CPS 843 Final Project: Road Lane Detection

Nathan Cheung
Department of Electrical, Computer & Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
nathan.cheung@ryerson.ca

Jules Gammad
Department of Electrical, Computer & Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
jgammad@ryerson.ca

Ivan Systerov
Department of Electrical, Computer & Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
isysterov@ryerson.ca

***Abstract*—The use of technology in modern vehicles, such as cameras and smart ECUs, has led to advancements in assisted and autonomous driving. One of the main obstacles in the development of autonomous vehicles is the accurate detection of roads and obstacles. Road lane detection algorithms, a type of computer vision technology, are used in autonomous vehicles to identify lanes on the road and track the vehicle's position within them. Previous research on lane detection has focused on methods such as geometrical modelling and convolutional neural networks. In this paper, we propose an approach to lane detection using a hybrid model that combines the Canny edge detector and Hough Transform to detect road lanes with real-time performance in mind. Our approach shows promising results in various challenging scenarios, such as complex road topologies and low lighting. GitHub source code: https://github.com/ncheungg/road-lane-detection**

I. Introduction

Technology has been increasingly incorporated in modern-day vehicles including GPS, Infotainment systems, smart ECU's, IoT embedded devices, radars, lidars, cameras, and many more. As such, developments in image processing and computer vision have given rise to the concept of assisted driving and more recently, autonomous driving. Such technologies are expected to grow rapidly towards full autonomy within the next decade and so, extensive research and development have been undertaken to solve the complex obstacles in the way.

The development of autonomous driving experiences two main bottlenecks: road/lane and obstacle detection [1]. In particular, road lane detection algorithms are a type of computer vision technology that is used in autonomous vehicles to identify the lanes on a road and track the vehicle's position within them. Lane detection is an essential component in the future of driving as it allows the vehicle to navigate safely and efficiently on the road whether by alerting the driver of dangers and obstacles ahead or fully taking control of the driving. Some of the features that require road lane detection are shown in Table I.

TABLE I
Current and Expected Automotive Features and their Road/Lane Detection Demands and/or Understandings

Feature	Description	Road/Lane Detection Demands and/or Understandings	Reference
Lane Departure Warning (LDW)	Issue warnings for near lane departure events	Host lane, short distance (40–50 m ahead)	[1]
Adaptive Cruise Control (ACC)	Follow the nearest vehicle in the host lane with safe headway distance	Host lane, short distance	[1]
Lane keeping	Return the car to the lane center when unsignaled lane departure occurs	Host lane, short distance, higher reliability than LDW	[1]
Lane centering	Keep the car in the middle of the lane at all times	Host lane, medium distance, high reliability, lane split (non-linear lane topology) identification	[1]
Lane change assist	Autonomous lane change on demand	Multiple lanes, front and rear, large distance (150 m) ahead	[1]
Turn assist	Autonomous turn on driver demand or as part of automatic navigation	Multiple lanes, lane semantics (identify turning lanes), non-linear lane and road topology (splits and merges)	[1]
Full autonomous driving for paved roads	Autonomous driving in city and highway	All of the above plus complex road topologies such as junctions/roundabouts/road under construction	[1]
Full autonomous driving for cross country driving	Autonomous driving in non-paved areas	Full rough road understanding but somewhat easier than paved-road autonomy with respect to lack of lanes, sparser traffic	[1]

II. Past Work

A lane detection algorithm is a computer vision system that processes video inputs from cameras mounted on the vehicle and identifies the location and orientation of lane markings on the road. Some of the leading research on this includes a novel lane detection based on geometrical model and Gabor filter [2]. In this approach, Zhou *et al*. [2], propose a three stage algorithm: 1) calibration of mounted camera, 2) estimation of lane model parameters using local Hough transform, and 3) model matching the lanes for best fitted lane model. Their results show robustness in proper lane perception and modelling (especially with shadows and poor lighting conditions) which is crucial to advancing the LDW automotive feature [2].

Moreover, ZuWhan [3], proposes a multistage approach that includes using artificial neural networks to detect road lane markings, then using the RANSAC algorithm and particle filtering to find the lane boundaries, and finally, using probabilistic reasoning to group the lane boundaries. His approach shows over 80% accuracy in lane detection, and around 16% in failed lane detection which is mostly the cause of poor image quality [3].

