In Situ Detection of Road Lanes Using Raspberry Pi

Ashwani Chahal
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Chahal, Ashwani, "In Situ Detection of Road Lanes Using Raspberry Pi" (2018). All Graduate Theses and Dissertations. 7051.
https://digitalcommons.usu.edu/etd/7051

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact dylan.burns@usu.edu.
IN SITU DETECTION OF ROAD LANES USING RASPBERRY PI

by

Ashwani Chahal

A thesis submitted in partial fulfillment
of the requirements for the degree
of
MASTER OF SCIENCE
in
Computer Science

Approved:

Vladimir Kulyukin, Ph.D.  Nicholas Flann, Ph.D.
Major Professor  Committee Member

Xiaojun Qi, Ph.D.  Mark R. McLellan, Ph.D.
Committee Member  Vice President for Research and
Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah
2018
ABSTRACT

In Situ Detection of Road Lanes using Raspberry Pi

by

Ashwani Chahal, Master of Science
Utah State University, 2018

This thesis portrays in detail the algorithm intended to recognize lane lines on streets and highways in various climate conditions. Lane detection technique, to a noteworthy degree, improves the security in the independent vehicles. A self-propelled or a self-sufficient vehicle is an independent machine that detects the natural conditions and settles on an appropriate human-like decision. The purpose of this research is to find a feasible solution to detect lanes on a low voltage computer that can be easily powered in a regular auto vehicle.

This research uses a 1.3 rev PiCamera, to capture the real time video, attached with a RaspberryPi 3.0 to process each captured frame. The data collection is done while driving in a Jeep Wrangler on the course of multiple days with different daylight conditions depending on the weather (rainy, sunny, or snowy). The RaspberryPi, appended to a wooden board, is put on the vehicle’s dashboard; and the PiCamera, facing the road, is settled to the auto’s windshield to gather the data. A 30-minute video with 10 frames for each second gives enough data(18,000 frames) to run the experiments to calculate algorithm’s performance.

The algorithm described in this research uses concepts of Computer Vision, Open CV, along with employment of basic linear algebra to compute and draw the lane lines in the captured frames. The code for the algorithm is written using python 2.7. This algorithm
provides approximately 97% accurate detection of both lane lines in all the weather conditions mentioned above.

This research provides a significant contribution to the field of autonomous driving. Lane line detection, if utilized as a part of a vehicle, can help the driver while changing lanes, bringing about a superior and a more secure driving experience. The detailed approach portrayed in this postulation provides a fairly accurate and an economical solution to detect the lane lines.
PUBLIC ABSTRACT

In Situ Detection of Road Lanes using Raspberry Pi
Ashwani Chahal

A self-driven car is a vehicle that can drive without human intervention by making correct decisions based on the environmental conditions. Since the innovation is in its beginning periods, totally moving beyond the human inclusion is still a long shot. However, rapid technological advancements are being made towards the safety of the driver and the passengers. One such safety feature is a Lane Detection System that empowers vehicle to detect road lane lines in various climate conditions.

This research provides a feasible and economical solution to detect the road lane lines while driving in a sunny, rainy, or snowy weather condition. An algorithm is designed to perform real time road lane line detection on a low voltage computer that can be easily powered in a regular auto vehicle.

The algorithm runs on a RaspberryPi computer placed inside the car. A camera, attached to the vehicle’s windshield, captures the real time images and passes them to the RaspberryPi for processing. The algorithm processes each frame and determines the lane lines. The detected lane lines can be viewed on a 7 inch display screen connected to the Raspberry Pi. The entire system is mounted inside a Jeep Wrangler to conduct the experiments and is powered by the vehicle’s standard charger of 12V-15V power supply. The algorithm provides approximately 97% accurate detection of road lane lines in all weather conditions.
ACKNOWLEDGMENTS

I would like to offer my sincere gratitude to my advisor, Dr. Vladimir Kulyukin, who has helped me all through my exploration with his important recommendations. He generally urged me to attempt distinctive methodologies keeping in mind the end goal to locate the most ideal solution. His patience, inspiration, and immense knowledge has enormously contributed to the accomplishment of this research.

Besides my advisor, I recognize my appreciation to the rest of my committee members, Dr. Xiaojun Qi and Dr. Nicholas Flann, for perusing my reports and for providing their insightful remarks.

I likewise thank my fellow labmates for continually participating in the stimulating discussions identified with this exploration.

