# LANE DETECTION

### A MINI PROJECT REPORT

**Submitted by**

**Srinivasan E V 212221060261**

**Rohan T 212221060228**

**Kishore Kumar R 212221060128**

***in partial fulfilment for the award of the degree of of***

**BACHELOR OF ENGINEERING** IN

**Electronics and communication** **Engineering**

**SAVEETHA ENGINEERING COLLEGE ANNA UNIVERSITY: CHENNAI- 600 025**

**DECEMBER 2022**

# ANNA UNIVERSITY: CHENNAI 600 025

**BONAFIDE CERTIFICATE**

Certified that this project report **“LANE DETECTION”,** is the bonafide work of **“Srinivasan E V – 212221060261 ,Rohan T -212221060228**

**Kishore Kumar R - 212221060128”,** who carried out the project work under our supervision.

### SIGNATURE SIGNATURE

Dr.LOGANATHAN Dr. G. Nagappan, M.E., PhD

Associate Professor Professor

### SUPERVISOR HEAD OF THE DEPARTMENT

Dept of Computer Science and Dept of Computer Science and Engineering, and Engineering,

Saveetha Engineering College, Saveetha Engineering College, Thandalam, Chennai 602105. Thandalam, Chennai 602105.

### DATE OF THE VIVA VOCE EXAMINATION: ……………………..

**INTERNAL EXAMINER EXTERNAL EXAMINER**

### ACKNOWLEDGEMENT

We express our deep sense of gratitude to our honourable and beloved Founder President **Dr. N. M. Veeraiyan**, our President **Dr.Saveetha Rajesh**, our Director **Dr. S. Rajesh** and other management members for providing the infrastructure needed.

We express our wholehearted gratitude to our principal, **Dr. N. Duraipandian**, for his wholehearted encouragement in completing this project.

We convey our thanks to **Dr. G. Nagappan, Professor** and Head of the Department of Computer Science and Engineering, Saveetha Engineering College, for his kind support and for providing necessary facilities to carry out the project work.

We would like to express our sincere thanks and deep sense of gratitude to our Supervisor **Dr.LOGANATHAN** and Project Coordinator **Dr.V.UmaRani Associate Professor**, Department of Computer Science and Engineering, Saveetha Engineering College, for her valuable guidance, suggestions and constant encouragement that paved the way for the successful completion of the project work and for providing us necessary support and details at the right time and during the progressive reviews.

We owe our thanks to all the members of our college, faculty, staff and technicians for their kind and valuable cooperation during the course of the project. We are pleased to acknowledge our sincere thanks to our beloved parents, friends and well-wishers who encouraged us to complete this project successfully.

### ABSTRACT

Lane Line detection is a critical component for self driving cars and also for computer vision in general. This concept is used to describe the path for self- driving cars and to avoid the risk of getting in another lane.

In this article, we will build a machine learning project to detect lane lines in real-time. We will do this using the concepts of computer vision using OpenCV library. To detect the lane we have to detect the white markings on both sides on the lane.

Using computer vision techniques in Python, we will identify road lane lines in which autonomous cars must run. This will be a critical part of autonomous cars, as the self-driving cars should not cross it’s lane and should not go in opposite lane to avoid accidents.

|  |  |
| --- | --- |
| **TABLE OF CONTENTS** |  |
| **TITLE** | **PAGE NO** |
| **ABSTRACT** | **I** |
| **LIST OF FIGURES** | **II** |
| **1.INTRODUCTION** |  |
| 1.1 OBJECTIVE | 2 |
| 1.1.1 DIGITAL IMAGE PROCESSING | 2 |
| 1.1.2 STEPS IN IMAGE PROCESSING | 2 |
| 1.1.3 IMAGE IN MATRIX REPRESENTATION | 2 |
| 1.2 SELF DRIVING CARS | 3 |
| 1.3 WORKING OF SELF DRIVING CARS | 3 |
| 1.3.1 DISORTION CORRECTION | 3 |
| 1.3.2 CREATE BINARY IMAGE | 4 |
| 1.4 PROBLEM STATEMENT  **2. LITERATURE SURVEY** | 5 |
| 2.1 EDGE DETECTION | 6 |
| 2.2 HOUGH TRANSFORM SPACE | 6 |
| 2.2.1 THEORY OF HOUGH TRANSFORM SPACE | 6 |
| 2.3 IMPLEMENTATION OF HOUGH TRANSFORM SPACE | 7 |
| 2.4 VARIATION AND EXTENTION | 7 |
| 2.4.1 Kernel Based Hough Transform (KHT) | 7 |
| 2.4.2 3-D Kernel-based Hough transform for plane detection (3DKHT) | 7 |
| 2.5 Detection of 3D objects (Planes and cylinders) | 7 |
| 2.6 USING WEIGHTED FEATURES | 8 |
| 2.7 EXISTING SYSTEM | 8 |
| 2.8 PROPOSED SYSTEM | 8 |

### 3.METHODOLOGY

|  |  |
| --- | --- |
| 3.1 IMAGE PROCESSING METHODOLOGY | 9 |
| 3.2 ARCHITECTURE | 11 |
| 3.3 CANNY EDGE DETECTION ARCHITECTURE: | 11 |
| 3.4 PROJECT MODULES | 12 |
| **4.IMPLEMENTATION** |  |
| 4.1 SYSTEM CONFIGURATION | 18 |
| 4.1.1 Software Configuration | 18 |
| 4.1.2 Hardware Configuration | 18 |
| 4.2 SAMPLE CODE | 19 |
| **5.CONCLUSION AND FUTURE WORKS** | 33 |
| **6.REFERENCES** | 35 |
| **7.APPENDIX** | 36 |

**LIST OF FIGURES**

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | FIGURE NO | TITLE | PAGE NO |
| 1. | 1.1 | MATRIX REPRESENTATION | 02 |
| 2. | 2.1 | HESSE NORMAL FORM GRAPH | 07 |
| 3. | 3.1 | ARCHITECTURE  DIAGRAM | 11 |
| 4. | 3.2 | CANNY EDGE  ARCHITECTURE | 12 |
| 5. | 5 | OUTPUTS | 36-40 |

# ABSTRACT

## Lane Line detection is a critical component for self driving cars and also for computer vision in general. This concept is used to describe the path for self- drivingcars and to avoid the risk of getting in another lane.