Another popular approach to lane detection is based on convolutional neural networks (CNNs), which are a type of machine learning model that is well-suited to image processing tasks. CNNs can be trained to identify lanes in images by providing them with a large number of annotated examples of lane images. The CNN can then learn to extract features from the images that are relevant for identifying lanes, and use these features to make predictions about the presence of lanes in new images. One such approach is with Li *et al*. [4], where they propose a 2-stage neural network: 1) a multitask deep CNN that extracts the latent features of the image for road lane markings and their geometric attributes, and 2) a recurrent neural network (RNN) that uses its internal memory to construct a continuous structure of the data (i.e. lane markings) throughout the frames. The results from their research was effective in identifying road lanes with an emphasis in traffic situations which is a major breakthrough in autonomous city driving [4].

III. Methodology

While previous works use machine and deep learning techniques to detect edges, we propose a simpler, yet elegant, way of detecting edges that require less computational power using the Canny edge detector algorithm. The Hough transform is then applied to the processed image to detect and display the lines within a region of interest. Finally, we localize the left and right side of the image and then apply an averaging method to estimate the lines which represent the road lane markings.

A. Canny Edge Detector

The canny edge detection algorithm is a multi-stage approach to detect an image. Canny [5], first applies Gaussian smoothing to the input image to reduce the noise and other irrelevant details in the image. Gaussian smoothing requires an $n \times n$ kernel represented in (1), where $h(x, y)$ is the value of the filter based on the center of the image. The resulting filter is then convolved with the original image to get the smoothed image.

$$h(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

Next, Canny [5], uses the first-order derivative of Gaussian to compute the gradient magnitude to calculate the edge strength in (2), and gradient direction to indicate the orientation of the image in (3).

$$\begin{aligned} M(x,y) &= \sqrt{g_x^2 + g_y^2} \\ &\approx |g_x| + |g_y| \\ \text{where } grad(f) &= \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} \end{aligned} \tag{2}$$

$$\alpha(x,y) = tan^{-1} \begin{bmatrix} \frac{g_y}{g_x} \end{bmatrix} \tag{3}$$

The third step is to apply non-maximum suppression to the gradient magnitude image, which removes any pixels that are not local maxima along the gradient direction [5]. This helps to thin the edges and eliminate any false edges that may have been introduced by the noise in the image. Finally, Canny [5], applies hysteresis (double) thresholding to identify the edges in the image. This involves applying two threshold values, a high threshold and a low threshold, to the gradient magnitude image. Pixels with a gradient magnitude above the high threshold are definitely edges, while pixels with a gradient magnitude below the low threshold are definitely not edges. Pixels with a gradient magnitude between the two thresholds are considered edges if they are connected to a pixel with a gradient magnitude above the high threshold. This step helps to further eliminate any false edges and produce a clean, thin edge map of the image.

B. Generalized Hough Transform for Lines

The Hough transform is a technique used to detect lines or other shapes in an image. Hough [6], bases his approach on the idea of representing lines in the image in a parameter space, where each point in the space represents a line in the image. To detect lines in the image using the Hough transform, we first need to iterate over each point (x, y) in the binary

image and compute the corresponding point (x', y') in the parameter space. This is represented in (4), where θ and r are the orientation and distance of the line in the image, respectively.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta & r \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{4}$$

We can then increment the value of the point (x', y') by 1 to "vote" for the line represented by that point [6]. After all the points in the binary image have been processed, the points in the parameter space with the highest values are considered to be the lines in the image. These points can then be converted back into lines in the image space and used to detect the lines in the original image [6].

C. Final Processing

The result from combination of the Canny and Hough transform results in all the lines included within the region of interest. Therefore, a final processing step must be added. These steps can be seen in the following:

1) Horizontal and nearly horizontal lines are removed
2) Lines are bucketed between left & right lane lines
3) Average line equations in each bucket
4) Stitch the two detected lane lines back to the original image

Firstly, to remove the horizontal lines detected, we apply a threshold value of $< \pm 10^\circ$ to lines which are considered to be horizontal or nearly horizontal lines. This equation is shown in (5). Then, the remaining lines are bucketed into “right" and “left" buckets based on a positive or negative slope respectively. Notice that a positive slope corresponds to the “right" bucket as the Y-axis of the cartesian plane of the images processed “Fig 1" is inversed. The equation of the lines in each bucket are then averaged, and when identifying lane lines in a video frame, the identified lane lines are averaged again with a sliding window of the last 10 frames. The averaging equation can be seen in (6). This is to ensure frames that happen to miss a lane line detection would still have memory from previous frames. Finally, a line is drawn from the max y value seen in the overall buckets to approximately the middle of the image at $y = 190$.