Ashwani Chahal
CONTENTS

Page

ABSTRACT ................................................................. iii
PUBLIC ABSTRACT ...................................................... v
ACKNOWLEDGMENTS .................................................... vi
LIST OF TABLES ........................................................ viii
LIST OF FIGURES ....................................................... ix
1 INTRODUCTION ....................................................... 1
2 RELATED WORK ....................................................... 4
3 LANE DETECTION ALGORITHM ..................................... 6
  3.1 Overview ....................................................... 6
  3.2 Data Collection ................................................ 7
  3.3 Data Preprocessing ............................................. 8
    3.3.1 Masking .................................................. 8
    3.3.2 Grayscaling .............................................. 9
    3.3.3 Gaussian Blurring ....................................... 10
  3.4 Detecting Gradient Change - Edge detection using Sobel Derivative ... 11
  3.5 Computing Hough lines based on the detected edges ................. 12
  3.6 Filtering resulting lines ..................................... 15
    3.6.1 Filtering left and right lines based on the slope value ...... 15
    3.6.2 Filtering lines based on the intercept value ................ 17
    3.6.3 Draw Lines .............................................. 18
  3.7 Using History to prevent false negatives ........................ 19
4 EXPERIMENTS AND RESULTS ....................................... 23
  4.1 Data Set Numbers .............................................. 23
  4.2 Experiments .................................................. 24
  4.3 Comparison with the previous work .......................... 29
  4.4 Results and Analysis ........................................ 31
5 CONCLUSION AND FUTURE WORK ................................. 33
REFERENCES ............................................................. 35
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Performance results of experiments conducted on different weather conditions - Rainy, Sunny, and Snowy data.</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Results of comparison with the previous work for input set 1</td>
<td>30</td>
</tr>
<tr>
<td>4.3 Results of comparison with the previous work for input set 2</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Results of comparison with the previous work for input set 3</td>
<td>31</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A RaspBerryPi Touchscreen display (green box) mounted on the car’s dashboard. Pi Camera facing the road is shown inside the red box.</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Hardware setup for data collection: A Pi camera(red box), attached to the Raspberry Pi (green box), facing the road.</td>
<td>7</td>
</tr>
<tr>
<td>3.3</td>
<td>Resized original image.</td>
<td>9</td>
</tr>
<tr>
<td>3.4</td>
<td>Image after Masking.</td>
<td>10</td>
</tr>
<tr>
<td>3.5</td>
<td>Image after gray scaling.</td>
<td>11</td>
</tr>
<tr>
<td>3.6</td>
<td>Image after Gaussian Blurring.</td>
<td>12</td>
</tr>
<tr>
<td>3.7</td>
<td>Image with detected edge pixels.</td>
<td>13</td>
</tr>
<tr>
<td>3.8</td>
<td>A line in a parametric form.</td>
<td>14</td>
</tr>
<tr>
<td>3.9</td>
<td>Image with detected lines using Probabilistic Hough Transform.</td>
<td>15</td>
</tr>
<tr>
<td>3.10</td>
<td>Finding slope of a line.</td>
<td>16</td>
</tr>
<tr>
<td>3.11</td>
<td>Finding y intercept of a line.</td>
<td>18</td>
</tr>
<tr>
<td>3.12</td>
<td>Results after filtering based on slope and intercept values.</td>
<td>18</td>
</tr>
<tr>
<td>3.13</td>
<td>Drawing resulting lines in the original image.</td>
<td>19</td>
</tr>
<tr>
<td>3.14</td>
<td>Image used to store left lane history.</td>
<td>20</td>
</tr>
<tr>
<td>3.15</td>
<td>Drawing left lane line using history.</td>
<td>21</td>
</tr>
<tr>
<td>3.16</td>
<td>Image used to store right lane history.</td>
<td>21</td>
</tr>
<tr>
<td>3.17</td>
<td>Drawing right lane line using history.</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of successful detection of lane lines in rainy data set.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of False Positive in rainy data set.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of successful detection of lane lines in sunny data set.</td>
<td>27</td>
</tr>
</tbody>
</table>
4.4 Example of False Positive in sunny data set ........................................ 27
4.5 Example of successful detection of lane lines in snowy data set .......... 28
4.6 Example of False Positive in snowy data set ..................................... 29
4.7 Real time lane detection (a) .............................................................. 32
4.8 Real time lane detection (b) .............................................................. 32
CHAPTER 1
INTRODUCTION

A self driving car, also known as autonomous car or a robotic car is a vehicle that is capable of sensing and analyzing the environment and making suitable decisions without requiring human inputs or commands. Also referred to as driver-less car, a self driving car can drive itself by detecting the surroundings using multiple techniques such as radar, GPS, Computer Vision etc. [1]

Computerized reasoning in autos has highlighted in numerous exploration tasks and trials have been led since the 1980s when first models for self-ruling autos were displayed via Carnegie Mellon College’s Navlab. From that point forward, there have been a lot of innovative headways in the field of self-governing vehicles and Navlab 11, the most recent auto by CMU’s Route Research facility, is a 2000 Jeep Wrangler introduced with several pieces of equipment such as GPS, magnetometers, proximity laser censors, and omni-directional camera [2]. Stanley, an autonomous car created by Stanford University in cooperation with Volkswagon won first driver-less car racing challenge known as ‘DARPA Grand Challenge’ in 2005 [3]

Numerous studies have been led about vehicle robotization and Driver Assistance Systems. A portion of the highlights of a self-governing vehicle incorporates Automatic Cruise Control, Automatic Parking, Collision Avoidance as well as Lane Departure Warning systems. Aside from the said, the potential advantages of autonomous cars incorporate decreased framework costs, expanded security, expanded consumer satisfaction, and a critical lessening in car accidents. [4]. Some in favor of autonomous vehicles additionally trust that conveying robotics to the car will kill more typical wrongdoings like insurance scams and vehicle burglary. [5]

On account of above mentioned benefits, many countries have taken a step forward to bring self-driving cars on the roads for public. UK, in 2014, announced that driver less cars
will be allowed on public roads and soon a prototype called 'Lutz pod' was launched as UK’s first driver-less car [6] [7]. Moreover, In 2017, first autonomous vehicle was demonstrated at Christchurch Airport [8]. However, despite significant amount of benefits, there exist some foreseeable challenges in completely accepting self-driven vehicles. A few opposers believe that widespread adoption of driver-less cars will bring dearth of driving related jobs and will also compromise with the passenger’s safety [9] [10] [11]. It is also believed that an autonomous car will likely lead to the loss of privacy and increased risks of hacking attacks and terrorism [12] [13].