In this article, we will build a machine learning project to detect lane lines in real- time. We will do this using the concepts of computer vision using OpenCV library.To detect the lane we have to detect the white markings on both sides on the lane.

## Using computer vision techniques in Python, we will identify road lane lines in which autonomous cars must run. This will be a critical part of autonomous cars, asthe self-driving cars should not cross it’s lane and should not go in opposite lane to avoid accidents.

### CHAPTER 1 INTRODUCTION

1. OBJECTIVE OF THE PROJECT

Lane Line detection is a critical component for self driving cars and also for computer vision in general. This concept is used to describe the path for self-driving cars and to avoid the risk of getting in another lane.

* 1. DIGITAL IMAGE PROCESSING

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines too

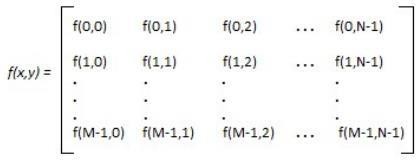
Image processing is basically signal processing in which input is an image and output is image or characteristics according to requirement associated with that image.

* + 1. Steps in Image Processing:

Image processing basically includes the following three steps:

* + - * Importing the image via image acquisition tools;
      * Analysing and manipulating the image;
      * Output in which result can be altered image or report that is based on image analysis.
    1. Image in Matrix Representation

As we know, images are represented in rows and columns we have the following syntax in which images are represented:



The right side of this equation is digital image by definition. Every element of this matrix is called image element, picture element, or pixel.

* 1. SELF DRIVING CARS

A self-driving car (sometimes called an autonomous car or driverless car) is a vehicle that uses a combination of sensors, cameras, radar and artificial intelligence (AI) to travel between destinations without a human operator. To qualify as fully autonomous, a vehicle must be able to navigate without human intervention to a predetermined destination over roads that have not been adapted for its use.

Companies developing and/or testing autonomous cars include Audi, BMW, Ford, Google, General Motors, Tesla, Volkswagen and Volvo. Google's test involved a fleet of selfdriving cars -- including Toyota Prii and an Audi TT -- navigating over 140,000 miles of California streets and highways.

* 1. WORKING OF SELF DRIVING CARS

Lane lines are being drawn as the car drives. Also, you can see the radius of curvature is being calculated to help the car steer. It is cheap to equip cars with a front facing camera. Much cheaper than RADAR or LIDAR. Once we get a camera image from the front facing camera of self - driving car, we make several modifications to it. The steps I followed are detailed below:

* + 1. Distortion correction

Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image. This transformation isn’t always perfect and distortion can result in a change in apparent size, shape or position of an object. So we need to correct this distortion to give the

camera an accurate view of the image. This is done by computing a camera calibration matrix by taking several chessboard pictures of a camera.

* + 1. Create a binary image

Now that we have the undistorted image, we can start our analysis. We need to explore different schemes so that we can clearly detect the object of interest on the road, in this case lane lines while ignoring the rest. I did this in two ways:

Using Sobel operator

to compute x-gradient The gradient of an image can be used to identify sharp changes in colour in a black and white image. It is a very useful technique to detect edges in an image. For the image of a road, we usually have a lane line in either yellow or white on a black road and so x- gradient can be very useful.

Explore other colour channels

HSV (Hue, Saturation and Value) colour space can be very useful in isolating the yellow and line white lines because it isolates colour (hue), amount of colour (saturation) and brightness (value). We can use the S colour channel in the image.

Birds Eye View Image

After the thresholding operation, we perform a perspective transform to change the image to bird’s eye view. This is done because from this top view we can identify the curvature of the lane and decide how to steer the car. To perform the perspe ctive transform, I identified 4 source points that form a trapezoid on the image and 4 destination points such that lane lines are parallel to each other after the transformation. The destination points were chosen by trial and error but once chosen works well for all images and the video since the camera is mounted in a fixed position. OpenCV can be used to perform this. See how clearly the curvature of the lane lines is visible in this view.

Fit curve lines to the bird eye view image

Plot the result identified by the system clearly. This plotting can be done filling the space area with transparent colour using OpenCV.

Thus, Self-Driving car works and road detection can be useful in detection of road from an image captured from car.

* 1. PROBLEM STATEMENT

Given an image captured from a camera attached to a vehicle moving on a road in which captured road may or may not be well levelled, or have clearly delineated edges, or some prior 9 | P a g e known patterns on it, then road detection from a single image can be applied to find the road in an image so that it could be used as a part in automation of driving system in the vehicles for moving the vehicle in correct road. In this process of finding the road in the image captured by the vehicle, we can use some algorithms for vanishing point detection using Hough transform space, finding the region of interest, edge detection using canny edge detection algorithm and then road detection. We use thousands of images of different roads to train our model so that the model could detect the road which is present in the new image processed through the vehicle.

#### CHAPTER 2

**LITERATURE SURVEY**

* 1. EDGE DETECTION

Edges characterize boundaries and are therefore a problem of fundamental importance in image processing. Edges in images are areas with strong intensity contrasts – a jump in intensity from one pixel to the next. Edge detecting an image significantly reduces the amount of data and filters out useless information, while preserving the important structural properties in an image.