$$\theta = \left| tan^{-1} \left(\frac{y_2 - y_1}{x_2 - x_1} \right) \right| < 10^\circ \tag{5}$$

$$l_{avg} = \frac{1}{n} \sum_{i=1}^{n} l_i$$
$$\text{where } l_i = a_i x + b_i y + c_i = 0 \tag{6}$$

IV. Approach

We collaborated on road lane detection algorithm using Jupyter Notebooks on Google Colab. We opted to use modern industry standards such as Python3 and OpenCV to perform image processing. Source code for all steps and additional documentation are provided in (Appendix A).

Our algorithm can be broken down into the following steps:

1) Fetch video frame
2) Resize frame to 640×360
3) Convert frame to RGB colorspace
4) Convert frame to grayscale for processing
5) Define a region of interest
6) Add gaussian blur
7) Find edges with canny edge detector
8) Mask edges with region of interest
9) Find lines using hough transform
10) Remove nearly horizontal lines
11) Get left & right lane lines
12) Display output frame

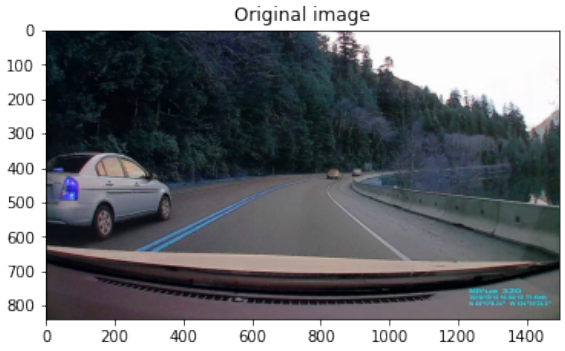

Fig. 1. Fetch Video Frame

Fig. 2. Resize Frame to 640×360

V. Experimental Results

The results seen between “Fig 1" and “Fig 12" from Section IV shows a good and effective result when detecting road lane lines. Double lane markings seen on the left side of “Fig 8" was generalized into 1 lane line after the Hough

