacles through computer vision methods, could be useful for avoiding collisions, in addition to using one or more ultrasonic distance sensors.

Bibliography

- [1] R. E. W. Rafael C. Gonzalez, “Digital image processing,” 2008.
- [2] S. Seidman, “Pwm,” [Online; accessed 22-March-2017]. [Online]. Available: <http://electronics.stackexchange.com/questions/128804/generating-a-variable-dc-signal-with-pwm>
- [3] “H bridge schematic diagram,” [Online; accessed 22-March-2017]. [Online]. Available: <http://pneuhiver.info/h-bridge-schematic-diagram/>
- [4] ElecFreaks, “Ultrasonic ranging module hc-sr04,” [Online; accessed 21-February-2017]. [Online]. Available: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>
- [5] modDIY, “Usb connectors and pinouts,” [Online; accessed 23-March-2017]. [Online]. Available: <https://www.moddiy.com/pages/USB-2.0-%7B47%7D-3.0-%7B47%7D-3.1-Connectors-%26-Pinouts.html>
- [6] X-toaster, “Pid tuning for toaster reflow ovens,” [Online; accessed 24-March-2017]. [Online]. Available: <https://www.x-toaster.com/resources/pid-tuning-for-toaster-reflow-oven/>
- [7] T. M. Company, “Full self-driving hardware on all cars,” [Online; accessed 13-February-2017]. [Online]. Available: <https://www.tesla.com/autopilot>
- [8] T. F. M. Company, “Model t facts,” [Online; accessed 14-February-2017]. [Online]. Available: <https://media.ford.com/content/fordmedia/fna/us/en/news/2013/08/05/model-t-facts.html>
- [9] NHTSA, “National motor vehicle crash causation survey - report to congress,” july 2008. [Online]. Available: <https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/811059>
- [10] L. L. Torkel Glad, “Reglerteknik - grundläggande teori,” 2006.
- [11] e. a. Hans Johansson, Per-Erik Lindahl, *Elektroteknik*. KTH Institutionen för Maskinkonstruktion, 2013.

BIBLIOGRAPHY

- [12] R. P. Foundation, “Raspberry pi 3 model b,” [Online; accessed 23-March-2017]. [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>
- [13] I. SSH Communications Security, “Ssh (secure shell) home page,” [Online; accessed 30-March-2017]. [Online]. Available: <https://www.ssh.com/ssh/>
- [14] R. P. Foundation, “Camera module,” [Online; accessed 23-March-2017]. [Online]. Available: <https://www.raspberrypi.org/documentation/hardware/camera/>
- [15] ServoCity, “S3003 servo,” [Online; accessed 19-February-2017]. [Online]. Available: <https://www.servocity.com/s3003-servo>
- [16] L. DAGU Hi-Tech Electronic Co., “Dg02s-a130gearmotor,” [Online; accessed 18-February-2017]. [Online]. Available: www.arexx.com.cn
- [17] P. G. Mike Thompson, “About raspbian,” [Online; accessed 29-March-2017]. [Online]. Available: <https://www.raspbian.org/RaspbianAbout>
- [18] S. Inc., “Debian: The universal operating system,” [Online; accessed 15-May-2017]. [Online]. Available: <https://www.debian.org/>
- [19] G. Sung, “Github repository: Road lane line detection,” [Online; accessed 09-May-2017]. [Online]. Available: https://github.com/georgesung/road_lane_line_detection
- [20] D. Jones, “Picamera docs: 6.1 camera modes,” [Online; accessed 09-May-2017]. [Online]. Available: <http://picamera.readthedocs.io/en/release-1.12/fov.html>
- [21] P. Magazine, “Definition of yuv,” [Online; accessed 09-May-2017]. [Online]. Available: <http://www.pcmag.com/encyclopedia/term/55165/yuv>