Canny edge detection algorithm is also known as the optimal edge detector. Cranny’s intentions were to enhance the many edge detectors in the image.

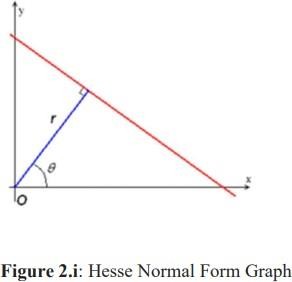
* + - The first criterion should have low error rate and filter out unwanted information while the useful information preserve.
    - The second criterion is to keep the lower variation as possible between the original image and the processed image.
    - Third criterion removes multiple responses to an edge.
  1. HOUGH TRANSFORM SPACE

The Hough transform is a feature extraction technique used in image analysis, computer vision, and digital image processing. The purpose of the technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure. This voting procedure is carried out in a parameter space, from which object candidates are obtained as local maxima in a so -called accumulator space that is explicitly constructed by the algorithm for computing the Hough transform.

2.2.1 Theory of Hough Transform Space

In automated analysis of digital images, a sub problem often arises of detecting simple shapes, such as straight lines, circles or ellipses. In many cases an edge detector can be used as apre- processing stage to obtain image points or image pixels that are on the desired curve in the image space. Due to imperfections in either the image data or the edge detector, however, there may

be missing points or pixels on the desired curves as well as spatial deviations be tween the ideal line/circle/ellipse and the noisy edge points as they are obtained from the edge detector. For. The purpose of the Hough transform is to address this problem by making it possible to perform groupings of edge points into object candidates by performing an explicit voting procedure over a set of parameterized image objects (Shapiro and Stockman, 304). The simplest case of Hough transform is detecting straight lines. In general, the straight-line y = mx + b can be represented as a point (b, m) in the parameter space. However, vertical lines pose a problem. They would give rise to unbounded values of the slope parameter m. Thus, for computational reasons, Duda and Hart proposed the use of the Hesse normal form. These reasons, it is often non-trivial to group the extracted edge features to an appropriate set of lines, circles or ellipses.



* 1. IMPLEMENTATION OF HOUGH TRANSFORM SPACE

The linear Hough transform algorithm uses a two-dimensional array, called an accumulator, to detect the existence of a line described by r=x cos theta+ y sin theta. The dimension of the accumulator equals the number of unknown parameters, i.e., two, considering quantized values of r and θ in the pair (r, θ). For each pixel at (x, y) and its neighbourhood, the Hough tr ansform algorithm determines if there is enough evidence of a straight line at that pixel.

The final result of the linear Hough transform is a two-dimensional array (matrix) similar to the accumulator—one dimension of this matrix is the quantized angle θ and the other dimension is the quantized distance r. Each element of the matrix has a value equal to the sum of the points or pixels that are positioned on the line represented by quantized parameters (r, θ). So, the element with the highest value indicates the straight line that is most represented in the input image.

* 1. VARIATIONS AND EXTENSIONS
     1. Kernel Based Hough Transform (KHT)

Fernandes and Oliveira suggested an improved voting scheme for the Hough transform that allows a software implementation to achieve real-time performance even on relatively large images (e.g., 1280×960). The Kernel-based Hough transform uses the same (r, theta) parameterization proposed by Duda and Hart but operates on clusters of approximately collinear pixels. For each cluster, votes are cast using an oriented Elliptical- Gaussian kernel that models the uncertainty associated with the best-fitting line with respectto the corresponding cluster. The approach not only significantly improves the performanceof the voting scheme, but also produces a much cleaner accumulator and makes the transform more robust to the detection of spurious lines.

* + 1. 3-D Kernel-based Hough transform for plane detection (3DKHT)

Limberger and Oliveira suggested a deterministic technique for plane detection in unorganized point clouds whose cost is nlogn in the number of samples, achieving real -time performance for relatively large datasets (up to 105 points on a 3.4 GHz CPU). It is based on a fast Hough-transform voting strategy for planar regions, inspired by the Kernel-based Hough transform (KHT). This 3D Kernel-based Hough transform

* 1. Detection of 3D objects (Planes and cylinders)

Hough transform can also be used for the detection of 3D objects in range data or 3D point clouds. The extension of classical Hough transform for plane detection is quite straightforward. A plane is represented by its explicit equation z=axx+ayy+d for which we can use a 3D Hough space corresponding to ax, ay and d. This extension suffers from the same problems as its 2D counterpart i.e., near horizontal planes can be reliably detected, while the performance deteriorates as planar direction becomes vertical (big values of ax and ay amplify the noise in the data).

For generalized plane detection using Hough transform, the plane can be parametrized by its normal vector n (using spherical coordinates) and its distance from the origin row resulting in a three dimensional Hough space.

Hough transform has also been used to find cylindrical objects in point clouds using a two-step

approach. The first step finds the orientation of the cylinder and the second step finds the position and radius.

* 1. USING WEIGHTED FEATURES

One common variation detail. That is, finding the bins with the highest count in one st age canbe used to constrain the range of values searched in the next.

* 1. EXISTING SYSTEM

In the current existing system is permitted only to use in ideal road conditions such as runway.

This could not be used in general roads because the edge detection used till now was Simulink Edge Detection which is implemented in MATLAB. The secondary thing is in current system Hough transform Space is only used for angle rotation and has very limited road dataset to detect the objects in single dimension of an image.

* 1. PROPOSED SYSTEM

In our proposed system we use Canny Edge Detection replacing the Simulink Edge Detectionwhich is recent and efficient implementation in Python instead of MATLAB. Since, Python is the Scripting and Statistical Modelling Language it supports faster execution for mathematical functions which could be used by Canny Edge Detection technique. Secondly, we use Hough Transform Space for 3- Dimensional Object detection which could faster and accurate compared to single dimension object detection.