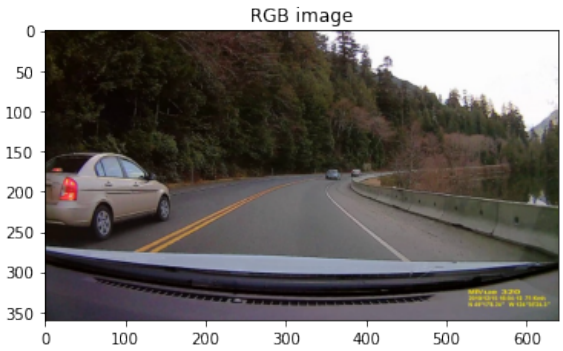

Fig. 3. Convert Frame to RGB Colorspace

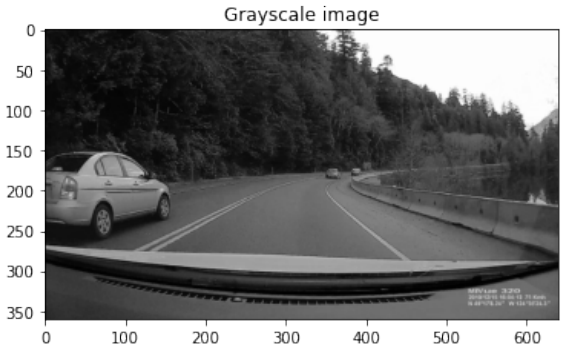

Fig. 4. Convert Frame to Grayscale for Processing

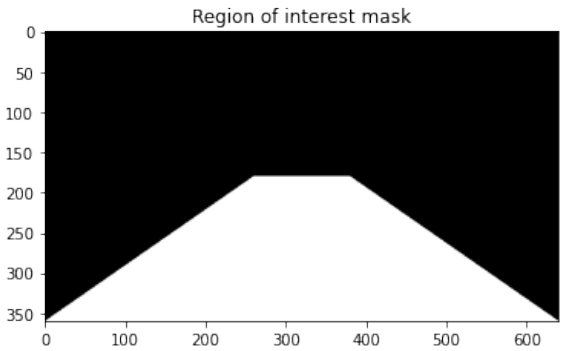

Fig. 5. Define a Region of Interest

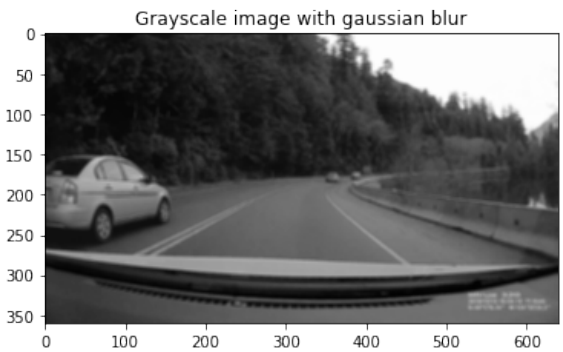

Fig. 6. Add Gaussian Blur

Fig. 7. Find Edges with Canny Edge Detector

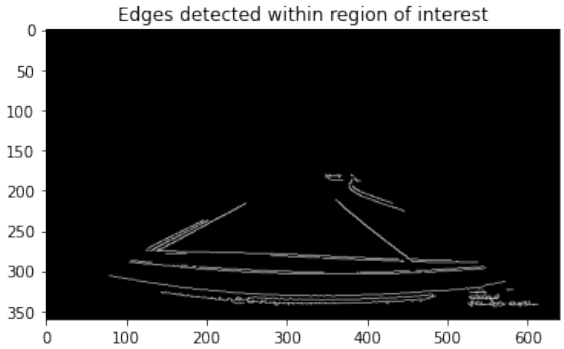

Fig. 8. Mask Edges with Region of Interest

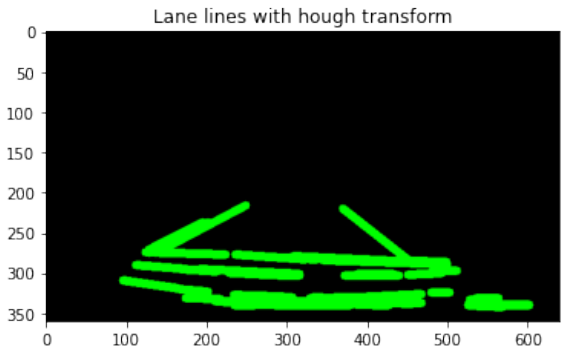

Fig. 9. Find Lines using Hough Transform

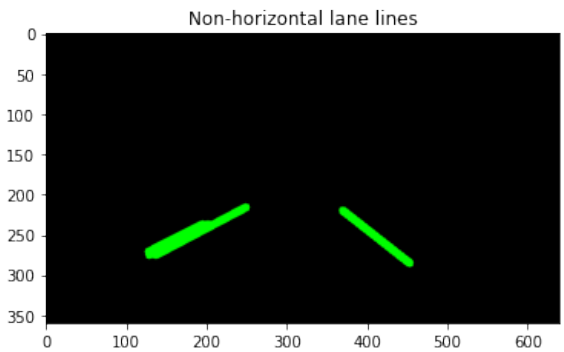

Fig. 10. Remove Nearly Horizontal Lines

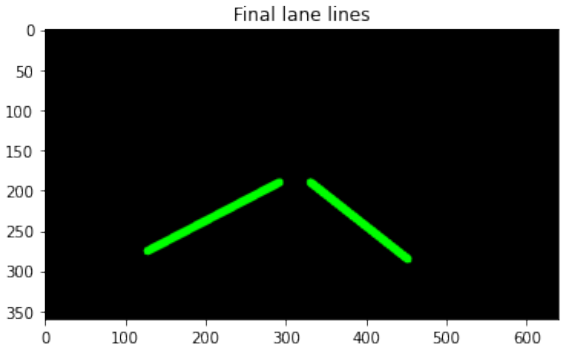

Fig. 11. Get Left & Right Lane Lines

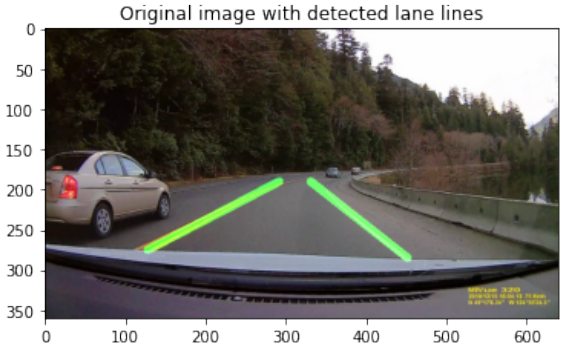

Fig. 12. Display Output Frame

transform, and objects with long straight patterns such as the barricade on the right side of "Fig 1" was properly ignored as it is not a lane line.

Moreover, we applied our algorithm to a live dashcam video of a car driving in a highway shown in "Fig 13(a)". The first frame of the video show the original input image frame to be processed. From "Fig 13(b) – (i)", the algorithm works with relatively good detection and extrapolation of the lane markings given that the lane markings are not connected to each other (i.e. dashed lines).

One concern is the region of interest to which the algorithm localizes the lane markings. Because we set a strict region of interest, the lane lines are not fully detected until it comes into the region of the next frame. Furthermore, there are some inconsistent line detection in the bottom right side of the frames. This is due to the disconnected lane markings, poor lighting conditions, and the region of interest.

The time taken for the algorithm was also calculated using the Python autotime library and averaged over many iterations. The calculation for each step of our algorithm for one frame is shown in Table II. In total, our algorithm took $246.68ms$ to detect and produce the lane markings of one frame. Taking into account that more than 80% of the time taken is pre-processing the input frame, if we take pre-processing out of the calculation, our algorithm is expected to run at $31.91ms$. Given a modern day 30fps video camera, that gives us enough time to process each frame without delay.