Appendix A

Python 2.7 Code

```
#####  
#                                                                 #  
#  Algorithm for autonomous lane positioning using #  
#  digital camera and a Raspberry Pi 3.          #  
#  Publication number: MDAB 645 MMK 2017:27       #  
#  By Stanislav Minko and Johan Ehrenfors.        #  
#  KTH, Mechatronics, 19 May 2017.               #  
#  Script: Main.py                               #  
#                                                                 #  
#####  
  
# Main algorithm for autonomous driving  
# coding: utf-8  
  
# Importing standard packages:  
import numpy as np  
import cv2  
import time  
import threading  
import picamera  
import RPi.GPIO as GPIO  
  
# Import own-written functions:  
from PictureProcessingFunction import *  
from LineDetectionFunction import *  
from NavigationFunction import *  
from DrivingFunction import *  
  
import globalVariables
```

```

def main():

    print "——_Initializing_goodkex_autonomous_vehicle._——"

    # Initiate global variables
    globalVariables.motorADC = 100
    globalVariables.distError = []
    globalVariables.posError = []
    globalVariables.emergencyStop = False # When True, all data \
                                           collection and driving will stop

    # Setup initial lane defaults
    L_m = - 1                                #  $y = mx + b$ 
    R_m = - L_m
    L_b = 75                                # Origin at top left corner.
    R_b = 75 - R_m * 205
    globalVariables.laneDefaults = [L_m, L_b, R_m, R_b]
    print "——Global_variables_instantiated.——"

    # Setup pins:
    GPIO.setwarnings(False)                 # Recommended state: False
    GPIO.setmode(GPIO.BCM)

    # H-bridge
    enA = 25
    in3 = 8
    in4 = 7

    GPIO.setup(enA, GPIO.OUT)               # enableA (motor A), green
    GPIO.setup(in3, GPIO.OUT)              # Input 3 H-bridge, blue
    GPIO.setup(in4, GPIO.OUT)              # Input 4 H-bridge, purple

    GPIO.output(in3, GPIO.HIGH)             # Set forward driving as default
    GPIO.output(in4, GPIO.LOW)
    motorA = GPIO.PWM(enA, 500)            # GPIO.PWM (channel, frequency)

    globalVariables.motorADC = 0
    motorA.start(globalVariables.motorADC) # Set start DC to 0%

    # Camera part
    camera = picamera.PiCamera()

    camera.resolution = (102, 77)          # Roughly 1/32th of max resolution
    camera.framerate = 40

```

APPENDIX A. PYTHON 2.7 CODE

```

time.sleep(2)                                # Let the camera warm up

# Ultrasonic
globalVariables.TRIG = 23
globalVariables.ECHO = 18

GPIO.setup(globalVariables.TRIG, GPIO.OUT)
GPIO.setup(globalVariables.ECHO, GPIO.IN)

GPIO.output(globalVariables.TRIG, GPIO.LOW)

# Servo
servoPin = 24

GPIO.setup(servoPin, GPIO.OUT)

servo = GPIO.PWM(servoPin, 50)
straightDC = calcAngleToDC(0)
servo.start(straightDC)    # Wheels turned straight ahead
print "——Hardware setup complete (GPIO pins, etc) ——"

# Initiating refresh rate
refreshHz = 20                # Default value
FixedRefreshRate = False # Keep True while conducting experiment 2
output = PositionLogger(refreshHz, FixedRefreshRate)

camera.start_recording(output, 'yuv')
print "——Image recognition is running. ——"

# Begin ultrasonic distance measurement loop
# It updates global variable(distError) with new data
threading.Thread(target = DistanceLogger).start()
print "——Thread 2——Distance sensor is running. ——"

# Keep driving while program is running
print "——The autonomous vehicle is up and running. Enjoy! ——"
while not globalVariables.emergencyStop:
    globalVariables.motorADC = Driving(servo, globalVariables.motorADC,
    motorA, enA, in3, in4)

camera.stop_recording()
servo.stop()
motorA.stop()