### CHAPTER 3 METHODOLOGY

* 1. IMAGE PROCESSING METHODOLOGY

Digital image processing consists of the manipulation of images using digital computers. Its use has been increasing exponentially in the last decades. Its applications range from medicine to entertainment, passing by geological processing and remote sensing. Multimedia systems, one of the pillars of the modern information society, rely heavily on digital image processing.

Digital image processing is to process images by computer. Digital image processing can be defined as subjecting a numerical representation of an object to a series of operations in order to obtain a desired result.

Each pixel has two attributions: position and grey level. The position is determined by the two coordinates of sampling point in the scanning line, namely row and column. The integer indicating the brightness of the pixel position is called grey level. Images displayed by digital matrix are called digital images, and all digital image processing is based on the digital matrix. The digital matrix is the object process

f(i, j) = the grey level of pixel (i, j).

On the basis of image processing, it is necessary to separate objects from images by pattern recognition technology, then to identify and classify these objects through technologies provided by statistical decision theory. Under the conditions that an image includes several objects, the pattern recognition consists of three phasesssed by a computer.

* + - The first phase includes the image segmentation and object separation. In this phase, different objects are detected and separate from other background.
    - The second phase is the feature extraction. In this phase, objects are measured. The measuring feature is to quantitatively estimate some important features of objects, and
    - a group of the features are combined to make up a feature vector during feature extraction.
    - The third phase is classification. In this phase, the output is just a decision to determine which category every object belongs to.
  1. ARCHITECTURE

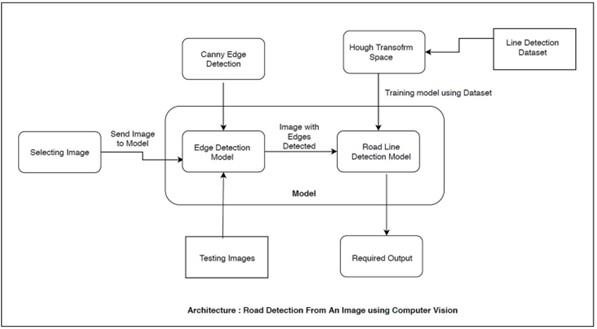
To build the architecture required by a project, we use incremental process model in which we test each prototype and then clubbed with the actual model on observing a correct output. Each Prototype is built along each model and then clubbed together with an actual model. In this way project is built using incremental model.

Figure 3.a: Architecture of Lane detection

* 1. CANNY EDGE DETECTION ARCHITECTURE:

The canny edge detection mainly focuses on change in intensity in an image. The change in pixel’s intensities from high intensity to low intensity is known as edge.

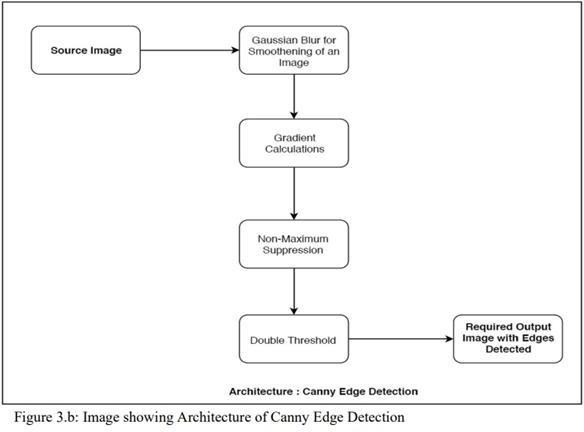
At first, the color image is changed into black and white image and is passed to smoothening technique.

We use Gaussian blur as a smoothening technique followed by gradient calculations, Nonmaximum suppression and double threshold.

Edge detection mainly use derivatives of pixel intensities in an image and then reduce the complexity of an image. The edge is detected when there is change in intensity from high to low which refers white shades to black shades (in gray scale image) in an image.

Gray Scale image is used because it would be easy to process the gray scale image than the colored image.

Thus, canny edge detection is used and its architecture is built.



* 1. PROJECT MODULES

Road Detection from a single image using Computer Vision consists of image insertion, model building and then testing. The model evaluation is done manually by the developer. We divided the complete project into mainly four modules. They are as follows:

Module 1: Selecting the appropriate testing image and video

Module 2: Preprocessing the selected image

Module 3: Edge Detection Implementation

Module 4: Hough Transformation

Module 5: Evaluating the output

Module 1: Selecting the appropriate testing image and video

It is the most important process in the project. Single Image from testing dataset is taken such a way that it reaches our implementation of a model. Each model we implement takes a resultant image as an input and process it further to produce an output. This selection of image is more important because implementation of each model requires an image input for processing. And if the processing is done, then output is produced. If output produced for the testing image is same as required , then the resultant image is sent to next process that we need to develop further. In order to observe the clear output, the best suitable image should be selected such a way that the testing image should be able to produce the clear required output at the end of processes.

Module 2: Preprocessing the selected image

Preprocessing plays a major role in producing the required output in sufficient required amount of time.

Preprocessing of selected image mainly undergo the gray scale conversion and smoothening techniques which would be considered as the first process in Canny’s process

. The selected image is converted into gray scale through the open source computer vision package. And then smoothening is applied by implementing the Gaussian Blur algorithm on the selected gray scale image.

A gray scale image mainly consists of change in variants from white to black that represents the color mixes of red, green and blue.

The normalization is main process of Gaussian Blur process conversion which is done through multiplying each intensities of a pixels by their corresponding normalized matrix values. Thus, preprocessing is done on the selected image.

