Project Report

Automatic Lane Line Detection Using OpenCV

CS6384 Computer Vision

Submitted On - 2nd April 2022

Submitted By -
Neetesh Kumar Dadwariya (nkd200001)
The University of Texas at Dallas.
Master's in Computer Science

Contents

1. Introduction

The problem of detecting road lanes is well-famous. Autonomous driving cars are one of the most disruptive innovations in AI. An autonomous car can go anywhere a traditional car can go and does everything that an experienced human driver does. But it's very essential to train it properly. One of the many steps involved during the training of an autonomous driving car is lane detection, which is the preliminary step.

In this project, our objective is to count the number of road lane lines, and draw them on the image. The lane lines can be continuous or dotted lines. They can be white, yellow, or red (fire lane. We will evaluate the approaches for the steps taken as well the outcomes of the steps which give promising results.

2. Approach, Steps and Pipeline

The task for lane detection is a challenging one and requires careful treatment of the provided image as the image can be taken from a variety of angles. The provided images can also possess various types of noises which makes it more difficult to generalize across different test dataset.

We will discuss here the approaches taken, and their corresponding outputs to demonstrate the effectiveness of the solutions.

1. Image Resizing

The input images can be of varying sizes and dimensions. To apply standard operations on the images, the images should be resized to certain fixed dimensions, so that the behavior of the operations can be generalized easily. In the code, the images are resized to [720x480]. For image resizing, we are using OpenCV resizing function -

- cv2.resize(image, dim, interpolation=cv2.INTER_LINEAR)

2. Image Segmentation using Color Selection

The problem statement has provided the constraints that the images would be of asphalt roads with white, yellow and red lines. This insight allows us to take the opportunity to perform color segmentation and ROI selection basis on the gray color of road.

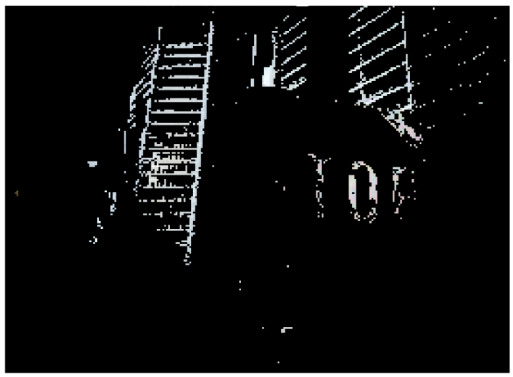
The images can be viewed in different color spaces, with each color space having different properties. The visualization for color masks for the different color spaces is shown below -

A. Color Mask in RGB color space -

Below are the images for mask in RGB color space.