```

```

time.sleep(0.5)           # To avoid GPIO error
GPIO.cleanup()

class PositionLogger(object):
def __init__(self, refreshRate, requireRate):
self.requireRefreshRate = requireRate
self.refreshHz = refreshRate
self.refreshPeriod = 1.0/refreshRate
self.consecutiveTooSlow = 0
self.lastDataPoint = time.time()

def write(self, buf):
# This function will be called once for each output frame.
# buf is a bytes object containing the frame data in YUV420
# format, of which the Y-channel is used for a monochrome image

y_data = np.frombuffer(buf, dtype=np.uint8, \
count=128*80).reshape((80,128))
# Reads 128*80 ints from buffer, ordered in a matrix.

grey = y_data[:77,:102]

# Crop top of the image:
original_image_height = grey.shape[0]
cropPercentage = 0.55 # Vertical crop percentage
masked_img = grey[int(cropPercentage *
original_image_height):int(original_image_height - 5)]
# Cropping only y-axis. [y-from :y-to]
# Cut 5 px from the bottom to avoid ultrasonic parts

# Edge detection using CannyEdgeDetection function:
binImage = Canny(masked_img)

# Find lines in image using HoughLinesFunction:
lines = houghLines(binImage)

# Relevant image information:
image_height, image_width = binImage.shape[:2]
imageCenterPx = round(image_width / 2)
imgInfo = [image_height, image_width, imageCenterPx]

# Calculate the lane position
if lines == None: # Stop the vehicle if no lines were detected.

```

APPENDIX A. PYTHON 2.7 CODE

```

globalVariables.emergencyStop = True
else:
    left_b , left_m , right_b , right_m = lineDetection(lines , imgInfo)
    # Lane detection (lines + img info in, lanes out)
    Navigation(left_b , left_m , right_b , right_m , imgInfo)
    # Calculates error and logs to global

# Maintain a specied refresh rate if required (experiment 2)
if self.requireRefreshRate and ((time.time() - self.lastDataPoint)
> self.refreshPeriod):
    self.consecutiveTooSlow += 1
    maxTooSlow = 10 # Maximum number of consecutive cycles exceeding self
    print "Warning: \_Too\_slow!",self.consecutiveTooSlow,"/",maxTooSlow
    if self.consecutiveTooSlow > maxTooSlow:
        print "CRITICAL\_ERROR: \_Refresh\_rate\_could\_not\_be\_sustained!"
        globalVariables.emergencyStop = True
    elif self.requireRefreshRate:
        self.consecutiveTooSlow = 0
    while time.time() - self.lastDataPoint < self.refreshPeriod:
        time.sleep(0.001)

    self.lastDataPoint = time.time()
    print "\_" # Blank line

def flush(self):
    # Called at the end, does nothing
    pass

def DistanceLogger():
    while not globalVariables.emergencyStop:
        # Trigger
        GPIO.output(globalVariables.TRIG, GPIO.HIGH)
        time.sleep(0.00001) # Short burst
        GPIO.output(globalVariables.TRIG, GPIO.LOW)

    # Read time until echo registered
    validMeasurement = False

    timeTriggered = time.time()
    while GPIO.input(globalVariables.ECHO) == 0:
        pulse_start = time.time()
        if time.time() - timeTriggered > 0.5:
            print "NOTE: \_Ultrasonic\_was\_not\_triggered\_within\_0.5s.\_Continuing"
            break

```



```

while GPIO.input(globalVariables.ECHO) == 1:
    validMeasurement = True
    pulse_end = time.time()

    if validMeasurement:
        pulse_duration = pulse_end - pulse_start

        time.sleep(0.1)

        distance = pulse_duration * 17150          # [cm]
        timestamp = time.time()

        safeDistance = 25 # [cm] used as desired value for PID braking
        distanceError = distance - safeDistance
        if distanceError < 0:
            print "ERROR: Distance too close - collision risk!"
            globalVariables.emergencyStop = True

# Save data to global vector
globalVariables.distError.append([distanceError, timestamp])

if __name__ == "__main__":
    main()

#####
#
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3.           #
# Publication number: MDAB 645 MMK 2017:27        #
# By Stanislav Minko and Johan Ehrenfors.         #
# KTH, Mechatronics, 19 May 2017.                 #
# Script: PictureProcessingFunction.py             #
#                                                  #
#####

# PictureProcessingFunction
import numpy
import cv2

def Canny(maskedImage, lowThresh = 30, highThresh = 170):
    """ From image and spec. threshold value generates image edges. """
    edges = cv2.Canny(maskedImage, lowThresh, highThresh)

