

## 19CSE435 – COMPUTER VISION

### CASE STUDY

#### *Road Lane Detection for Autonomous Vehicles*

**Name:** *Manoj Parthiban*

**Abstract:**

Road lane detection plays a crucial role in autonomous driving and advanced driver assistance systems. This study presents a road lane detection algorithm based on computer vision techniques using OpenCV. The proposed approach utilizes edge detection, perspective transformation, and Hough line transformation to extract and classify road lane markings. Experimental results demonstrate accurate and real-time lane detection, enabling effective vehicle navigation and lane departure warning systems for enhanced road safety.

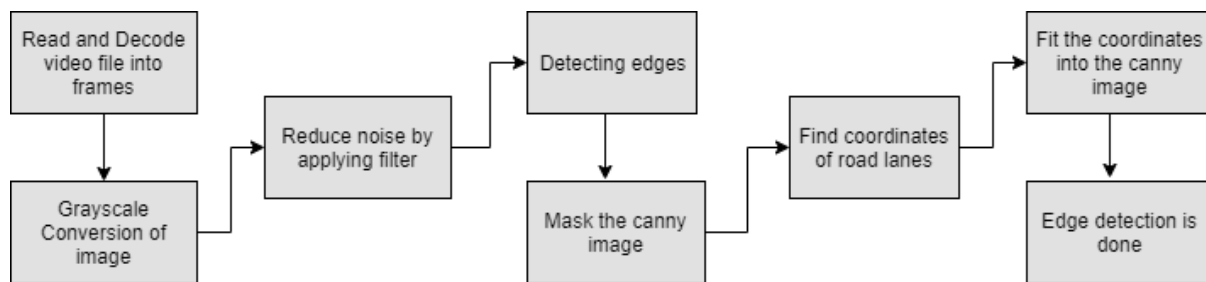
**Environment Variability:**

In addition to the intended application of the vision lane detection system, it is important to evaluate the type of conditions that are expected to be countered. Road markings can vary greatly not only between regions, but also over nearby stretches of road. Roads can be marked by well-defined solid lines, segmented lines, circular reflectors, physical barriers, or even nothing at all. The road surface can be comprised of light or dark pavements or combinations. An example of the variety of road conditions can be seen in Fig, some roads show a relatively simple scene with both solid line and dashed line lane markings. Lane position in this scene can be considered relatively easy because of the clearly defined markings and uniform road texture. But in other complex scene in which the road surface varies and markings consist of circular reflectors as well as solid lines the lane detection will not be an easy task. Furthermore, shadowing obscuring road markings makes the edge detection phase more complex. Along with the various types of markings and shadowing, weather conditions, and time of day can have a great impact on the visibility of the road surface as shown in Figure. All these circumstances must be efficiently handled in order to achieve an accurate vision system.



**Steps involved in lane detection:**

- 1) Apply Color Selection
- 2) Apply Canny edge detection.
- 3) Apply Gray scaling to the images.
- 4) Apply Gaussian smoothing.
- 5) Perform Canny edge detection.
- 6) Determine the region of interest.
- 7) Apply Hough transform.
- 8) Average and extrapolating the lane lines.
- 9) Apply on video streams.



The first thing the algorithm does is to convert the image to a grayscale image in order to minimize the processing time. Secondly, as presence of noise in the image will hinder the correct edge detection. Therefore, we apply canny edge detection algorithm to make the edge detection more accurate. Then the edge detector is used to produce an edge image by using canny filter with automatic thresholding to obtain the edges, it will reduce the amount of learning data required by simplifying the image edges considerably. Then edged image sent to the line detector after detecting the edges which will produces a right and left lane boundary segment. The projected intersection of these two-line segments is determined and is referred to as the horizon. The lane boundary scan uses the information in the edge image detected by the Hough transform to perform the scan. The scan returns a series of points on the right and left side. Finally, pair of hyperbolas is fitted to these data points to represent the lane boundaries.

**i. Image capturing:**

The input data is a color image sequence taken from a moving vehicle. A color camera is mounted inside the vehicle at the front-view mirror along the central line. It takes the images of the environment in front of the vehicle, including the road, vehicles on the road, roadside, and sometimes incident objects on the road. The on-board computer with image capturing card will capture the images in real time (up to 30 frames/second), and save them in the computer memory. The lane detection system reads the image sequence from the memory and starts processing. A typical scene of the road ahead is depicted by Figure 1. In order to obtain good estimates of lanes and improve the speed of the algorithm, the original image size was reduced to 620x480 pixels by Gaussian pyramid.