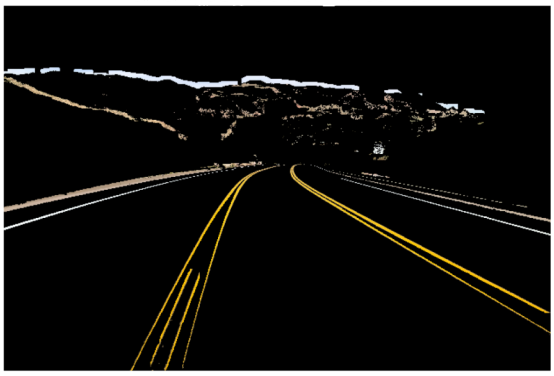
B. Color Mask in HSV color space -

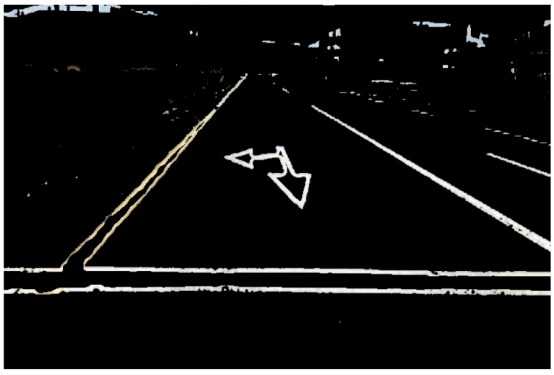
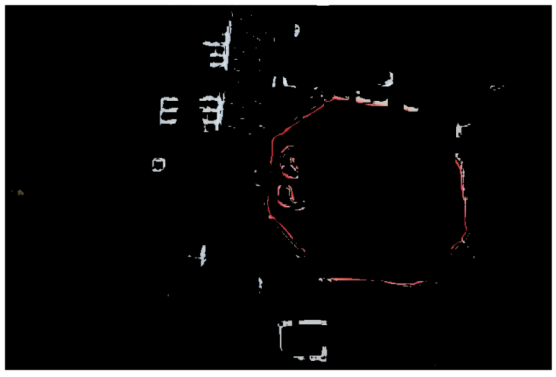
Below are the images for mask in HSV color space.



C. Color Mask in HSL color space -

Below are the images for mask in HSL color space.


As we can see, the colors in HSL color space are sharp, and suitable to use as mask for the white, yellow and red lines. Hence, we have proceeded by choosing HSL color space as our image segmentation.

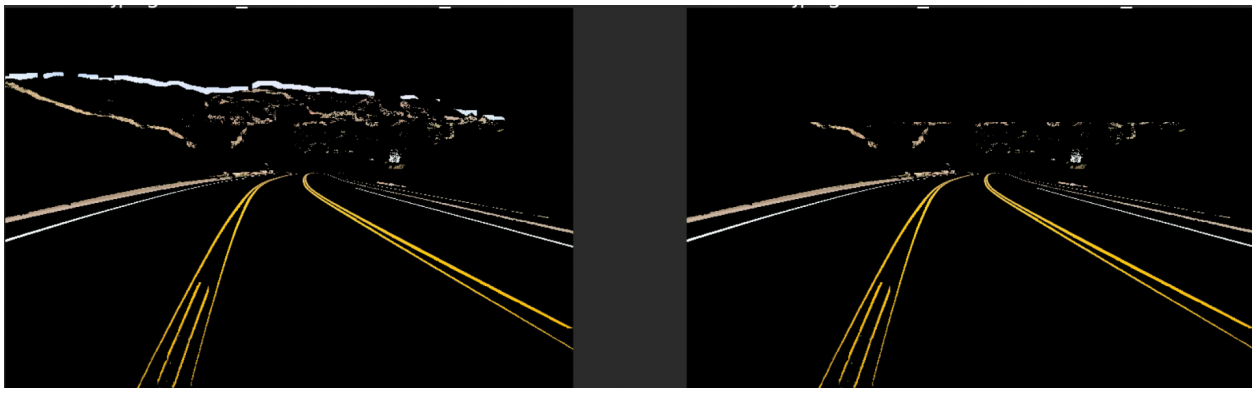
To perform image segmentation using color masks, we performed following steps -

1. Convert image from RGB → HLS Color space
2. Apply Gaussian Kernel to reduce noise in the space for better color extraction
3. With the help of Contouring and Dilation, reduced extra boundaries to get clear separation of road and lanes.
4. Combined finely tuned red, yellow white and gray color masks, and applied on top of the original image to get a segmented color mask for the region of interest.

3. Region of Interest Selection -

In Image processing, region of interest (ROI) selection is an important procedure. This helps us to segregate the required operating area from the rest of the image. ROI can be thought of as a closed polygon which has the bounds for the area to be worked upon. We are choosing 70% of the image are from the bottom, as rest 30% of top of the image would most likely be sky or flyovers.

We use *fillPoly* method to fill the ROI. Pixels in the mask are set to 0 for the area outside ROI.