```

```

return edges

def houghLines(binImage):
    """ Takes binear image data, the edges, as input.
        It links edges and returns them as lines """
    imgHeight = binImage.shape[0]
    minLength = imgHeight / 3
    maxGap = imgHeight / 3.5
    rho = 1
    theta = numpy.pi / 180 # Angular resolution in radians of Hough-grid
    houghThres = imgHeight / 5 # Minimum number of intersections in Hough
    lines = cv2.HoughLinesP(binImage, rho, theta, houghThres, \
numpy.array([]), minLineLength = minLength, maxLineGap = maxGap)
return lines

#####
#
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3. #
# Publication number: MDAB 645 MMK 2017:27 #
# By Stanislav Minko and Johan Ehrenfors. #
# KTH, Mechatronics, 19 May 2017. #
# Script: LineDetectionFunction.py #
# #
#####

# LineDetectionFunction

import globalVariables
import numpy
import cv2

def lineDetection(lines, imageInformation):

    # Image information:
    image_height = imageInformation[0]
    image_width = imageInformation[1]
    imageCenterPx = imageInformation[2]

    # Change "lines" format and calculate slope for each line:
    detectedLines, slopes = slopeCalculator(lines)

    # Determine left and right lines:
    rightLines, leftLines = lineSorting(imageCenterPx, \

```

```

detectedLines, slopes)

# Determine actual road edges using linear interpolation:
right_m, right_b, foundRight = lineFit(rightLines, image_height)
left_m, left_b, foundLeft = lineFit(leftLines, image_height)

# Keeping last known value as default for the lanes:
# Order of defaults:
# globalVariables.laneDefaults = [L_m, L_b, R_m, R_b]
if foundLeft:
# Set new defaults, in case the line is lost in next photo.
globalVariables.laneDefaults[0] = left_m
globalVariables.laneDefaults[1] = left_b
else:
# Use last successfully detected left lane
left_m = globalVariables.laneDefaults[0]
left_b = globalVariables.laneDefaults[1]

if foundRight:
globalVariables.laneDefaults[2] = right_m
globalVariables.laneDefaults[3] = right_b
else:
right_m = globalVariables.laneDefaults[2]
right_b = globalVariables.laneDefaults[3]

if not (foundLeft or foundRight):
print "ERROR: _Lost_sight_of_both_lanes_-_not_safe_to_continue!"
globalVariables.emergencyStop = True

return left_b, left_m, right_b, right_m


def slopeCalculator(lines):
""" slopeCalculator function takes lines found
using Hough transform as input and calculates the
slope of each corresponding line. Lines are now
stored in differnt format for easier future calculations. """

slopes = []
newLines = []

for line in lines[0]:
x1, y1, x2, y2 = line      # line = [[x1, y1, x2, y2]]

```

APPENDIX A. PYTHON 2.7 CODE

```

# Calculate slope
if (x1 == x2):           # Corner case, avoiding division by 0
    slope = float(99999)   # Practically infinite slope
elif x1 < x2:
    slope = float((y2 - y1)) / (x2 - x1)
else:
    slope = float((y1 - y2)) / (x1 - x2)

slopes.append(slope)
newLines.append(line)

return newLines, slopes

def lineSorting(imageCenterPx, detectedLines, slopes):
    """ Function takes lines and their corresponding slopes as input.
    It sorts lines into right and left lines. Right/left lane lines
    must have positive/negative slope, and be on the right/left half
    of the image. """

    rightLines = []
    leftLines = []

    numRight = 0
    numLeft = 0

    slopeMinThres = 0.35
    slopeMaxThres = 12
    borderOffset = 0.25

    for i in xrange(len(slopes)):
        x1, y1, x2, y2 = detectedLines[i]

        # Lines are sorted as left and right depending on their slope and
        # position in picture. Right/Left lane only counts if it passes
        # specified threshold value and placed on the correct side of the
        # image(borderOffset)
        # Right lines:
        if (slopes[i] > slopeMinThres) and (min(x1, x2) > (1 - \
borderOffset) * imageCenterPx) and (slopes[i] < slopeMaxThres):
            rightLines.append(detectedLines[i])
            numRight += 1
        # Left lines
        elif (slopes[i] < -slopeMinThres) and (max(x1, x2) < \
(1 + borderOffset) * imageCenterPx) and (slopes[i] > \