It is understood that the discrepancy between people’s beliefs of the necessary government intervention may cause a delay in accepting autonomous cars on the road [14]. This will keep the drivers in control of the car for some years to come. Therefore, it is important to seek solutions to make their driving experience safer and easier. This thesis presents one such solution - Real time road lane lines detection in different weather conditions. A real time vision-based lane detection system can be used to assist the driver in locating the lanes or warning the driver if the vehicle goes out of lane.

In this thesis, a Lane Detection algorithm is presented for real-time detection of road lane lines in various climate conditions. The algorithm runs on a 7 inch display attached to a RaspberryPi 3.0 computer with a Pi camera installed. The entire system is placed on the car’s dashboard. The Pi Camera is placed inside the car fixed to the windshield to capture the real time video while driving. The display screen and the RaspberryPi computer is powered by a 12V-to-5V car power supply port. Lane Detection algorithm presented in this thesis implements the concepts of Computer Vision - like gradient change and probabilistic Hough transform, to detect the lane lines in the road images. Moreover, certain filtering techniques, such as filtering based on slope and intercept values, are used to determine the exact location of road lane lines. The algorithm is implemented in Python 2.7 with OpenCV 3.0.

The entire thesis is organized as follows. In Chapter 2, other work related to the lane detection or autonomous driving is discussed. Chapter 3 contains the details of the
algorithm used to detect road lane lines in this research. Chapter 4 presents the statistics of the results from the experiments conducted during this research. Chapter 5 presents the conclusion and talks about the future work.
CHAPTER 2
RELATED WORK

Enhancing autonomous driving, especially by performing accurate road lane line detection, has been a major research interest in past few years. Several researchers have performed real time vision based lane detection using various techniques. This chapter discusses few of those research works and related algorithms in brief.

Yuan et al. [15] established a novel method for tracking road lanes for vision-guided autonomous vehicle navigation. They use an inverse perspective mapping to remove the perspective from the camera and then detect the edges of the road lanes from the inverse perspective mapping images. An algorithm for 'particle filtering' is used to compute the likelihood between all the particles with the edge images, henceforth estimating the three parameters of the real state of road lane lines. This lane detection method is tested in real road images to achieve reliable results.

P. Mandlik and A.B. Deshmukh [16] presented a Lane Departure Detection System (LDWS) in accordance with Advanced Diver Assistance System (ADAS) to warn the driver when the vehicle tends to depart it’s lanes. LDWS is based on the lane identification and tracking algorithm and uses OpenCV implementations of ’Canny Edge Detection’ and ’Hough Transform’ to detect vehicle lane departure on a Raspberry Pi. The experiments are conducted on the images captured using a toy vehicle with a USB camera, ‘Intex IT-305WC webcam’, mounted on top. Out of many straight lines detected by Hough Transform, the longest straight lines are identified as the lane lines. The results are collected using Intel Core i3 1.80 Gbhz processor.

K. H. Lim et al. [17] constituted a real-time implementation of lane detection and tracking system to localize lane boundaries and estimate a linear-parabolic lane model. For experiments, a CCD camera is used to capture video frames and stored in video port buffer of a TMS320DM642 DSP board. Horizon localization is used to discern the sky from road
in the input image. To recognize lane markings, road pixels are removed from the road region by performing lane analysis. Once lane boundaries are located, the conceivable edge pixels are scanned to ceaselessly to obtain the lane model. A Linear-parabolic model is used to construct the geometry of the lane and the model parameters are updated with Kalman filtering.

X. Du et al. [18] proposed a robust vision-based methodology to deal with challenges during lane detection like shadows, shifting lighting conditions, faded-away lane lines, etc. The methodology incorporates four key advances: Using a ridge detector, the line line pixels are pooled. Then, using a noise filtering mechanism, noisy pixels are removed. After removing noises, a sequential Random Sample consensus is employed to ensure that each lane line in the image is collected correctly. In the final step, a technique: parallelism reinforcement is employed to enhance the accuracy of the model. The model is also fit to localize vehicles with respect to the road lane lines.

Q. Truong and B.R. Lee [19] used the principal approach to detect road boundaries and lanes using a vision-based system in the vehicle. The paper presented a methodology to detect and estimate the curvature of lane boundaries. A vector-lane-concept and non-uniform B-spline (NUBS) interpolation method is used to construct the boundaries of road lane lines. Based on the lane boundary, the curvature of left and right lane boundaries are calculated. For experimental purposes, images are captured using a monocular camera. Experimental results are based on real world road images, as presented in the paper.

A.A.M. Assidiq et al. [20] exhibited a vision-based lane detection approach to handle frequently varying lighting and shadow conditions. The framework acquires the frontal view of the road by utilizing a camera mounted on the vehicle. A couple of hyperbolas, fitting to the edges of the lane, are paired with Hough Transform to extract the lane lines. It has also been asserted that the proposed lane detection framework can be used on both painted as well as unpainted roads, as curved and straight roads. The experimental results demonstrate that the proposed framework can be utilized for real time requirements.
CHAPTER 3
LANE DETECTION ALGORITHM

3.1 Overview

This section describes in detail the algorithm designed for the real time road lane lines detection. The algorithm is divided into five stages - data collection, data preprocessing, gradient change detection using Sobel, line detection using Hough, and line filtering using slope and intercept values. Subsequent sections describe each stage in detail.

The hardware used for this research is shown in the Fig. 3.1. A wooden board is used to hold the touchscreen display and the Raspberry Pi computer together. This wooden board is placed on the vehicle's dashboard. A Pi camera, attached to the Raspberry Pi, is held straight with the help of a tripod and is placed facing the road against the vehicle's windscreen as shown in Fig. 3.2.