Module 3: Edge Detection Implementation

The next step in the process if edge detection, which is main part in the program and required to detect the edges in the image irrespective of details present in an image. We use Canny Edge Detection Algorithm to implement the edge detection techniques because the other processes which are also used to find the edges in an image would contain detailed images compared to Canny Edge Detection Technique.

The Canny filter is a multi-stage edge detector. It uses a filter based on the derivative of a Gaussian in order to compute the intensity of the gradients. The Gaussian reduces the effect of noise present in the image. Then, potential edges are thinned down to 1- pixel curves by removing non- maximum pixels of the gradient magnitude. Finally, edge pixels are kept or removed using hysteresis thresholding on the gradient magnitude. The Canny has three adjustable parameters: the width of the

Gaussian (the noisier the image, the greater the width), and the low and high threshold for the hysteresis thresholding.

The general criteria for edge detection include:

* Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible.
* The edge point detected from the operator should accurately localize on the center of the edge.
* A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

Module 4: Hough Transformations

The Hough transform is a technique which can be used to isolate features of a particular shape within an image. Because it requires that the desired features be specified in some parametric form, the classical Hough transform is most commonly used for the detection of regular curves such as lines, circles, ellipses, etc. A generalized Hough transform can be employed in applications where a simple analytic description of a feature(s) is not possible.

The Hough technique is particularly useful for computing a global description of a feature(s) (where the number of solution classes need not be known a priori), given (possibly noisy) local measurements.

As a simple example, consider the common problem of fitting a set of line segments to a set of discrete image points (e.g. pixel locations output from an edge detector). Figure 1 shows some possible solutions to this problem. Here the lack of a priori knowledge about the number of desired line segments (and the ambiguity about what constitutes a line segment) render this problem under- constrained.

We can analytically describe a line segment in a number of forms. However, a convenient equation for describing a set of lines uses parametric or normal notion:

X cos t + Y sin t = r

where r is the length of a normal from the origin to this line and t is the orientation of r with respect to the X-axis. (See Figure 2.) For any point (X, Y) on this line, r and t are constant.

Module 5: Evaluating the output

Evaluation of output is done through confusion matrix and the accuracy metrics when a testing dataset is passed. When all the images of testing dataset are passed into the model, then the output is observed on every image and then the true positives, false positives, true negatives and false negatives are noted. This forms a confusion matrix and then accuracy, precision scores can be noted. This process is known as Evaluation. Thus, all the accuracy and precision of a model are noted.

### CHAPTER 4

**IMPLEMENTATION**

* 1. SYSTEM CONFIGURATION
     1. Software Configuration

These are the Software Configurations that are required.

* Operating System: Windows 10/8/7 (incl. 64-bit), Mac OS, Linux
* Language: Python 3
* IDE: Jupyter Notebook/ Spyder 3
* Framework: Tkinter
  + 1. Hardware Configuration

These are minimum Hardware configurations that are required.

* Processor: Intel core 2 duo or higher.
* RAM: 1 GB or higher
* HDD: 256 GB or higher
* Monitor: 1024 x 768 minimum screen resolution.
* Keyboard: US en Standard Keyboard
  1. SAMPLE CODE

#### COLOR SELECTION

import matplotlib.pyplot as plt import matplotlib.image as mpimg import numpy as np

# Read in the image

image = mpimg.imread('test\_images/solidWhiteRight.jpg')

# Grab the x and y size and make a copy of the image ysize = image.shape[0]

xsize = image.shape[1] color\_select = np.copy(image)

# Define color selection criteria

###### MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION

red\_threshold = 200

green\_threshold = 200

blue\_threshold = 200 ######

rgb\_threshold = [red\_threshold, green\_threshold, blue\_threshold]

# Do a boolean or with the "|" character to identify # pixels below the thresholds

thresholds = (image[:,:,0] < rgb\_threshold[0]) \

| (image[:,:,1] < rgb\_threshold[1]) \

| (image[:,:,2] < rgb\_threshold[2]) color\_select[thresholds] = [0,0,0]

# Display the image plt.imshow(image) plt.title("Input Image") plt.show() plt.imshow(color\_select) plt.title("Color Selected Image") plt.show()

#### REGION MASKING

**import matplotlib.pyplot as plt**

#### import matplotlib.image as mpimg import numpy as np

***# Read in the image***

#### image = mpimg.imread('test\_images/solidWhiteRight.jpg')

***# Grab the x and y size and make a copy of the image***

#### ysize = image.shape[0] xsize = image.shape[1]

**color\_select = np.copy(image) line\_image = np.copy(image)**

***# Define color selection criteria***

***# MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION***

#### red\_threshold = 200

**green\_threshold = 200**

#### blue\_threshold = 200

**rgb\_threshold = [red\_threshold, green\_threshold, blue\_threshold]**

***# Define the vertices of a triangular mask.***

***# Keep in mind the origin (x=0, y=0) is in the upper left***

***# MODIFY THESE VALUES TO ISOLATE THE REGION # WHERE THE LANE LINES ARE IN THE IMAGE***

#### left\_bottom = [100, 539]

**right\_bottom = [950, 539]**

#### apex = [480, 290]

***# Perform a linear fit (y=Ax+B) to each of the three sides of the triangle # np.polyfit returns the coefficients [A, B] of the fit***

#### fit\_left = np.polyfit((left\_bottom[0], apex[0]), (left\_bottom[1], apex[1]), 1)

**fit\_right = np.polyfit((right\_bottom[0], apex[0]), (right\_bottom[1], apex[1]), 1)**

#### fit\_bottom = np.polyfit((left\_bottom[0], right\_bottom[0]), (left\_bottom[1], right\_bottom[1]), 1)

***# Mask pixels below the threshold***

#### color\_thresholds = (image[:,:,0] < rgb\_threshold[0]) | \ (image[:,:,1] < rgb\_threshold[1]) | \

**(image[:,:,2] < rgb\_threshold[2])**

***# Find the region inside the lines***

#### XX, YY = np.meshgrid(np.arange(0, xsize), np.arange(0, ysize)) region\_thresholds = (YY > (XX\*fit\_left[0] + fit\_left[1])) & \