```

```

-slopeMaxThres):
    leftLines.append(detectedLines[i])
    numLeft += 1

return rightLines, leftLines

def lineFit(Lines):
    slopes = []
    xIntercepts = []

    if len(Lines)>0:
        for line in Lines:
            x1, y1, x2, y2 = line

            # Find slope of line
            mLine = float(y2-y1) / float(x2-x1)      # dy/dx

            # Determine x-intercept (at top of image, y=0) using one point
            bLine = y1 - mLine * x1 # based on (x1,y1), calculated slope m
            xInter = -bLine / mLine # Solve for x when y=0

            # Log slope and x-intercept
            slopes.append(mLine)
            xIntercepts.append(xInter)

            # Calculate average slope (to be used as m of detected lane)
            m = sum(slopes) / len(slopes)
            # Calculate average x-intercept of found (potential) lane markings
            avgXInter = sum(xIntercepts) / len(xIntercepts)
            # Calculate b (y-intercept, x=0)
            b = -m * avgXInter
            foundLine = True
        else:
            print("Warning! _Line_was_NOT_found.")
            foundLine = False
            m, b = 0, 0 # To be able to return
    return m, b, foundLine

#####
#
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3. #
# Publication number: MDAB 645 MMK 2017:27 #
# By Stanislav Minko and Johan Ehrenfors. #

```

APPENDIX A. PYTHON 2.7 CODE

```
# KTH, Mechatronics, 19 May 2017. #
# Script: NavigationFunction.py #
# #
#####

# Navigation Function

import time
import globalVariables

def Navigation(left_b, left_m, right_b, right_m, imgInfo):
    """ Calculate the pixel offset between the middle of the
        lane and the camera's center """

    # Image data:
    image_height = imgInfo[0]
    imageCenterPx = imgInfo[2]

    # Determine position of the point between the lane lines
    y2_L = image_height - 1
    x2_L = int((y2_L - left_b) / left_m)

    y2_R = image_height - 1
    x2_R = int((y2_R - right_b) / right_m)

    lane_position_px = (x2_R + x2_L) / 2
    car_position_px = imageCenterPx - lane_position_px

    currentPosTime = time.time()
    currentPosError = car_position_px

    globalVariables.posError.append([currentPosError, currentPosTime])

#####
# #
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3. #
# Publication number: MDAB 645 MMK 2017:27 #
# By Stanislav Minko and Johan Ehrenfors. #
# KTH, Mechatronics, 19 May 2017. #
# Script: DrivingFunction.py #
# #
#####
```

```

# Driving function
# coding: UTF-8

from scipy import misc
import RPi.GPIO as GPIO
import numpy
import math
import time

import globalVariables

def Driving(servo, motorADC, motorA, enA, in3, in4):
    # motorADC is used to keep track of the old value

    # Steering:
    # Using error calculate output PID val, which is used for steering.
    steeringOutput = PID(globalVariables.posError, \
        KP=0.80, KI=0.05, KD=0.100)
    maxSteeringPWM = setSteering(servo, steeringOutput)

    newMotorADC = maxSteeringPWM

    # Ramp up if speeding up, limited to max +1% per run
    if newMotorADC > motorADC + 1:
        newMotorADC = motorADC + 1

    motorA.ChangeDutyCycle(newMotorADC)

    return newMotorADC

def integralCalc(dataVec, tStart):
    """ Calculated using the cumtrapz-method. Will integrate
        from tStart to final value. """

    # "Backwards" linear search from most recent data
    # point to find which start index to use
    startIndex = len(dataVec) - 1
    while dataVec[startIndex-1][1] >= tStart:
        if startIndex == 0:      # Do not go out of bounds
            break
        else:
            startIndex -= 1

    index = startIndex