Fig. 3.1: A RaspBerryPi Touchscreen display (green box) mounted on the car’s dashboard. Pi Camera facing the road is shown inside the red box.
Fig. 3.2: Hardware setup for data collection: A Pi camera (red box), attached to the Raspberry Pi (green box), facing the road

Algorithm 3.1 Pi Camera Settings

Input:
None

Output:
Video in h264 format

Begin
  camera = PiCamera()
  camera.resolution = (1920, 1080)
  camera.framerate = 10
  camera.iso = 200
  camera.sharpness = 10
  camera.video_stabilization = True
  sleep(1800)
End

3.2 Data Collection

The data set is collected over multiple drives during varying climate conditions. A Pi camera is used to capture 10 frames per second over an interval of 30 minutes (i.e. 1800 seconds). The camera settings for capturing the data is given in the algorithm section below. Algorithm 3.1.

For experiments, the collected videos are saved to the local Pi in the memory space
Algorithm 3.2 Frame Collection

Input:
- path to .mp4 video

Output:
- .jpg format images

Begin
\[ fps = 10 \]
\[ cap = cv2.VideoCapture(VideoPath) \]
While(cap.isOpened()):
\[ ret, frame = cap.read() \]
\[ cv2.imwrite(path, frame) \]
End

provided by the attached memory card. The videos collected in h264 format are converted into mp4 and individual frames are extracted from mp4 videos and are converted to jpg images. Algorithm 3.2.

The results from the above algorithm are stored in a directory on the local machine and each file is read individually to be processed and saved in the output directory.

3.3 Data Preprocessing

The images collected by extracting frames using the above algorithm are iterated over individually and re-sized. A resized original image can be seen in Fig. 3.3.

\[ dim = (400, 255) \]
\[ resized = cv2.resize(input_image, dim, interpolation = cv2.INTER_AREA) \]

The resizing is performed using cv2.resize() method of OpenCV. [21] The resized image then undergoes masking, grayscaling and Gaussian blurring.

3.3.1 Masking

It is understood that in any image containing road (or lane lines), the road surface area is present in the bottom half of the image. Using this knowledge, the region of interest is decided.
Algorithm 3.3 Masking

Input:
A row number above which the image will be discarded

Output:
A masked image

Begin

def region_of_interest(img, vertices):
    cv2.fillPoly(mask, vertices, (0, 0, 0))
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image

End

The entire $y$ section (vertical height) of the image is horizontally cut into the half and the region lying above that horizontal line is discarded. This was performed using $cv2.fillPoly()$ and $cv2.bitwise_and()$ method of OpenCV. This method takes a row number as an input and creates a mask such that the region of interest is only below the given row. Masking is illustrated in Algorithm 3.3 and Fig. 3.4.

3.3.2 Grayscale

In the future stages of the presented algorithm, edge detection and Hough transformation, the image is required to be converted into a single color scale, also called as grayscale. Therefore, the region of interest extracted in the previous stage during masking, is rendered
Algorithm 3.4 Grayscale

Input:
A colored image with three color channels - Blue, Green, and Red

Output:
A grayscale image

Begin
    \text{def grayscale(img) :}
    \text{return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)}
End

to grayscale image as shown in Fig. 3.5.

This is performed using \text{cv2.cvtColor()} method of OpenCV. This method takes a colored image as an input (RGB) and returns a grayscale image as an output. Further processing is done on the resulting grayscale image.

Gray scaling is illustrated in Algorithm 3.4.

3.3.3 Gaussian Blurring

Once the grayscaling is done, noise reduction is performed on the image. A gaussian kernel is used to blur/smoothen the image. It is done with the help of an OpenCv function, \text{cv2.GaussianBlur().} The width and height of the kernel are required to be defined which should be positive and odd(both are defined as 5 in our case). We should also specify the standard deviation in X and Y direction, sigmaX and sigmaY respectively (defined as 0 in...
Algorithm 3.5 Gaussian Blurring

Input:
- A grayscaled image

Output:
- A grayscaled image with reduced noise.

Begin
    def gaussian_blur(img, kernel_size):
        return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)
End

This method takes a grayscaled image as an input and returns a blurred image as an output depending on the kernel size. Further processing is done on the resulting grayscale blur(smooth) image. Gaussian blurring is illustrated in Algorithm 3.5 and Fig. 3.6.

3.4 Detecting Gradient Change - Edge detection using Sobel Derivative

One way to detect the edges in an image is to traverse the image horizontally(say left to right) and for every pixel, calculate the amount of change in the gradient value. This process marks every pixel that shows a gradient change along the x-axis(including the pixels related to the lane lines).

While moving across the image from left to right direction, the pixel intensity changes
significantly whenever a road lane line is encountered. These changes can be expressed by using Sobel derivatives. A higher degree of change in the gradient value accounts for an edge.

The horizontal changes $\varphi_x$ are computed by convolving Image (I) with a Kernel (K) with odd size. [24] [25] For example for a kernel size of 3, $\varphi_x$ would be computed as:

$$\varphi_x = K * I$$

$$K = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}$$

Using above method, all the edges present in the given image are detected and marked for the selected region of interest. This is performed using an OpenCV method cv2.Sobel(). Edge detection using Sobel Operator is illustrated in Algorithm 3.6. An image with detected edges is shown in Fig. 3.7.