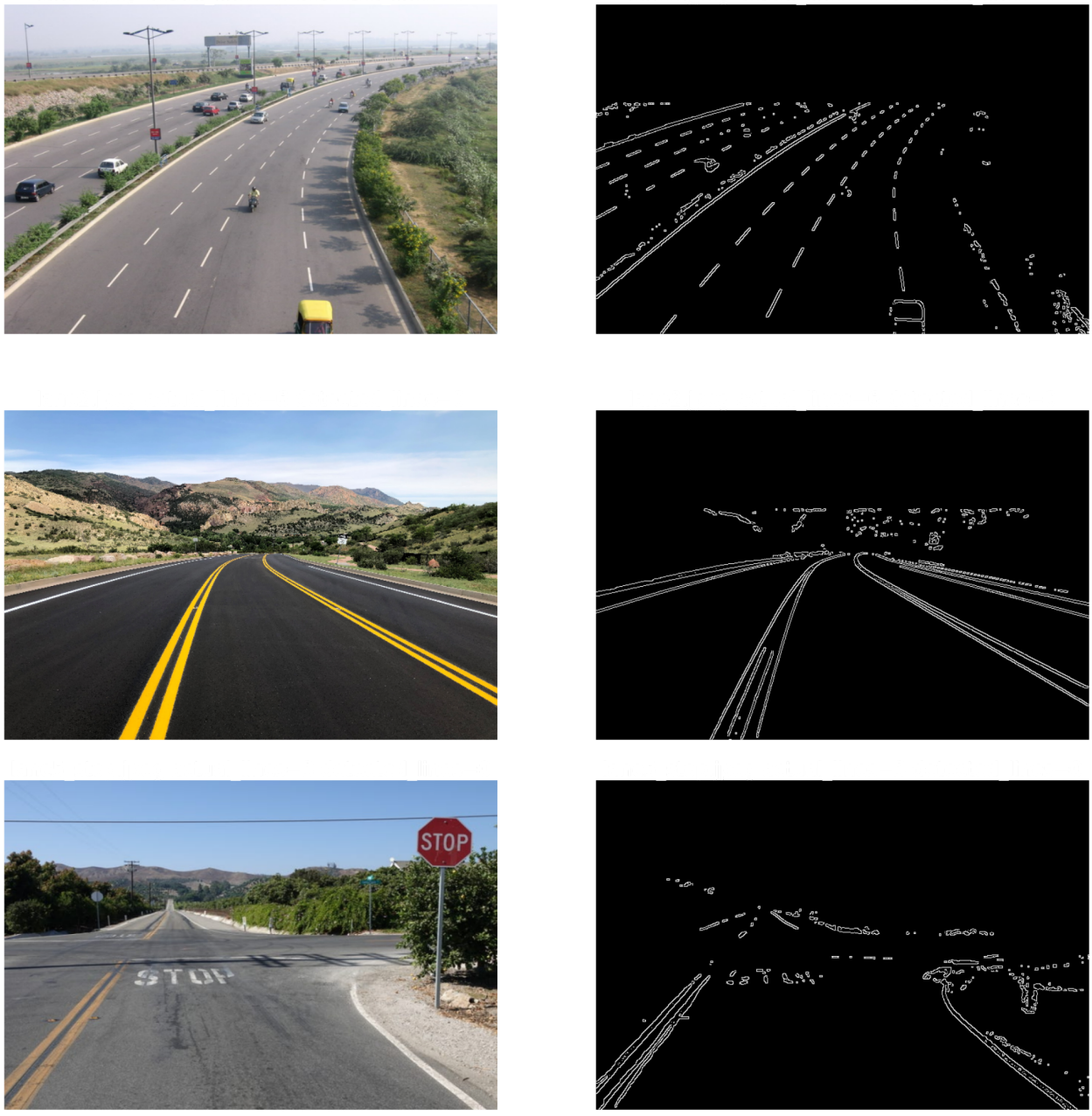
Region of Interest Selection

4. Gaussian and Canny Edge detection -

Gaussian blur is a technique used to reduce noise as well as fine details from the image. We perform this step to keep only the most prominent edges in the image and remove any trivial ones. It helps to smooth out the edges before applying an edge detection technique.

Edge detection is performed for detecting rising and falling edge. It provided useful information of where the lane is most likely to be present. We can use Sobel or Robert's operator to detect edges but on trial, we found Canny edge detection to produce the best results.

Canny internally performs Gaussin blur to remove the noise. It also performs non-maximum suppression of the false edges.