```

APPENDIX A. PYTHON 2.7 CODE

```

totalArea = 0

# Check that next element in list is always accessible:
while index < len(dataVec) - 1:
    avgHeight = (dataVec[index][0] + dataVec[index + 1][0]) / 2 # Error
    width = dataVec[index + 1][1] - dataVec[index][1] # Time
    area = avgHeight * width
    totalArea += area
    index += 1

return totalArea

def setSteering(servo, angle):
    max_angle = 20
    min_angle = -max_angle

    # Do not permit excessive steering.
    if angle > max_angle:
        angle = max_angle
    elif angle < min_angle:
        angle = min_angle

    # Give pum to servo to point wheels.
    dutycycle = calcAngleToDC(angle)
    servo.ChangeDutyCycle(dutycycle)

    # Calculate maximum speed allowed when steering at this angle
    # Map angle (-30 to 30) to radians (-pi to pi)
    radians = angle * 2 * numpy.pi/360

    # Max PWM (12+5=17) close to zero radians, low PWM at extreme
    # wheel angles. Factor 4 for greater braking impact.
    maxSteeringPWM = 15 + 5 * math.cos(4 * radians)
    return maxSteeringPWM

    # Calculate Duty Cycle from input angle.
    def calcAngleToDC(steering_angle):
        servo_angle = 2*steering_angle

        servo_min_angle = -90
        servo_max_angle = 90
        servo_min_DC = 5
        servo_max_DC = 10
        DC = numpy.interp(servo_angle, [servo_min_angle, servo_max_angle], \

```



```

[servo_min_DC , servo_max_DC]) # Maps servo angle to dutycycle

return DC

def PID(globalErrorVec , KP=1,KI=0,KD=0):
    """ Calculates the PID output signal by using the
        provided argument data. """
    errorVec = globalErrorVec # In case data changes while running PID
    # Make sure that the data is in a suitable format:
    if len(errorVec)==0:
        return 0 # Assume OK until first data collected
    elif len(errorVec)==1:
        errorVec.insert(0,[0 , time.time() - 1]) # Add leading zero

    # Find necessary recent errors and time stamps
    lastError = errorVec[len(errorVec) - 1][0]
    secondToLastError = errorVec[len(errorVec) - 2][0]

    lastTime = errorVec[len(errorVec) - 1][1]
    secondToLastTime = errorVec[len(errorVec) - 2][1]
    timeOffset = 0.5

    # Calculate the three parts
    proportional = KP * lastError
    integral = KI * integralCalc(errorVec , lastTime - timeOffset)
    denominator = float(lastTime) - secondToLastTime
    if denominator == 0:
        denominator = 0.01
    derivative = KD * (float(lastError) - secondToLastError) \
    / denominator # Rise over run to approximate derivative

    # Add up the three parts and return
    output = proportional + integral + derivative
    return output

#####
#
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3. #
# Publication number: MDAB 645 MMK 2017:27 #
# By Stanislav Minko and Johan Ehrenfors. #
# KTH, Mechatronics, 19 May 2017. #
# Script: globalVariables.py #

```

APPENDIX A. PYTHON 2.7 CODE

```

#
#####

# File to store global variables

# Global variables:
global distError, posError, motorADC, emergencyStop
global TRIG, ECHO, laneDefaults

#####
#
# Algorithm for autonomous lane positioning using #
# digital camera and a Raspberry Pi 3. #
# Publication number: MDAB 645 MMK 2017:27 #
# By Stanislav Minko and Johan Ehrenfors. #
# KTH, Mechatronics, 19 May 2017. #
# Script: Experiment1.py #
#
#####

# coding: utf-8
# Experiment 1. Examine the error of calculated vehicle position
# depending on actual vehicle position.
import numpy as np
import cv2
import time
import picamera
from picamera.array import PiRGBArray

from PictureProcessingFunction import *
from LineDetectionFunction import *

def pixelToMillimeter(pixel_offset, lane_width):
    millimeter = 250 * pixel_offset / lane_width
    return millimeter

def main():

# Camera part
# The camera's default resolution is the display's resolution.
camera = picamera.PiCamera()

camera.resolution = (102, 77) # Roughly 1/32th of max res
camera.framerate = 40