**ii. Conversion to grayscale:**

To retain the color information and segment the road from the lane boundaries using the color information this proved difficulties on edge detection and it will affect the processing time. In practice the road surface can be made up of many different colours due to shadows, different pavement style or age, which causes the color of the road surface and lane markings to change from one image region to another. Therefore, color image is converted into grayscale. However, the processing of grayscale images becomes minimal as compared to a color image. This function transforms a 24-bit, three-channel, color image to an 8-bit, single-channel grayscale image by forming a weighted sum of the red component of the pixel value \* 0.3 + Green component of the pixel value \* 0.59 + Blue component for the pixel value \* 0.11 the output is the Grayscale value for the corresponding pixel.

**iii. Noise reduction:**

Noise is a real-world problem for all systems and computer vision is no exception. The algorithms developed must either be noise tolerant or the noise must be eliminated. As presence of noise in our system will hinder the correct edge detection, so that noise removal is a pre requisite for efficient edge detection with the help of (F.H.D.) algorithm that removes strong shadows from a single image. The basic idea is that a shadow has a distinguished boundary. Removing the shadow boundary from the image derivatives and reconstructing the image should remove the. A shadow edge image can be created by applying edge-detection on the invariant image and the original image, and selecting the edges that exist in the original image but not in the invariant image and to reconstruct the shadow free image by removing the edges from the original image using a pseudo-inverse filter.

**iv. Edge detection:**

Lane boundaries are defined by sharp contrast between the road surface and painted lines or some type of non-pavement surface. These sharp contrasts are edges in the image. Therefore, edge detectors are very important in determining the location of lane boundaries. It also reduces the amount of learning data required by simplifying the image considerably, if the outline of a road can be extracted from the image. The edge detector implemented for this algorithm and the

one that produced the best edge images from all the edge detectors evaluated was the 'canny' edge detector. To find the maxima of the partial derivative of the image function  $I$  in the direction orthogonal to the edge direction, and to smooth the signal along the edge direction.

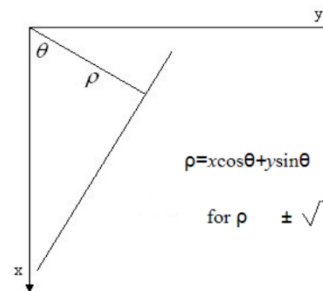
Thus, Canny's operator looks for the maxima of :

$$G_{\sigma} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{x^2+y^2}{2\sigma^2}\right] \quad n = \frac{\nabla G * I}{|\nabla G * I|}$$

The canny edge detector also has a very desirable characteristic in that it does not produce noise like the other approaches.

**v. Line detection:**

The line detector used is a standard Hough transform with a restricted search space. The standard Hough transforms searches for lines using the equation shown in Figure.

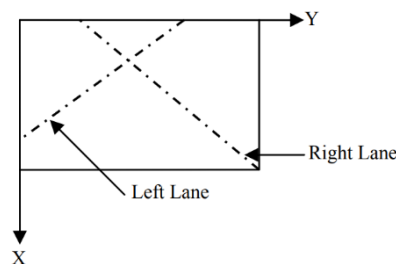


$\rho = x \cos \theta + y \sin \theta$  over the range of  $-90^\circ$  to  $90^\circ$  for  $\theta$  and  
for  $\rho \pm \sqrt{\text{rows}^2 + \text{cols}^2}$

The restricted Hough transform was modified to limit the search space to  $45^\circ$  for each side. Also, the input image is split in half yielding a right and left side of the image. Each the right and left sides are searched separately returning the most dominant line in the half image that falls within the  $45^\circ$  window. The horizon is simply calculated using the left and right Hough lines and projecting them to their intersection.

**vi. Lane boundary scan:**

The lane boundary scan phase uses the edge image the Hough lines and the horizon line as input. The edge image is what is scanned and the edges are the data points it collects. The scan begins where the projected Hough lines intersect the image border at the bottom of the image. Once that intersection is found, it is considered the starting point for the left or right search, depending upon which intersection is at hand. From the starting point, the search begins a certain number of pixels towards the centre of the lane and then proceeds to look for the first edge pixel until reaching a specified number of pixels after the maximum range. The range will be set to the location of the previously located edge pixel plus a buffer number of pixels further.



**vii. Hyperbola Fitting:**

The hyperbola pair fitting phase uses the two vectors of data points from the lane scan as input. A least squares technique is used to fit a hyperbola to the data. One hyperbola is fit to each of the vectors of data points; however, they are solved in simultaneously since they are a pair model. The parameters of the two hyperbolas are related because they must converge to the same point, due to the geometry of the roadway. The formula for expressing the lane boundary as a hyperbola.

$$u = \frac{k}{u-h} + b(v-h) + c$$

u and v are the x- and y-coordinate in the image reference frame, h is the Y-coordinate of the horizon in the image reference frame, and k, b and c are the parameters of the curve, which can be calculated from the shape of the lane.

### Experimental Results:

#### Code:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
import os
import glob
from moviepy.editor import VideoFileClip

def RGB_color_selection(image):

    #White color mask
    lower_threshold = np.uint8([200, 200, 200])
    upper_threshold = np.uint8([255, 255, 255])
    white_mask = cv2.inRange(image, lower_threshold, upper_threshold)

    #Yellow color mask
    lower_threshold = np.uint8([175, 175, 0])
    upper_threshold = np.uint8([255, 255, 255])
    yellow_mask = cv2.inRange(image, lower_threshold, upper_threshold)

    #Combine white and yellow masks
    mask = cv2.bitwise_or(white_mask, yellow_mask)
    masked_image = cv2.bitwise_and(image, image, mask = mask)

    return masked_image

def convert_hsv(image):

    return cv2.cvtColor(image, cv2.COLOR_RGB2HSV)

def HSV_color_selection(image):

    #Convert the input image to HSV
    converted_image = convert_hsv(image)

    #White color mask
    lower_threshold = np.uint8([0, 0, 210])
    upper_threshold = np.uint8([255, 30, 255])
    white_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

    #Yellow color mask
    lower_threshold = np.uint8([18, 80, 80])
    upper_threshold = np.uint8([30, 255, 255])
    yellow_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

    #Combine white and yellow masks
    mask = cv2.bitwise_or(white_mask, yellow_mask)
    masked_image = cv2.bitwise_and(image, image, mask = mask)

    return masked_image

def convert_hsl(image):

    return cv2.cvtColor(image, cv2.COLOR_RGB2HLS)

def HSL_color_selection(image):

    #Convert the input image to HSL
    converted_image = convert_hsl(image)
```

```

    #White color mask
    lower_threshold = np.uint8([0, 200, 0])
    upper_threshold = np.uint8([255, 255, 255])
    white_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

    #Yellow color mask
    lower_threshold = np.uint8([10, 0, 100])
    upper_threshold = np.uint8([40, 255, 255])
    yellow_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

    #Combine white and yellow masks
    mask = cv2.bitwise_or(white_mask, yellow_mask)
    masked_image = cv2.bitwise_and(image, image, mask = mask)

    return masked_image

def gray_scale(image):

    return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

def gaussian_smoothing(image, kernel_size = 13):

    return cv2.GaussianBlur(image, (kernel_size, kernel_size), 0)

def canny_detector(image, low_threshold = 50, high_threshold = 150):

    return cv2.Canny(image, low_threshold, high_threshold)

def region_selection(image):

    mask = np.zeros_like(image)
    #Defining a 3 channel or 1 channel color to fill the mask with depending on the
input image
    if len(image.shape) > 2:
        channel_count = image.shape[2]
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255
    #We could have used fixed numbers as the vertices of the polygon,
    #but they will not be applicable to images with different dimesnions.
    rows, cols = image.shape[:2]
    bottom_left = [cols * 0.1, rows * 0.95]
    top_left = [cols * 0.4, rows * 0.6]
    bottom_right = [cols * 0.9, rows * 0.95]
    top_right = [cols * 0.6, rows * 0.6]
    vertices = np.array([[bottom_left, top_left, top_right, bottom_right]],
dtype=np.int32)
    cv2.fillPoly(mask, vertices, ignore_mask_color)
    masked_image = cv2.bitwise_and(image, mask)
    return masked_image

def hough_transform(image):

    rho = 1 #Distance resolution of the accumulator in pixels.
    theta = np.pi/180 #Angle resolution of the accumulator in radians.
    threshold = 20 #Only lines that are greater than threshold will be
returned.
    minLineLength = 20 #Line segments shorter than that are rejected.
    maxLineGap = 300 #Maximum allowed gap between points on the same line to
link them
    return cv2.HoughLinesP(image, rho = rho, theta = theta, threshold = threshold,
minLineLength = minLineLength, maxLineGap = maxLineGap)

def draw_lines(image, lines, color = [255, 0, 0], thickness = 2):

    image = np.copy(image)
    for line in lines:

```

```

        for x1,y1,x2,y2 in line:
            cv2.line(image, (x1, y1), (x2, y2), color, thickness)
    return image

def average_slope_intercept(lines):

    left_lines    = [] #(slope, intercept)
    left_weights  = [] #(length,)
    right_lines   = [] #(slope, intercept)
    right_weights = [] #(length,)

    for line in lines:
        for x1, y1, x2, y2 in line:
            if x1 == x2:
                continue
            slope = (y2 - y1) / (x2 - x1)
            intercept = y1 - (slope * x1)
            length = np.sqrt(((y2 - y1) ** 2) + ((x2 - x1) ** 2))
            if slope < 0:
                left_lines.append((slope, intercept))
                left_weights.append((length))
            else:
                right_lines.append((slope, intercept))
                right_weights.append((length))
    left_lane = np.dot(left_weights, left_lines) / np.sum(left_weights) if
len(left_weights) > 0 else None
    right_lane = np.dot(right_weights, right_lines) / np.sum(right_weights) if
len(right_weights) > 0 else None
    return left_lane, right_lane

def pixel_points(y1, y2, line):

    if line is None:
        return None
    slope, intercept = line
    x1 = int((y1 - intercept)/slope)
    x2 = int((y2 - intercept)/slope)
    y1 = int(y1)
    y2 = int(y2)
    return ((x1, y1), (x2, y2))

def lane_lines(image, lines):

    left_lane, right_lane = average_slope_intercept(lines)
    y1 = image.shape[0]
    y2 = y1 * 0.6
    left_line = pixel_points(y1, y2, left_lane)
    right_line = pixel_points(y1, y2, right_lane)
    return left_line, right_line

def draw_lane_lines(image, lines, color=[255, 0, 0], thickness=12):

    line_image = np.zeros_like(image)
    for line in lines:
        if line is not None:
            cv2.line(line_image, *line, color, thickness)
    return cv2.addWeighted(image, 1.0, line_image, 1.0, 0.0)
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
import os
import glob
from moviepy.editor import VideoFileClip

%matplotlib inline
from moviepy import *
```

```

from IPython.display import HTML
from IPython.display import Image

def frame_processor(image):
    color_select = HSL_color_selection(image)
    gray         = gray_scale(color_select)
    smooth       = gaussian_smoothing(gray)
    edges        = canny_detector(smooth)
    region       = region_selection(edges)
    hough        = hough_transform(region)
    result       = draw_lane_lines(image, lane_lines(image, hough))
    return result

def process_video(test_video, output_video):
    input_video = VideoFileClip(os.path.join('test_videos', test_video),
    audio=False)
    processed = input_video.fl_image(frame_processor)
    processed.write_videofile(os.path.join('output_videos', output_video),
    audio=False)

from IPython.display import HTML
from base64 import b64encode
mp4 = open('/content/output2.mp4', 'rb').read()
data_url = "data:video/mp4;base64," + b64encode(mp4).decode()
HTML("""
<video width=600 controls>
    <source src="%s" type="video/mp4">
</video>
""" % data_url)

```

### Output:

<https://colab.research.google.com/drive/1iiQUtGIDOToRI5pMW8KHJs08PDPjnmgr?usp=sharing>

### **Summary and Conclusion:**

In this paper, a real time lane detection algorithm based on video sequences taken from a vehicle driving on highway was proposed. As mentioned above the system uses a series of images. Out of these series some of the different frames used are shown in the lane detection algorithm, Canny's algorithm conduct image segmentation and remove the shadow of the road. Since the lanes are normally long and smooth curves, we consider them as straight lines within a reasonable range for vehicle safety. The lanes were detected using Hough transformation with restricted search area. The proposed lane detection algorithm can be applied in both painted and unpainted road, as well as slightly curved and straight road. There remained some problems in the lane detection due to shadowing and the Hough line and horizon overlaid, then in the lower left edge with the lane boundary points overlaid and in few cases lane scan fails to track the correct lane.

Thank You 😊