Result of Canny Edge Detection

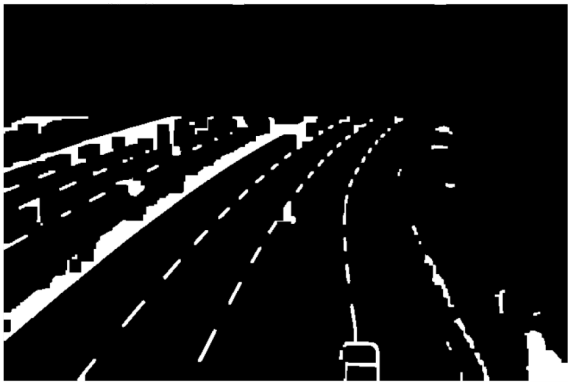
5. Noise Reduction Using Morphological Operators -

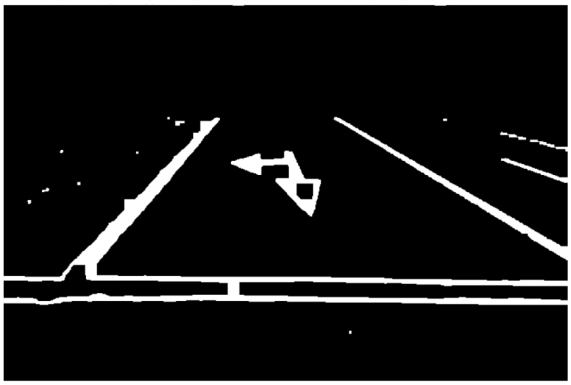
Morphological operators — such as dilation, opening and closing can be used to minimize the noise that is closer to the area of interest. Although Canny internally performs noise reduction, there can still be some noise after the canny in edge, which we might manually need to filter out.

For our cases, we are performing closing. Closing is useful to reduce small holes that are closer to the boundary of the lanes. We perform closing with a kernel size of 15 which gives optimum results.

We use the following OpenCV function to perform closing —

- cv2.morphologyEx(image, cv2.MORPH_CLOSE, np.ones((kernel_size, kernel_size), np.uint8))





6. Contour Detection -

Contour can be defined as the closed curve which can be defined by the exterior boundaries of an object. A contour can detect a closed figure unlike edge detection to solidify object where boundaries are outlined. OpenCV provides the functionalities of finding and drawing contours.

For Finding and filling contour, we use —

- cv2.findContours(image, mode=cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
- cv2.fillPoly(image, pts=contours, color=(255, 255, 255))

It will fill the contour internally by maintaining the external boundaries.