Overall, our results show the practicality of our algorithm to real-time systems in autonomous vehicles. The algorithm provides good lane detection at a speed that is usable for a real-time system to make quick decisions in an assisted/autonomous driving environment.

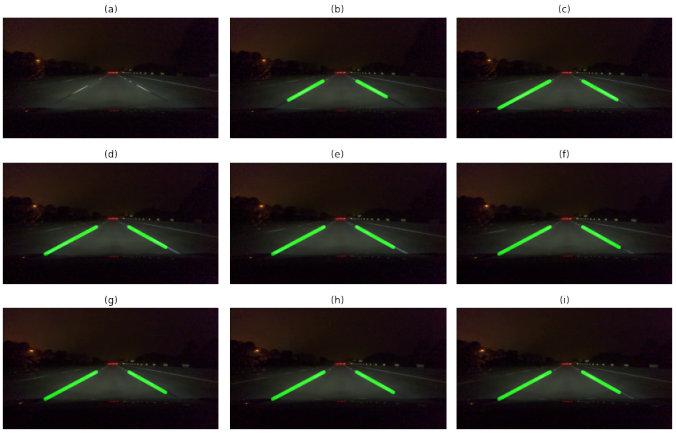

Fig. 13. Lane Line Detection of 9 Consecutive Video Frames

TABLE II
STEP BY STEP EXECUTION TIME FOR A SINGLE FRAME

Step	Average Time (ms)
1. Fetch video frame	199
2. Resize frame to 640×360	5.74
3. Convert frame to RGB colorspace	4.51
4. Convert frame to grayscale for processing	5.52
5. Define a region of interest	1.95
6. Add gaussian blur	2.80
7. Find edges with canny edge detector	3.63
8. Mask edges with region of interest	1.22
9. Find lines using hough transform	10.8
10. Remove nearly horizontal lines	3.53
11. Get left & right lane lines	7.98
12. Display output frame	
Total	**246.68**

VI. CONCLUSION

The goal of this assignment was to develop a machine vision system for detecting lanes on a road using Canny edge detection and Hough transform. This system is designed with real-time operation in mind to be applied in autonomous vehicles to help them navigate roads safely.

The Canny edge detection algorithm was used to identify the edges in the road images, which were then fed into the Hough transform algorithm to identify the lines representing the lanes. The Hough transform algorithm uses a voting system to identify the most likely lines in the image, which

were then used to draw the detected lanes on the road.

Overall, the lane detection system performed well, accurately detecting the lanes in a variety of road conditions and lighting conditions. However, there are several ways in which the system could be improved.