**(YY > (XX\*fit\_right[0] + fit\_right[1])) & \**

#### (YY < (XX\*fit\_bottom[0] + fit\_bottom[1]))

***# Mask color and region selection* color\_select[color\_thresholds | ~region\_thresholds] = [0, 0, 0] *# Color pixels red where both color and region selections met***

#### line\_image[~color\_thresholds & region\_thresholds] = [9, 255, 0]

***# Display the image and show region and color selections***

#### plt.imshow(image)

**x = [left\_bottom[0], right\_bottom[0], apex[0], left\_bottom[0]]y**

#### = [left\_bottom[1], right\_bottom[1], apex[1], left\_bottom[1]] plt.plot(x, y, 'r--', lw=4)

**plt.title("Region Of Interest") plt.show() plt.imshow(color\_select)**

#### plt.title("Color Selection in the Triangular Region") plt.show()

**plt.imshow(line\_image)**

#### plt.title("Region Masked Image [Lane Lines in Green]") plt.show()

* **CANNY EDGE DETECTION**

import matplotlib.pyplot as plt import matplotlib.image as mpimg import numpy as np

import cv2

*# Read in the image and convert to grayscale*

*# Note: in the previous example we were reading a .jpg # Here we read a .png and convert to 0,255 bytescale* image = mpimg.imread('test\_images/solidYellowLeft.jpg') gray = cv2.cvtColor(image,cv2.COLOR\_RGB2GRAY)

*# Define a kernel size for Gaussian smoothing / blurring*

kernel\_size = 5 *# Must be an odd number (3, 5, 7...)*

blur\_gray = cv2.GaussianBlur(gray,(kernel\_size, kernel\_size),0)

*# Define our parameters for Canny and run it*

low\_threshold = 180

high\_threshold = 240

edges = cv2.Canny(blur\_gray, low\_threshold, high\_threshold)

*# Display the image*

plt.imshow(edges, cmap='Greys\_r') plt.title("Canny Edge Detection Image") plt.show()

Hough Transform and detecting Lane Lines

*# Read in and grayscale the image*

image = mpimg.imread('test\_images/solidYellowLeft.jpg') gray = cv2.cvtColor(image,cv2.COLOR\_RGB2GRAY)

*# Define a kernel size and apply Gaussian smoothing*

kernel\_size = 5

blur\_gray = cv2.GaussianBlur(gray,(kernel\_size, kernel\_size),0)

*# Define our parameters for Canny and apply*

low\_threshold = 180

high\_threshold = 240

edges = cv2.Canny(blur\_gray, low\_threshold, high\_threshold)

*# Next we'll create a masked edges image using cv2.fillPoly()*

mask = np.zeros\_like(edges) ignore\_mask\_color = 255

*# This time we are defining a four sided polygon to mask*

imshape = image.shape

vertices = np.array([[(0,imshape[0]),(450, 290), (490, 290), (imshape[1],imshape[0])]], dtype=np.int32)

cv2.fillPoly(mask, vertices, ignore\_mask\_color) masked\_edges = cv2.bitwise\_and(edges, mask)

*# Define the Hough transform parameters*

*# Make a blank the same size as our image to draw on*

rho = 1 *# distance resolution in pixels of the Hough grid*

theta = np.pi/180 *# angular resolution in radians of the Hough grid*

threshold = 2 *# minimum number of votes (intersections in Hough grid cell)*

min\_line\_length = 4 *#minimum number of pixels making up a line*

max\_line\_gap = 5 *# maximum gap in pixels between connectable line segments*

line\_image = np.copy(image)\*0 *# creating a blank to draw lines on*

*# Run Hough on edge detected image*

*# Output "lines" is an array containing endpoints of detected line segments*

lines = cv2.HoughLinesP(masked\_edges, rho, theta, threshold, np.array([]), min\_line\_length, max\_line\_gap)

*# Iterate over the output "lines" and draw lines on a blank image*

for line **in** lines:

for x1,y1,x2,y2 **in** line: cv2.line(line\_image,(x1,y1),(x2,y2),(255,0,0),10)

*# Create a "color" binary image to combine with line image*

color\_edges = np.dstack((edges, edges, edges))

*# Draw the lines on the edge image*

lines\_edges = cv2.addWeighted(color\_edges, 0.8, line\_image, 1, 0)

lines\_edges = cv2.polylines(lines\_edges,vertices, True, (0,0,255), 10) plt.imshow(image)

plt.title("Input Image") plt.show() plt.imshow(lines\_edges)

plt.title("Colored Lane line [In RED] and Region of Interest [In Blue]") plt.show()

#### INTEGRATING THE PROCESS import math

**def grayscale(img):**

#### return cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)

***# Or use BGR2GRAY if you read an image with cv2.imread() # return cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)***