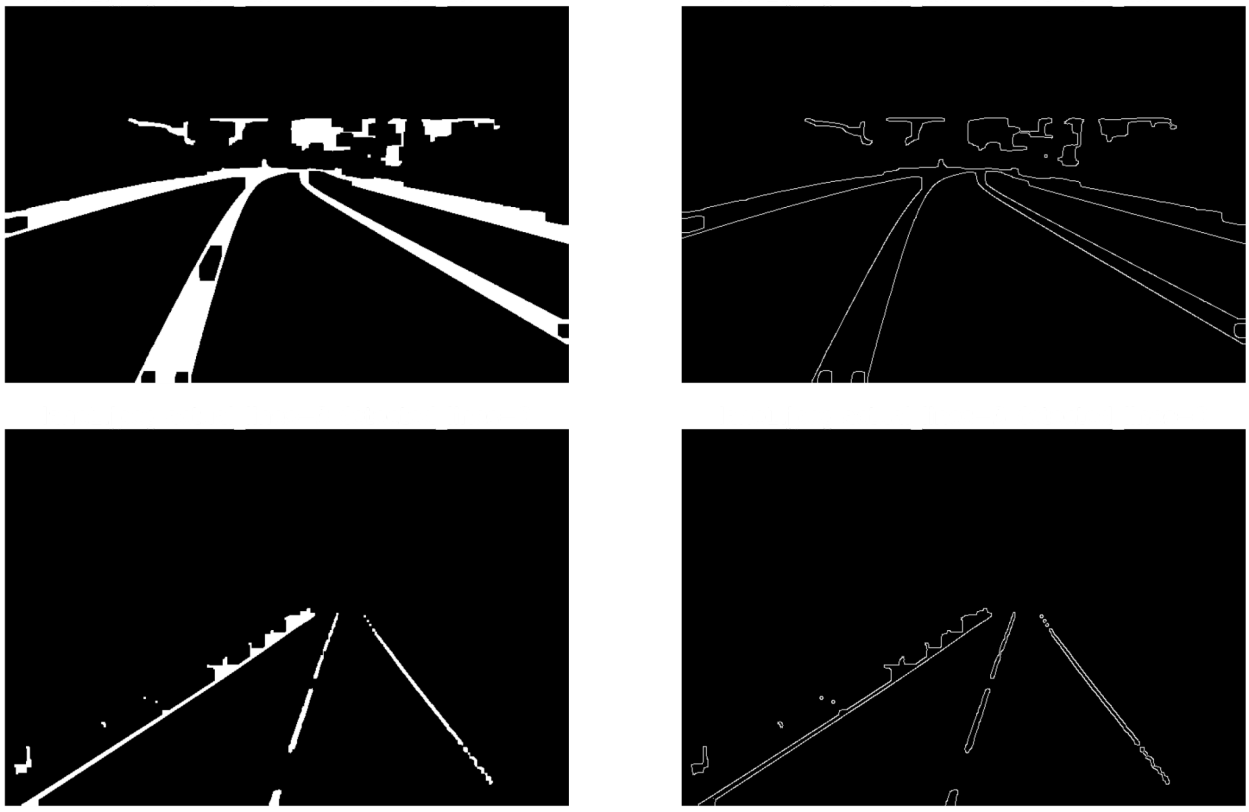
Effect of Contouring

7. Canny edge detection again to extract smooth lines —

From contouring, we have closed all the approximate regions and now our image is smooth enough to detect the lanes. So hence we apply canny edge detection again, that will give us best edges for the lane lines.

For Canny Edge Detection, we use the following function —

- cv2.Canny(image, threshold1=250, threshold2=255, apertureSize=7)