### 3.5 Computing Hough lines based on the detected edges

Hough Transform takes a linear line equation and represents it in the parametric form.
Algorithm 3.6 Edge detection using Sobel Derivative - x direction

Input:
A grayscaled blurred image

Output:
A black image with only edge pixels marked.

Begin

def abs_sobel_thresh(img, thresh_min, thresh_max):
    abs_sobel = np.absolute(cv2.Sobel(img, cv2.CV_64F, 1, 0))
    scaled_sobel = np.uint8(255 * abs_sobel/np.max(abs_sobel))
    output = np.zeros_like(scaled_sobel)
    output[(scaled_sobel >= thresh_min) and (scaled_sobel <= thresh_max)] = 1
    return output

End

Fig. 3.7: Image with detected edge pixels

Any shape can be detected if it is represented in the mathematical form. A linear line equation is given as-

\[ y = mx + c \]

While in parametric form, a line is represented as -

\[ \rho = x\cos\theta + y\sin\theta \]

where \( \rho \) is the perpendicular distance from origin to the line, and \( \theta \) is the angle formed by this perpendicular line and horizontal axis. A line in parametric form is shown in Fig. 3.8.
Any line can be represented in the form of $\rho$ and $\theta$. The motivation behind this method is to discover the instances belonging to a line shape by a voting procedure. This voting method is performed in a parameter space. [26]

In the given algorithm, this is performed by using an OpenCV method `cv2.HoughLinesP()`. This is illustrated in Algorithm 3.7.

HoughLinesP stands for Probabilistic Hough Transform, which is an optimization of Hough Transform discussed above. In addition to minimum number of votes it also takes two more parameters minimum length of line and maximum allowed gap.

Hough parameters used in the presented algorithm are defined below:

- $\rho = 1$
- $\theta = \pi/180$
- $threshold = 50$
- $min\_line\_len = 60$ pixels
- $max\_line\_gap = 400$ pixels

Where, $threshold$ is the minimum number of votes required to consider the given edge as a line, $min\_Line\_Length$ is the minimum length of line (line segments shorter than this are rejected) and $max\_Line\_Gap$ is the maximum allowed gap between line segments to treat them as single line.

An image with detected Hough lines is shown in Fig. 3.9.
Algorithm 3.7 Probabilistic Hough Transform to detect lines in an image containing edges

Input:
- An image with detected edges

Output:
- Array of vertices for all detected lines

Begin
\[ \text{lines} = \text{cv2.HoughLinesP}(\text{img}, \text{rho}, \text{theta}, \text{threshold}, \text{np.array}([]), \text{minLineLength} = \text{min.line.len}, \text{maxLineGap} = \text{max.line.gap}) \]
End

Fig. 3.9: Image with detected lines using Probabilistic Hough Transform

3.6 Filtering resulting lines

As shown in figure 3.9, multiple lines detected by Probabilistic Hough Transform do not belong to road lane lines. To discard the unwanted lines, two filtering methods are used - filtering using slope value and filtering using intercept value. These methods are described in the subsections below.

3.6.1 Filtering left and right lines based on the slope value

In every frame collected using Pi camera from inside the vehicle, the lane lines lie in the bottom half of the picture. The left lane line makes a positive angle with the x-axis (positive slope value) while the right lane line makes a negative angle with the x-axis (negative slope value).
For the collected dataset, depending on the angle at which camera is placed inside the car, the left lane lines make an angle between +30 to +70 degrees with the bottom edge (x-axis) while the right lane lines make an angle between -30 to -70 degrees with the bottom edge (x-axis). Hence, the lines with angle values less than the minimum or more than the maximum limit are discarded.

A line equation in form of slope is shown in Fig. 3.10.

The angle value for each detected line is found by using the line equation.

\[ y = mx + c \]

where \( m \) is the slope and \( c \) is the intercept value that line makes with the y axis. \( m \) can be found as:

\[ m = \frac{y_2 - y_1}{x_2 - x_1} \]

\( m \) is the value of tangent of the angle that line makes with the x-axis. Hence, \( m \) can also be written as -
Once the value of $m$ is found, the following equation can be used to calculate the value of angle that the line makes with the x-axis:

$$
\theta = \arctan(m)
$$

In the given algorithm the lines with the angle values less than 30 and greater than 70 are discarded and the rest are marked as left lane lines. Similarly, the lines with the angle values less than -70 and greater than -30 are discarded and the rest are marked as right lane lines.

### 3.6.2 Filtering lines based on the intercept value

This section describes another filtering method to filter the lane lines after they are marked as left or right lane lines based on their slope values.

The limitation of filtering lane lines based on the slope is that the filtering method considers the parallel lines(lines with the same slope values) as the potential lane lines. Therefore, another parameter is taken into account to determine the exact lane line position. This parameter is the y intercept value - $c$ in $y = mx + c$.

To remove the false positives (non lane lines parallel to the lane lines), the y-intercept value for each resulting line is calculated.

A line equation in form of intercept is shown in Fig. 3.11.

The equation of a line is:

$$
y = mx + c
$$

Hence, knowing the value of slope $m$, the value of $c$ can be found by -

$$
c = y - mx
$$
Once the value of $c$ for each resulting line is determined, the median of all the values is considered and the lines with intercept values farther to the median value are discarded. In the given algorithm, the lines with intercept values within 50 pixels left or right to the median value are considered as the lane lines.

The result after filtering process is presented in Fig. 3.12.