```

```

time.sleep(1)                                # Let camera settle

raw_input("Place the car... Press enter to capture the image...")

# Camera handling:
rawCapture = PiRGBArray(camera)
camera.capture(rawCapture, format = 'bgr', use_video_port = True)
img = rawCapture.array
# Coverting img to greyscale:
grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

original_image_height, original_image_width = img.shape[:2]

# X coordinate of center of image
img_center_px = round(original_image_width / 2)

""" Image processing below. """

# Crop the image to remove unnecessary "landscape" features
cropPercentage = 0.45                        # Vertical crop percentage
masked_img = grey[int(cropPercentage*original_image_height):\
int(original_image_height - 5)]
# NOTE: its img[y: y + h, x: x + w]
image_height, image_width = masked_img.shape[:2]

# Edge detection using CannyEdgeDetection function:
edges = Canny(masked_img)

# Find lines in image using HoughLinesFunction:
lines = houghLines(edges)

""" Finding and printing actual road edges below. """

# Calculate slope:
newLines, slopes= slopeCalculator(lines)

# Determine left and right lines:
rightLines, leftLines= lineSorting(img_center_px, newLines, slopes)

# Determine road edges by fitting lines useing lineFitFunction:
right_m, right_b, foundRight = lineFit(rightLines)
left_m, left_b, foundLeft = lineFit(leftLines)

```

APPENDIX A. PYTHON 2.7 CODE

```

""" Working with navigation point below. """

# Determine position of the point between the lane lines

y2_L = image_height - 1
x2_L = int((y2_L-left_b)/left_m)

y2_R = image_height - 1
x2_R = int((y2_R-right_b) / right_m)

lane_position_px = (x2_R + x2_L) / 2
# lane_pos_px can be negative if 1 of the lanes got very low slope.
lane_width_px = abs(x2_R-x2_L)
car_position_px = img_center_px - lane_position_px
# Negative means we are to the left of lane center.
car_position_mm = pixelToMillimeter(car_position_px, lane_width_px)

print "Car_position_pixel:", car_position_px, "_px."
print "The_car_is_at_position:", car_position_mm, \
"mm_from_the_center."
print "The_car_is_at_position", 125 + car_position_mm, \
"mm_from_the_left_lane_marking."

if __name__ == "__main__":
    main()

```

Appendix B

Experiment description

A detailed description of Experiment 1 and 2, including material list and a method to follow.

***Test 1.** Examine the error of calculated vehicle position depending on actual vehicle position.*

Variables:

Dependent: Calculated position

Independent: Known position in lane

Controlled: Camera and image recognition settings (some set to automatic adjustment), static vehicle, fixed lane width

Hypothesis: The average absolute error of calculated vehicle position will increase linearly while moving away from middle of the lane.

Material list:

- Ruler with millimeter markings
- White masking tape, 20 mm wide
- Non-reflective linoleum floor
- Vehicle prototype with custom software
- Computer with SSH software OR Raspberry connected to screen and keyboard

Method:

1. Find clear linoleum floor.
2. Set up right road edge on the floor using masking tape.
3. Measure, using the ruler, and place left road edge, parallel to the right road edge. The distance between the middle of the tape strips should be 250 mm. The tape strips should be 50 cm long.

APPENDIX B. EXPERIMENT DESCRIPTION

4. Run the script.
5. Place the car prototype so that camera is centered with left road edge. All wheels should be parallel to the road edges.
6. Use the computer to run test script (attached above) and take note of the output (calculated position from the left lane in millimeters).
7. Move the car, parallel to the road edges, to the right by 62.5 mm to the next position.
8. Repeat 5 and 6 four times for readings at 0, 62.5, 125, 187.5, and 250 mm from the middle of the left tape strip.
9. Repeat steps 4 to 7 for a total of three trials.
10. Plot the calculated position points against the actual position.
11. Plot the average error of the calculated positions against the actual position.

Specifications: Camera is fixed relative to the car. Pictures are taken and processed with exactly the same settings. Check the attached code for Canny threshold, Hough threshold, image resolution, iso and other parameters. The vehicle's speed is 0 m/s. The ruler is fixed to the floor.

Sources of error:

- Human factor while placing the vehicle at specified position.
- Light or other objects in the room might impact the output.
- Limited amount of data points.
- Camera resolution limits maximal precision.
- Some camera settings are set to automatic, as this will be the case in real world scenarios.