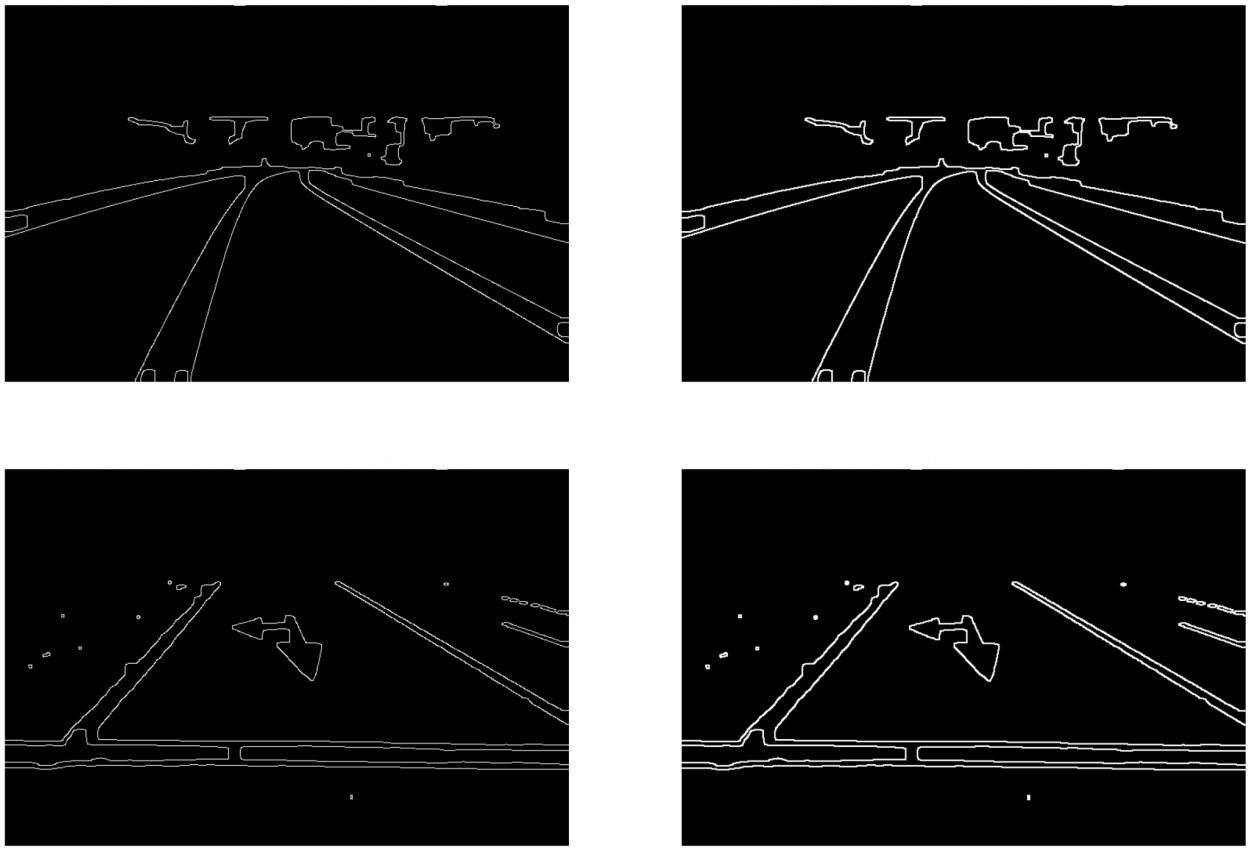
Effect of Canny 2nd time

8. Dilation —

Dilation helps in increasing the boundaries. This is an important step, as it will make edges more prominent.

We use OpenCV dilation function here —

- cv2.dilate(image, np.ones((kernel_size, kernel_size), np.uint8), iterations)



Effect of Dilation

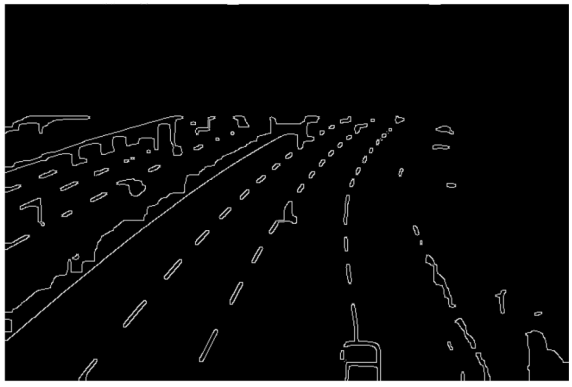
9. Line detection using Hough transform —

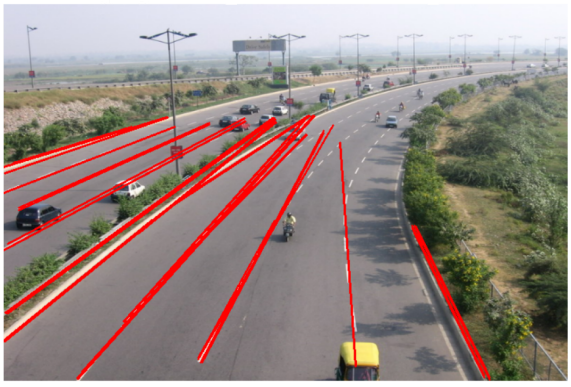
The Hough Transform algorithm detects lines by finding the (rho, thetha) pairs that have a number of intersections larger than a certain threshold. Hough transforms is the techniques that can be applied to detect straight lines in the image.

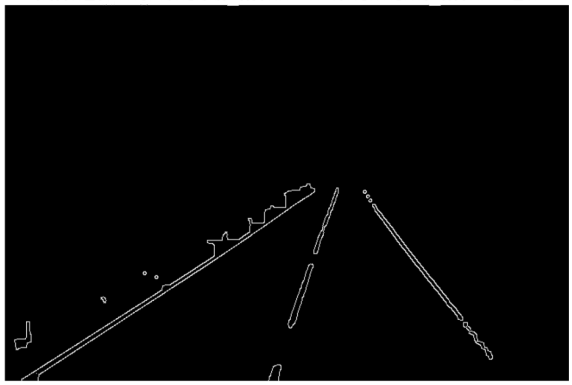
It helps to figure out the prominent lines and connect the disjoint edge points in an image. This is done by representing points as lines and sinusoidal on Cartesian and Polar co-ordinate systems respectively. An intersection of lines in Hough space will thus correspond to a line in Cartesian space. OpenCV's HoughLinesP function is used here to return all the lines in the edge image. We use Probabilistic Hough Transform since it is an optimized version of the Hough Transform and yields more precise lines. We draw the lines on the image using OpenCV's line function.