One potential improvement would be to use more modern computer vision techniques using convolutional neural networks and large datasets. Given a large image set with labeled lane line markings, a neural network could be trained to accurately recognize lane line markings under many edge cases where the canny edge detector & Hough transform algorithm fail.

Furthermore, camera parametric variables can be incorporated into the algorithm to add additional information into the system, such as information about the road surface or the surrounding environment. This could help the system to better understand the context of the road and improve its ability to detect lanes.

Finally, incorporation of more advanced sensors, such as lidar and radar, can be used to create a 3D map of the road and its surrounds. This additional information could assist the road lane detection problem with additional information per video frame to help train a neural network.

Ultimately, this lane detection system using Canny edge detection and Hough transform is a promising approach to detecting lanes on a road. Further improvements in faster and accurate techniques as well as hardware could better meet the needs of autonomous vehicles.

VII. Contributions

All group members contributed equally to the project. We had meetings and work sessions to discuss the progress of the project and what needs to be done. In particular, each group member had certain expertise that helped the group project move along seamlessly. Nathan was knowledgeable in the scope of current projects and online repositories regarding lane detection algorithm. Jules was keen on finding the right research papers and report writing, and Ivan was a great all-around member that helped with managing the project. Overall, there were no outliers for finishing the project and all members contributed as best they could.

References

[1] A. Bar Hillel *et al*, "Recent progress in road and lane detection: a survey," *Machine Vision and Applications*, vol. 25, *(3)*, pp. 727-745, 2014.

[2] S. Zhou *et al*, "A novel lane detection based on geometrical model and gabor filter," in 2010, . DOI: 10.1109/IVS.2010.5548087.

[3] Z. Kim, "Robust Lane Detection and Tracking in Challenging Scenarios," *IEEE Transactions on Intelligent Transportation Systems*, vol. 9, *(1)*, pp. 16-26, 2008.

[4] J. Li *et al*, "Deep Neural Network for Structural Prediction and Lane Detection in Traffic Scene," *IEEE Transaction on Neural Networks and Learning Systems*, vol. 28, *(3)*, pp. 690-703, 2017.

[5] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, *(6)*, pp. 679-698, 1986.

[6] D. H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, *(2)*, pp. 111-122, 1981.

APPENDIX

A. *Road_Lane_Detection.ipynb*

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
import matplotlib.animation as animation
import urllib
from matplotlib import rc
```

```
##### fetch dashcam image
def fetch_image():
  image_url = "https://raw.githubusercontent.com/ncheungg/road-lane-detection/master/assets/dashcam.png"
  url_response = urllib.request.urlopen(image_url)
  img_array = np.array(bytearray(url_response.read()), dtype=np.uint8)
  return cv2.imdecode(img_array, -1)

image_original = fetch_image()

plt.imshow(image_original)
plt.title("Original image")
plt.show()
```

```
##### resize image to 640x360
def resize_image(image):
  return cv2.resize(image, (640, 360))

image_resized = resize_image(image_original)

plt.imshow(image_resized)
plt.title("Resized image to 640x360")
plt.show()
```

```
##### convert image to RGB
def convert_image_to_rgb(image):
  return cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

image_rgb = convert_image_to_rgb(image_resized)

plt.imshow(image_rgb)
plt.title("RGB image")
plt.show()
```

```
##### convert to grayscale image
def convert_image_to_gray(image):
  return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

image_gray = convert_image_to_gray(image_rgb)

plt.imshow(image_gray, cmap='gray', vmin = 0, vmax = 255)
plt.title("Grayscale image")
plt.show()
```

```
##### define a region of interest for a 640x360 image
def create_region_of_interest(image):
  p1 = 0, 360
  p2 = 640, 360
  p3 = 380, 180
  p4 = 260, 180

  points = np.array([p1, p2, p3, p4])
  mask = np.zeros_like(image)
  cv2.fillPoly(mask, pts=[points], color=255)
  return mask

region_of_interest = create_region_of_interest(image_gray)

plt.imshow(region_of_interest, cmap='gray', vmin = 0, vmax = 255)
plt.title("Region of interest mask")
plt.show()
```

```
##### add gaussian blur
def add_gaussian_blur(image):
  return cv2.GaussianBlur(image, (5, 5), 10)

image_blur = add_gaussian_blur(image_gray)

plt.imshow(image_blur, cmap='gray', vmin = 0, vmax = 255)
plt.title("Grayscale image with gaussian blur")
plt.show()
```