3.6.3 Draw Lines

This section describes the last stage of road lane line detection algorithm. Once the edges (gradients change using Sobel operator) on the gray-scaled blur image are detected and transformed into potential lane lines using Probabilistic Hough Transform,
Algorithm 3.8 Drawing left and right lane lines on the original image

**Input:**
- Line coordinates after filtering false positives

**Output:**
- Resulting image with drawn lane lines

Begin

\[
\text{for } x_1, y_1, x_2, y_2 \text{ in LeftLane :}
\]

\[\text{cv2.line}(\text{img}, (x_1, y_1), (x_2, y_2), [0, 255, 0], 3),\]

\[
\text{for } x_1, y_1, x_2, y_2 \text{ in RightLane :}
\]

\[\text{cv2.line}(\text{img}, (x_1, y_1), (x_2, y_2), [255, 0, 0], 3),\]

End

---

Fig. 3.13: Drawing resulting lines in the original image

filtering is performed as mentioned in the above section. The resulting lines are marked in the original picture. A resulting image with lane lines drawn over it can be seen in Fig 3.13. The left lane lines (with positive slope values) are marked in ‘green’ and the right lane lines (with negative slope values) are marked in ‘red’ using \textit{cv2.line}(). This is performed using an OpenCv method \textit{cv2.line()} as illustrated in Algorithm 3.8.

### 3.7 Using History to prevent false negatives

This section discusses the way to remove the false negatives from the resulting images. False negatives are the cases where

False negatives are the images which contain road lane lines but the presented algorithm fails to detect them. Such cases are averted by taking into account the history of detected
Since the lane line detection is performed real time on multiple frames per second, the lane lines in the current frame of consideration lie very close to the lane lines detected in the previous frame. Therefore, the history of last detected lane lines (left and right) from the previous frame is stored. In case a false negative is encountered, lane lines from the history are used and drawn on the current frame of consideration.

The history is updated if the lane lines are detected in the current frame of consideration. Therefore, after every frame, the history is either updated or it is used to draw the lane lines on the current frame.

In the given algorithm, every time history is used, the lines are drawn with yellow color. The image used to store the left lane line history is shown in Fig 3.14 and the image where the left lane is drawn using history is shown in Fig. 3.15. Similarly, the image used to store the right lane line history is shown in Fig 3.16 and the image where the right lane is drawn using history is shown in Fig. 3.17

![Fig. 3.14: Image used to store left lane history](image-url)
Fig. 3.15: Drawing left lane line using history

Fig. 3.16: Image used to store right lane history
Fig. 3.17: Drawing right lane line using history
CHAPTER 4
EXPERIMENTS AND RESULTS

4.1 Data Set Numbers

This section describes the data sets which are used to carry out the experiments on.

The experiments are conducted on 3 different data sets each belonging to a different weather condition. Each data set is collected in form of a video while driving in a sunny, rainy, and a snowy day. Frames are extracted from each video and the algorithm is then tested on each extracted frame.

Second column of table 4.1 contains the total number of frames against different weather conditions.

Accuracy

Accuracy of the algorithm is tested by running the algorithm on each and every frame taken into consideration. The number of images where both the lanes are detected, are counted manually. These numbers are present in column ‘Both lines’ of table 4.1. The number of images where either one lane line is detected or which contains some false positives are counted separately (failed cases). These numbers are present in column ‘False positives’ of table 4.1.

The accuracy of the lane detection algorithm is calculated as -

\[ \text{Accuracy\%} = \frac{\text{Number of images with both lane lines detected}}{\text{Total images}} \times 100 \]

The combined result of all three data sets is shown in the last row of the table 4.1. The experiments were conducted on 26,425 images and both lanes were accurately detected on 25,962 images.

The combined accuracy of the algorithm comes out to be 98.24%.
Table 4.1: Performance results of experiments conducted on different weather conditions - Rainy, Sunny, and Snowy data.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Total images</th>
<th>Both lines</th>
<th>False positives</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rainy</td>
<td>3800</td>
<td>3711</td>
<td>89</td>
<td>97.65%</td>
</tr>
<tr>
<td>Sunny</td>
<td>6139</td>
<td>5996</td>
<td>143</td>
<td>96.67%</td>
</tr>
<tr>
<td>Snowy</td>
<td>16486</td>
<td>16255</td>
<td>231</td>
<td>98.59%</td>
</tr>
<tr>
<td>Combined</td>
<td>26425</td>
<td>25962</td>
<td>463</td>
<td>98.24%</td>
</tr>
</tbody>
</table>

4.2 Experiments

Experiments are run on a Linux machine with python 3.0 and OpenCV 2.7 installed. The data is collected using a RaspberryPi 3.0 attached with a rev 1.3 PiCamera.

There are certain parameters set for the algorithm to perform optimum edge detection and hence the lane lines. These parameters are kept same for all three data sets. The minimum and maximum angle for left and right lane lines are considered as:

- $minimumLeftLaneAngle = 30$ degrees
- $maximumLeftLaneAngle = 70$ degrees
- $minimumRightLaneAngle = -70$ degrees
- $maximumRightLaneAngle = -30$ degrees

All the lines which lie in the above mentioned interval are considered to be lane lines. Moreover, there are several parameters which are set for PiCamera to do the optimum frame collection in different weather conditions. These parameters are discussed in subsequent subsections below for respective weather condition.