Test 2. *Find what is the lane positioning error for different image sampling frequencies?*

Variables:

Dependent: Actual position in lane

Independent: Desired (centered) lane position (attempted via control system)

Controlled: Video camera height, vehicle camera and image recognition settings, fixed PWM-motor speed, fixed lane width

Hypothesis: The lane positioning error will decrease with a higher image sampling frequency. There may be a low threshold frequency at which the vehicle's control

system is not able to stay on the road. There may also be a high threshold frequency at which the performance is no longer improved due to physical time delays.

Material list:

- Digital video camera with tripod stand.
- White masking tape, 20 mm wide.
- Non-reflective floor
- Prototype vehicle with custom software
- Ruler with millimeter markings
- Computer with SSH

Method:

1. Find a clear area of non-reflective floor.
2. Set up right road edge on the floor using masking tape. The tape strip should be 2.5 m long.
3. Measure, using the ruler, and place the left road lane using masking tape, parallel to the right road edge. The distance between the middle of the tape strips should be 300 mm.
4. Mount the digital video camera on the tripod, and place the tripod above the end of the track. The beginning of the track will be used to allow the vehicle to accelerate until it reaches a stable speed. The video camera should be positioned at a height of 100 cm and angled straight down at the floor.
5. Place the car prototype so that camera is centered in the lane. All wheels should be parallel to the road edges.
6. Connect to the vehicle using SSH.
7. Start filming using the camera at the maximum possible resolution.
8. Run the attached script and set the image sampling frequency by entering a value as prompted by the script (initially 2.5 Hz). The car will attempt to stay in the lane while maintaining a fixed PWM-signal to the motors.
9. Stop filming once the car gets off the track (two or more wheels outside of the lane markings), or has reach the end of the track. If the car does not make it to the end repeatedly, discard the data and proceed to a higher frequency (step 11). If this occurs regularly, decrease the PWM-signal to the motors and restart the experiment.

APPENDIX B. EXPERIMENT DESCRIPTION

10. Repeat steps 7 to 9 until five trials have been recorded at the specified image sampling frequency. Rename the video clips in the format of Test2-XHz-TrialY where X and Y vary depending on the trial and frequency used.
11. Repeat steps 7 to 10 at the next image sampling frequency. Advance from 2.5 Hz in steps of 2.5 Hz, as far as the program allows it (the program will print a warning message if the frequency is too high to be processed in real time).
12. Turn off the vehicle prototype.
13. Open the video files in a video viewing program.
14. Step through the frames and determine where the vehicle is positioned as it reaches the end of the track. Use a ruler to measure (on the screen) the maximum offset of the camera sensor from the center of the lane, perpendicular to the lane markings. Compare this to the known distance between the lane markings (30 cm) to get an approximate error in centimeters. Log this data for all the trials for each frequency being investigated.
15. Set up a graph comparing the average and median absolute positioning error for each frequency where the vehicle managed to follow the track.

Specifications: Camera is fixed relative to the car. Pictures are taken and processed with exactly the same settings. Check the code for Canny threshold, Hough threshold, image resolution, and other parameters. Vehicle's initial speed is 0 m/s. Track lacks curvature. Observation camera films from above, parallel to the ground.

Sources of error:

- Human factor while measuring the offset. (Camera resolution also limits maximal precision, but this is negligible compared to the human error in this case)
- The tripod's legs may impact the image recognition capabilities of the vehicle.
- Only a single data point is logged per run.
- The speed may vary depending on how the vehicle reacts to the track, for example by steering hard.

Appendix C

Raw Data from Experiment 1

Known position from left lane [mm]	Trial 1 [mm]	Trial 2 [mm]	Trial 3 [mm]
0	9	10	10
62.5	57	54	59
125	122	119	120
187.5	193	184	180
250	232	235	233

Appendix D

Raw Data from Experiment 2

Vehicle offset from center at finish line [cm]						
Frame rate [Hz]	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	(Trial 6)
2.5	X	X	X	X	X	X
5	1	3	6	6	10	-
7.5	6	3	1	8	3	-
10	3	6	3	5	X	2
12.5	2	7	4	13	1	-
15	3	5	X	8	4	35
17.5	S	S	S	S	S	S

Note: S - Refresh rate not sustained, X - Demonstrator out of bounds