```
##### find edges using canny edge detector
def get_edges(image):
  return cv2.Canny(image, 50, 150)

image_edges = get_edges(image_blur)

plt.imshow(image_edges, cmap='gray', vmin = 0, vmax = 255)
plt.title("Edges detected with thresholds [50, 150]")
plt.show()
```

```
##### find edges in region of interest
def get_interesting_edges(image, mask):
  return cv2.bitwise_and(image, mask)

image_edges_region = get_interesting_edges(image_edges, region_of_interest)

plt.imshow(image_edges_region, cmap='gray', vmin = 0, vmax = 255)
plt.title("Edges detected within region of interest")
plt.show()
```

```
##### find lines using hough transform
def get_hough_lines(image):
  result = cv2.HoughLinesP(image, 2, np.pi/180, 75, np.array([]),
                           minLineLength=20, maxLineGap=15)
  return np.array([]) if result is None else result

lane_lines = get_hough_lines(image_edges_region)
image_lines = np.zeros_like(image_rgb)

# display our lines
for line in lane_lines:
  x1, y1, x2, y2 = line.reshape(4)
  cv2.line(image_lines, (x1, y1), (x2, y2), color=(0, 255, 0), thickness=10)

plt.imshow(image_lines)
plt.title("Lane lines with hough transform")
plt.show()
```

```
##### remove nearly horizontal lines

# function returns True if not nearly horizontal (+- 10 degrees)
def is_not_nearly_horizontal(line):
  x1, y1, x2, y2 = line.reshape(4)
  angle = np.arctan2(y2 - y1, x2 - x1) * 180 / np.pi
  return abs(angle) >= 10

def get_non_horizontal_lines(lines):
  return list(filter(is_not_nearly_horizontal, lines))

lane_lines_not_horizontal = get_non_horizontal_lines(lane_lines)
image_lines_non_horizontal = np.zeros_like(image_rgb)

# display our lines
for line in lane_lines_not_horizontal:
  x1, y1, x2, y2 = line.reshape(4)
  cv2.line(image_lines_non_horizontal, (x1, y1), (x2, y2), color=(0, 255, 0), thickness=10)

plt.imshow(image_lines_non_horizontal)
plt.title("Non-horizontal lane lines")
plt.show()
```

```
##### get left and right lane lines

# makes the final lane line given slope and low/high points of y values
def make_line(params, lowest, highest):
  m, b = params

  y1, y2 = lowest, highest
  x1 = int((y1 - b) / m)
  x2 = int((y2 - b) / m)
  return np.array([x1, y1, x2, y2])

def average_out_line(lines):
  if len(lines) == 0:
    return None

  line_params = []
  max_y = 0

  for line in lines:
    x1, y1, x2, y2 = line.reshape(4)
    m, b = np.polyfit((x1, x2), (y1, y2), 1)

    line_params.append((m, b))
    max_y = max(max_y, y1, y2)

  avg_params = np.average(line_params, axis=0)

  return make_line(avg_params, max_y, 190)

def get_left_and_right_lane_lines(lines):
  left_lines = []
  right_lines = []

  for line in lines:
    x1, y1, x2, y2 = line.reshape(4)
    m, b = np.polyfit((x1, x2), (y1, y2), 1)

    # positive slope is left line, negative slope is right line
    if m < 0:
      left_lines.append(line)
    else:
      right_lines.append(line)

  left_line = average_out_line(left_lines)
  right_line = average_out_line(right_lines)

  return left_line, right_line

final_lane_lines = get_left_and_right_lane_lines(lane_lines_not_horizontal)
image_final_lane_lines = np.zeros_like(image_rgb)

# display our lines
for line in final_lane_lines:
  if line is None:
    continue

  x1, y1, x2, y2 = line.reshape(4)
  cv2.line(image_final_lane_lines, (x1, y1), (x2, y2), color=(0, 255, 0), thickness=10)

plt.imshow(image_final_lane_lines)
plt.title("Final lane lines")
plt.show()
```

```
##### display input image with lane lines
def get_final_image(original_image, final_lane_lines):
  image_final_lane_lines = np.zeros_like(original_image)

  for line in final_lane_lines:
    if line is None:
      continue

    x1, y1, x2, y2 = line.reshape(4)
    cv2.line(image_final_lane_lines, (x1, y1), (x2, y2), color=(0, 255, 0), thickness=10)
```