Rainy Data

A 30 minutes video is collected while driving in rainy weather from inside the car using a PiCamera. The PiCamera settings are modified to accommodate for overcast conditions. Below are the parameters which are used to capture the data set for rainy conditions:
\texttt{camera.resolution} = (1920, 1080) \\
\texttt{camera.framerate} = 10 \\
\texttt{camera.iso} = 200 \\
\texttt{camera.sharpness} = 10 \\
\texttt{camera.video\_stabilization} = True \\

The frame collection resulted in 3800 frames to run the experiment on. Both lane lines are accurately in 3711 images out of 3800, however, 89 images contain false positives. The total time taken to run the algorithm over 3800 images comes out to be around 418 seconds. The performance of the algorithm for rainy data set is computed as below:

$$FPS = \frac{3800}{418} = 9.09$$

Hence, the performance of the algorithm in terms of time is 9.09 \textit{FPS} (Frames per second), which means that approx. 9 frames can be detected in one second. Few examples of how the success and the failure results look like are provided in Figs. 4.1 and 4.2.

Fig. 4.1: Example of successful detection of lane lines in rainy data set
Sunny Data

A 30 minutes video is collected while driving on a sunny day from inside the car using a PiCamera. PiCamera saturation and ISO levels are used as defaults because of the sufficient day light during sunny weather condition. The PiCamera settings are mentioned below:

\[
\begin{align*}
\text{camera.resolution} &= (1920, 1080) \\
\text{camera.framerate} &= 10 \\
\text{camera.video_stabilization} &= \text{True}
\end{align*}
\]

The frame collection resulted in 6139 clean frames to run the experiment on. Both lane lines are accurately in in 5996 images out of 6139, however, 143 images contain false positives. The total time taken to run the algorithm over 6139 images comes out to be around 677 seconds. The performance of the algorithm for rainy data set is computed as below:

\[
FPS = \frac{6139}{677} = 9.06
\]

Hence, the performance of the algorithm in terms of time is 9.06 FPS (Frames per second), which means that approx. 9 frames can be detected in one second. Few examples of how the success and the failure results look like are provided in Figs. 4.3 and 4.4.
Snowy Data

A 30 minutes video is collected while driving in snowy weather condition from inside the car using a PiCamera. The PiCamera settings are modified to accommodate for overcast conditions. Below are the parameters which are used to capture the data set for rainy conditions:
camera.resolution = (1920, 1080)
camera.framerate = 10
camera.iso = 250
camera.sharpness = 15
camera.video_stabilization = True

The frame collection resulted in 16486 clean frames to run the experiment on. Both lane lines are accurately detected in 16255 images out of 16486, however, 231 images contain false positives. The total time taken to run the algorithm over 16486 images comes out to be around 1784 seconds. The performance of the algorithm for rainy data set is computed as below:

\[
FPS = \frac{16486}{1784} = 9.24
\]

Hence, the performance of the algorithm in terms of time is 9.24 FPS (Frames per second), which means that approx. 9 frames can be detected in one second. Few examples of how the success and the failure results look like are provided in Figs. 4.5 and 4.6.

Fig. 4.5: Example of successful detection of lane lines in snowy data set
4.3 Comparison with the previous work

The performance and accuracy of the presented algorithm is tested against the previous work done in the same laboratory - Greedy HAAR Spiker [27].

Three random data sets containing 120 images are selected out of total 26,425 images. The resolution of each image is set to 400x255 pixels and the execution time and accuracy of both the algorithms are compared.

Comparison using Input Set 1

A data set of 120 random images is selected and both the algorithms are run separately to compare the performance and the accuracy. The current algorithm accurately detects both lane lines in 114 images out of 120 and accurately detects at least one lane line in all the images if the input data set. On the other hand, the Greedy HAAR Spiker is able to detect both lane lines in 58 images and at least one lane line in 102 images. The accuracy of the lane detection algorithm presented in this thesis comes out to be 95% for detecting both lanes and 100% for detecting at least one lane as compared to 48% and 85% respectively for Greedy HAAR Spiker. See table 4.2.
Table 4.2: Results of comparison with the previous work for input set 1

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total images</th>
<th>Both lanes</th>
<th>Accuracy</th>
<th>At least one lane</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Work</td>
<td>120</td>
<td>58</td>
<td>48%</td>
<td>102</td>
<td>85%</td>
</tr>
<tr>
<td>Current Work</td>
<td>120</td>
<td>114</td>
<td>95%</td>
<td>120</td>
<td>100%</td>
</tr>
</tbody>
</table>

Comparison using Input Set 2

A data set of 120 random images is selected and both the algorithms are run separately to compare the performance and the accuracy. The current algorithm accurately detects both lane lines in 116 images out of 120 and accurately detects at least one lane line in all the images if the input data set. On the other hand, the Greedy HAAR Spiker is able to detect both lane lines in 59 images and at least one lane line in 100 images. The accuracy of the lane detection algorithm presented in this thesis comes out to be 96.67% for detecting both lanes and 100% for detecting at least one lane as compared to 49.16% and 83.33% respectively for Greedy HAAR Spiker. See table 4.3.

Table 4.3: Results of comparison with the previous work for input set 2

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total images</th>
<th>Both lanes</th>
<th>Accuracy</th>
<th>At least one lane</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Work</td>
<td>120</td>
<td>59</td>
<td>49.16%</td>
<td>100</td>
<td>83.33%</td>
</tr>
<tr>
<td>Current Work</td>
<td>120</td>
<td>116</td>
<td>96.67%</td>
<td>120</td>
<td>100%</td>
</tr>
</tbody>
</table>

Comparison using Input Set 3

A data set of 120 random images is selected and both the algorithms are run separately to compare the performance and the accuracy. The current algorithm accurately detects both lane lines in 114 images out of 120 and accurately detects at least one lane line in all the images if the input data set. On the other hand, the Greedy HAAR Spiker is able to detect both lane lines in 63 images and at least one lane line in 102 images. The accuracy of
the lane detection algorithm presented in this thesis comes out to be 95% for detecting both lanes and 100% for detecting at least one lane as compared to 52.5% and 85% respectively for Greedy HAAR Spiker. See table 4.4.

Table 4.4: Results of comparison with the previous work for input set 3

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total images</th>
<th>Both lanes</th>
<th>Accuracy</th>
<th>At least one lane</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Work</td>
<td>120</td>
<td>63</td>
<td>52.5%</td>
<td>102</td>
<td>85%</td>
</tr>
<tr>
<td>Current Work</td>
<td>120</td>
<td>114</td>
<td>95%</td>
<td>120</td>
<td>100%</td>
</tr>
</tbody>
</table>

The combined accuracy of the lane detection algorithm presented in this thesis comes out to be 96% for detecting both lanes and 100% for detecting at least one lane as compared to 50% and 84% respectively for previous work. Moreover, the current algorithm is able to process 7 frames per second in real time on Raspberry Pi as compared to 4.5 frames per second for previous algorithm. This shows that the current algorithm is 55% faster than the previously algorithm.

4.4 Results and Analysis

In this section the hardware and software aspects of the system are analyzed. The section also talks about the performance of the algorithm when run on Raspberry Pi in the real time.

In real time, the algorithm runs on a Raspberry Pi 3.0 Model B ARMv8 1 GB RAM and connected with a PiCamera rev 1.3 and a 7 inch touch screen display. The entire system is placed inside the car and the camera is placed on the windshield to capture the video. The power supply to Raspberry Pi and the display screen is provided by using a standard 12V-to-5V (2 Ampere) car charger.

The hardware cost of the entire lane detection system is approximately 150 USD.

For real time testing, the lane detection algorithm is run on the Raspberry Pi while driving at around 45mph. The resolution of the PiCamera is set to 400x255 pixels to be able
to get the execution speed of approx. 7 frames per second. The algorithm provides a fairly accurate real time solution to lane detection even while driving on the roads with sharp curvatures. Images from real time lane detection on RaspberryPi are shown in Figs 4.7 and 4.8.

Fig. 4.7: Real time lane detection (a)

Fig. 4.8: Real time lane detection (b)
CHAPTER 5
CONCLUSION AND FUTURE WORK

An algorithm for real time road Lane Lines Detection is presented in this thesis. The hardware cost for the lane detection system is kept minimal by using a RspberryPi computer to run the algorithm. Data is captured using a Pi Camera and results are displayed on a 7-inch touchscreen display. The entire system costs around USD 150 and is powered by the vehicle’s standard charger 12V-15V. The power requirement for this system is 10 Watts. All the software components used in the proposed lane detection system are open source.

Since the processing speed of the algorithm is 7 frames per second, it can be inferred that for one frame to process, the algorithm takes 1/7 seconds. If a vehicle is driving at the speed of ‘s’ mph, the distance between each processing frame can be given by below formula.

\[
d = \frac{s}{3600 \times 7} \text{miles}
\]

Therefore, 0.00004 miles will be traveled before a frame is processed by the given algorithm.

To run the experiments, data is captured in varying climate condition and the algorithm is run on a combined set of 26,425 images - 3800 images pertaining to rainy weather condition, 6139 images to sunny weather, and 16,486 to snowy condition. The data sets used for experiments for three different climate conditions can be accessed at: [28] and the resulting images can be accessed at: [29].

The algorithm is based on finding the gradient change in horizontal direction of the image and mark the lane line pixels. Using the detected pixels, lane lines are drawn using Probabilistic Hough Transform. Several filtering methods are applied to ensure that the detected lines belong to the road lane lines only. To increase the accuracy of the algorithm,
‘history’ of detected lane lines is maintained from the previous frames. When no lane lines are found in the current frame under consideration, lines are drawn using the history hence taking away the possibility of false negatives (visible lane lines which are failed to be detected by the algorithm).

The algorithm is implemented in Python 2.7.9 with OpenCV 3.0. The performance of the algorithm is tested ‘In Situ’ i.e real time on a Raspberry Pi 3 Model B ARMv8 1GB RAM computer on three sample of images each pertaining to rainy, sunny, and snow weather condition respectively. The accuracy of detecting both lane lines in the first sample of images is 97.65%, on the second sample of images is 96.67%, and on the third sample of images is 98.59%. The combined accuracy of detecting both lane lines comes out to be 98.24%.

The real time testing of the algorithm is performed by successfully executing 7 frames per second on the Raspberry Pi while driving at around 45mph speed. The resolution of the PiCamera is set to be 400x255 pixels for real time execution.

The source code of the algorithm, which also includes code for capturing the images using RaspberryPi, can be accessed at: [30].

The performance of the current implementation of the algorithm can be further improved by incorporating techniques to select the region of interest before processing the image. However, this will affect the processing speed of the algorithm and thus require a system with higher processing power. The proposed algorithm provides a fairly accurate detection of road lane lines but fails to detect the radius of curvature of lanes during a turn. Computing curvature would prove fruitful in detecting how far the vehicle is from the left and the right lane lines. Furthermore, the algorithm could be enhanced by adding the features like vehicle tracking and speed limit signs detection.
REFERENCES


[29] Results. [Online]. Available: https://www.dropbox.com/sh/xzhg8jdpfihn7m0/AAB52qyC_6p-wROJwIGirbsoa?dl=0