HoughLinesP have many tuning parameters available which can be configured to optimize the best results —

a. rho: Distance resolution of the accumulator in pixels
b. theta: Angle resolution of the accumulator in radians
c. threshold: Accumulator threshold parameter. Line votes > threshold to be returned.
d. minLineLength: Line segments shorter that this are rejected
e. maxLineGap: Maximum allowed gap between points on the same line to link them





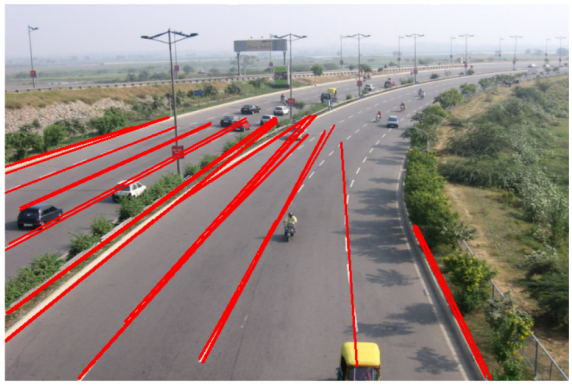


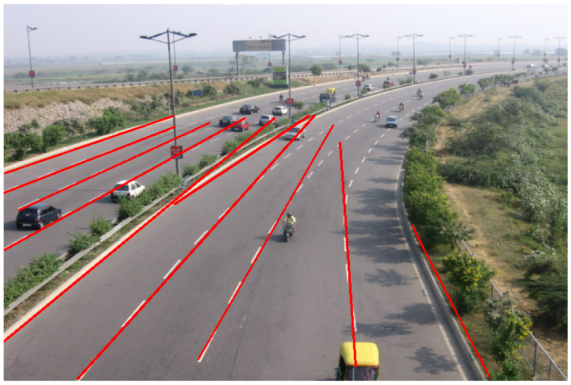
HoughLinesP Line detection

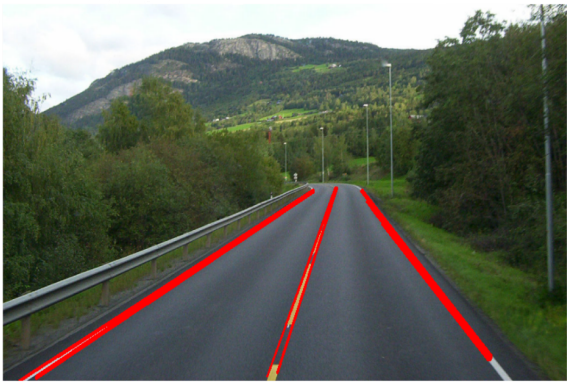
10. Merging of lines after Hough Transform —

HoughLinesP will give various lines but those lines are numerous in counts with similar lines being counted more than once. We need to apply a merging algorithm to merge similar lines. This algorithm works on the principle of maintaining min angle and minimum distance between points of two lines.

After performing, the merging and grouping of lines, the results obtained are as follow -







4. Additional Implementation for Stop sign detection —

We also need to detect a stop sign for the additional mark. Stop signs detections are the implementation of object detection algorithms. Stop signs have unique features — like Red background, White Boundary, Written “STOP” Text which can be detected individually to classify the object as a stop sign.

Here HAAR Cascade Classifier algorithm is used to detect stop signs. The detection of stop sign is done using CascadeClassifier’s detectMultiScale method that analyzes each block of the image. To plot a circle around the stop sign we use OpenCV’s circle function.