#### def canny(img, low\_threshold, high\_threshold):

***"""Applies the Canny transform"""***

#### return cv2.Canny(img, low\_threshold, high\_threshold)

**def gaussian\_blur(img, kernel\_size):**

***"""Applies a Gaussian Noise kernel"""***

#### return cv2.GaussianBlur(img, (kernel\_size, kernel\_size), 0)

**def region\_of\_interest(img, vertices):**

***#defining a blank mask to start with***

#### mask = np.zeros\_like(img)

***#defining a 3 channel or 1 channel color to fill the mask with depending on the input image***

#### if len(img.shape) > 2:

**channel\_count = img.shape[2] *# i.e. 3 or 4 depending on your image***

#### ignore\_mask\_color = (255,) \* channel\_count else:

**ignore\_mask\_color = 255**

***#filling pixels inside the polygon defined by "vertices" with the fill color***

#### cv2.fillPoly(mask, vertices, ignore\_mask\_color)

***#returning the image only where mask pixels are nonzero***

#### masked\_image = cv2.bitwise\_and(img, mask) return masked\_image

**def draw\_lines(img, lines, color=[255, 0, 0], thickness=10):**

#### for line in lines:

**for x1,y1,x2,y2 in line:**

#### cv2.line(img, (x1, y1), (x2, y2), color, thickness def slope\_lines(image,lines):

**img = image.copy()**

#### poly\_vertices = [] order = [0,1,3,2]

**left\_lines = [] *# Like /* right\_lines = [] *# Like \* for line in lines:**

#### for x1,y1,x2,y2 in line:

**if x1 == x2:**

**pass *#Vertical Lines***

#### else:

**m = (y2 - y1) / (x2 - x1) c = y1 - m \* x1**

#### if m < 0: left\_lines.append((m,c))

**elif m >= 0:**

#### right\_lines.append((m,c))

**left\_line = np.mean(left\_lines, axis=0) right\_line = np.mean(right\_lines, axis=0)**

***#print(left\_line, right\_line)***

#### for slope, intercept in [left\_line, right\_line]:

***#getting complete height of image in y1***

#### rows, cols = image.shape[:2]

**y1= int(rows) *#image.shape[0]***

***#taking y2 upto 60% of actual height or 60% of y1***

**y2= int(rows\*0.6) *#int(0.6\*y1)***

***#we know that equation of line is y=mx +c so we can write it x=(y-c)/m***

#### x1=int((y1-intercept)/slope) x2=int((y2-intercept)/slope) poly\_vertices.append((x1, y1)) poly\_vertices.append((x2, y2))

**draw\_lines(img, np.array([[[x1,y1,x2,y2]]]))**

#### poly\_vertices = [poly\_vertices[i] for i in order]

**cv2.fillPoly(img, pts = np.array([poly\_vertices],'int32'), color = (0,255,0)) return cv2.addWeighted(image,0.7,img,0.4,0.)**

***#cv2.polylines(img,np.array([poly\_vertices],'int32'), True, (0,0,255), 10) #print(poly\_vertices)***

#### def hough\_lines(img, rho, theta, threshold, min\_line\_len, max\_line\_gap):

***"""***

***`img` should be the output of a Canny transform.***

***Returns an image with hough lines drawn. """***

#### lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=min\_line\_len, maxLineGap=max\_line\_gap)

**line\_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)**

***#draw\_lines(line\_img, lines)***

#### line\_img = slope\_lines(line\_img,lines) return line\_img

***# Python 3 has support for cool math symbols.***

#### def weighted\_img(img, initial\_img, α=0.1, β=1., γ=0.): lines\_edges = cv2.addWeighted(initial\_img, α, img, β, γ)

***#lines\_edges = cv2.polylines(lines\_edges,get\_vertices(img), True, (0,0,255), 10)***

#### return lines\_edges def get\_vertices(image):

**rows, cols = image.shape[:2] bottom\_left = [cols\*0.15, rows] top\_left = [cols\*0.45, rows\*0.6] bottom\_right = [cols\*0.95, rows] top\_right = [cols\*0.55, rows\*0.6]**

#### ver = np.array([[bottom\_left, top\_left, top\_right, bottom\_right]], dtype=np.int32) return ver

linkcode

***# Lane finding Pipeline***

#### def lane\_finding\_pipeline(image):

***#Grayscale***

#### gray\_img = grayscale(image)

***#Gaussian Smoothing***

#### smoothed\_img = gaussian\_blur(img = gray\_img, kernel\_size = 5)

***#Canny Edge Detection***

#### canny\_img = canny(img = smoothed\_img, low\_threshold = 180, high\_threshold = 240)

***#Masked Image Within a Polygon***

#### masked\_img = region\_of\_interest(img = canny\_img, vertices = get\_vertices(image))

***#Hough Transform Lines***

#### houghed\_lines = hough\_lines(img = masked\_img, rho = 1, theta = np.pi/180, threshold = 20, min\_line\_len = 20, max\_line\_gap = 180)

***#Draw lines on edges***

#### output = weighted\_img(img = houghed\_lines, initial\_img = image, α=0.8, β=1., γ=0.

**return output**

#### TESTING THE PIPELINE WITH DIFFERENT IMAGES for image\_path in list(os.listdir('./test\_images')):

**fig = plt.figure(figsize=(20, 10))**

#### image = mpimg.imread(f'./test\_images/{image\_path}') ax = fig.add\_subplot(1, 2, 1,xticks=[], yticks=[]) plt.imshow(image)

**ax.set\_title("Input Image")**

**ax = fig.add\_subplot(1, 2, 2,xticks=[], yticks=[]) plt.imshow(lane\_finding\_pipeline(image)) ax.set\_title("Output Image [Lane Line Detected]") plt.show()**

### CHAPTER 5

**CONCLUSION AND FUTURE WORKS**

* 1. CONCLUSION

When we drive, we use our eyes to decide where to go. The lines on the road that show us where the lanes are act as our constant reference for where to steer the vehicle.

Naturally, one of the first things we would like to do in developing a self -driving vehicle is to automatically detect lane lines using an algorithm. The road detection regio n of interest (ROI), must be flexible. When driving up or down a steep incline, the horizon will change and no longer be a product of the proportions of the frame. This is also something to consider for tight turns and bumper to bumper traffic. This project is entirely based on image processing and road detection in self-driving vehicles in which has a great scope in future.