```
  return cv2.addWeighted(image_final_lane_lines, 0.8, original_image, 1, 1)

image_rgb_with_lines = get_final_image(image_rgb, final_lane_lines)

plt.imshow(image_rgb_with_lines)
plt.title("Original image with detected lane lines")
plt.show()
```

```
##### import dashcam videos
videos = [
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video2.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video3.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video4.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video5.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video6.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video7.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video8.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video9.hevc',
  'https://github.com/ncheungg/road-lane-detection/raw/master/assets/video10.hevc',
]

captures = lambda x: cv2.VideoCapture(videos[x])
```

```
##### CVRoadLaneDetector class that processes frames
##### it stores a sliding window of the last 10 frames

class CVRoadLaneDetector:
  def __init__(self, image):
    self.window = []
    self.region_of_interest = create_region_of_interest(image)

  def get_average_lane_lines_in_window(self):
    left_lines = []
    right_lines = []
    left_max = right_max = 0

    for left_line, right_line in self.window:
      if left_line is not None:
        left_lines.append(left_line)

      if right_line is not None:
        right_lines.append(right_line)

    left_line = average_out_line(left_lines)
    right_line = average_out_line(right_lines)

    return left_line, right_line

  # the main detection algorithm
  def compute_frame(self, frame):
    frame_resized = resize_image(frame)
    frame_rgb = convert_image_to_rgb(frame_resized)
    frame_gray = convert_image_to_gray(frame_rgb)

    frame_edges = get_edges(frame_gray)
    frame_edges_in_region = get_interesting_edges(frame_edges, self.region_of_interest)
    lane_lines = get_hough_lines(frame_edges_in_region)
    lane_lines_not_horizontal = get_non_horizontal_lines(lane_lines)

    # add current lane lines to sliding window of last 10 frames
    current_frame_lane_lines = get_left_and_right_lane_lines(lane_lines_not_horizontal)
    self.window = self.window[-9:] + [current_frame_lane_lines]

    # average the last 10 frame's lane lines
    final_lane_lines = self.get_average_lane_lines_in_window()

    final_image = get_final_image(frame_rgb, final_lane_lines)
    return final_image
```

```
##### functions that processes/displays videos in notebook
```

```
##### NOTE: these are very expensive functions, use with caution

rc('animation', html='jshtml')

def display_video_original_processed(cap):
  ims = []
  fig = plt.figure(figsize=(9, 12));
  ax1 = fig.add_subplot(2, 1, 1)
  ax2 = fig.add_subplot(2, 1, 2)
  ax1.title.set_text("Original video in RGB")
  ax2.title.set_text("Processed video with lane lines")

  # initializes CVRoadLaneDetector class
  detector = CVRoadLaneDetector(image_gray)

  while True:
    ret, frame = cap.read()

    if not ret:
      break

    rgb_frame = convert_image_to_rgb(frame)
    processed_frame = detector.compute_frame(frame)

    im1 = ax1.imshow(rgb_frame)
    im2 = ax2.imshow(processed_frame)
    ims.append([im1, im2])

  return animation.ArtistAnimation(fig, ims, interval=50)
```

```
display_video_original_processed(captures(8))
```

```
##### additional
##### generates subplot of first 9 video frames

detector = CVRoadLaneDetector(image_gray)
cap = captures(8)
fig, axs = plt.subplots(3, 3, figsize=(15,10), gridspec_kw=dict(wspace=0.05, hspace=0))


for i in range(3):
  for j in range(3):
    ret, frame = cap.read()

    rgb_frame = convert_image_to_rgb(frame)
    processed_frame = detector.compute_frame(frame)

    axs[i, j].imshow(processed_frame)
    axs[i, j].axis('off')

axs[0, 0].set_title('(a)')
axs[0, 1].set_title('(b)')
axs[0, 2].set_title('(c)')
axs[1, 0].set_title('(d)')
axs[1, 1].set_title('(e)')
axs[1, 2].set_title('(f)')
axs[2, 0].set_title('(g)')
axs[2, 1].set_title('(h)')
axs[2, 2].set_title('(i)')

fig
```