We have completed the entire implementation using specific algorithms to detect the road clearly. If the people’s thought hasn’t changed about the self-driving cars being safe, these cars are already safe and are becoming safer. Only if they believe and give a try to technology, they get to enjoy the luxury of computerized driving.

Driverless cars appear to be an important next step in transportation technology. They are a new all-media capsule- text to your heart’s desire and it’s safe. Developments in autonomous cars is continuing and the software in the car is continuing to be updated.

Though it all started from a driverless thought to radio frequency, cameras, sensors, more semiautonomous features will come up, thus reducing the congestion, increasing the safety with faster reactions and fewer errors.

* 1. FUTURE WORKS

Self-driving technology has the potential to reduce crashes, but some high-profile accidents have raised questions about risks posed by poorly functioning autonomous -driving systems. In January 2016, a man was killed in China after his Tesla crashed into the back of a cleaning vehicle. The Tesla reportedly had its self-driving features activated at the time of the crash. This marked the first reported death in which a vehicle's ADS features were viewed as a potential contributing factor, although the police did find that the Tesla driver had not been paying attention to the road in accordance with the autopilot rules. The public data available for evaluating how safe self-driving cars are remains somewhat limited. Most of the cities and states in which autonomous driving testing is taking place tend to have relatively dr y weather conditions and simple road systems that make it easier for driverless vehicles to function. California is also the only state in America that requires companies testing driverless cars to submit reports detailing each accident involving autonomou s vehicles on

### 6. REFERENCES

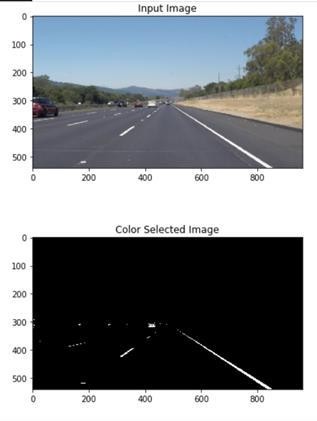
1. General Road Detection From A Single Image, TIP-05166-2009, ACCEPTED, Hui Kong , Member, IEEE, Jean-Yves Audibert, and Jean Ponce , Fellow, IEEE Willow Team, Ecole Normale Superieure / INRIA / CNRS, Paris, France Imagine team, Ecole des Ponts ParisTech, Paris, France.
2. J. C. McCall and M. M. Trivedi, “Video based lane estimation and tracking for driver assistance: Survey, system, and evaluation,” IEEE Trans. on Intelligent Transportation Systems, pp. 20 –37, 2006.
3. K.-Y. Chiu and S.-F. Lin, “Lane detection using color-based segmentation,” IEEE Intelligent Vehicles Symposium, 2005. 1
4. H. Kong, J.-Y. Audibert, and J. Ponce, “Vanishing point detection for road detection,” CVPR, 2009.
5. Y. Wang, E. K. Teoh, and D. Shen, “Lane detection and tracking using b-snake,” Image and Vision Computing, pp. 269–280, 2004
6. A. Lookingbill, J. Rogers, D. Lieb, J. Curry, and S. Thrun, “Reverse optical flow for selfsupervised adaptive autonomous robot navigation,” IJCV, vol. 74, n o. 3, pp. 287–302, 2007.
7. A. Broggi, C. Caraffi, R. I. Fedriga, and P. Grisleri, “Obstacle detection with stereo vision for off - road vehicle navigation,” IEEE International Workshop on Machine Vision for Intelligent Vehicles, 2005.
8. J. Sparbert, K. Dietmayer, and D. Streller, “Lane detection and street type classification using laser range images,” IEEE Proceedings in Intelligent transportation Systems, pp. 456 – 464, 2001.
9. J. B. Southhall and C. Taylor, “Stochastic road shape estimation,” ICCV, pp . 205–212, 2001.

32

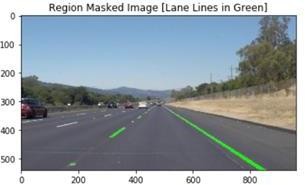
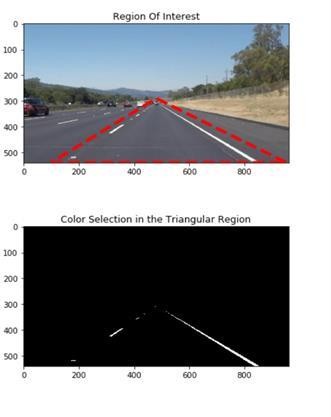
### APPENDIX

OUTPUTS

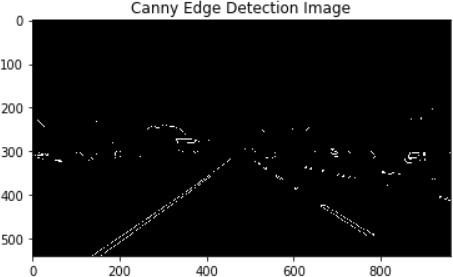
#### COLOR SELECTION:



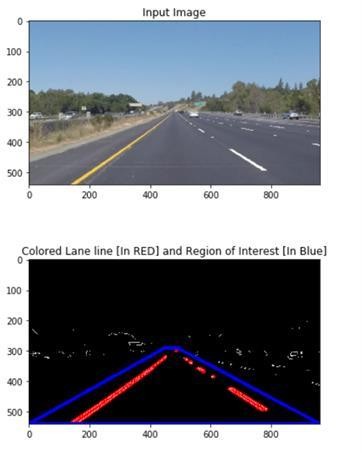
1. **REGION MASKING**



#### CANNY EDGE DETECTION



1. **HOUGH TRANSFORMATION**



#### FINAL OUTPUT:



