ORIGINAL RESEARCH PAPER

Engineering

DETECTION OF LANE LINES FOR SELF-DRIVING CARS BY USING EFFECTIVE COMPUTER VISION TECHNIQUES WITH OPENCV IN PYTHON

KEY WORDS: Computer Vision, OpenCV, Lane Detection, Gradient and HLS Thresholding, Artificial Intelligence, Self driving vehicles,

Aryan Verma* Student, Shahpura, Bhopal-462039 (MP). *Corresponding Author

ABSTRACT

Presently computer vision is amongst the hottest topics in Artificial Intelligence and is being extensively used in Robotics, Detecting Objects, Classification of Images, Autonomous Vehicles & tracking, Semantic Segmentation along with photo correction in various apps. In Self driven cars/ vehicles, vision remains the main source of information for detecting lanes, traffic lights, pedestrian crossing and other visual features. [2]

I. INTRODUCTIONS

Innovations are powered by Science and they happen continuously with necessity of present requirement. One such future technological innovation is detection of driving lane lines by vehicle they could be manual or self driven. For example Self-driving cars, which are capable of navigating through the roads, sensing environmental inputs, fulfill the transportation capabilities without any human efforts. These vehicles analyse its surrounding through camera's, RADAR's, LIDAR's, GPS and process its navigational paths according to the data without any human support. This could be making a huge impact in the people lives who are differently able. Presently everyone is aware with Tesla, Google and others companies work in self -driving cars like Tesla Model S whose Autopilot handles highway driving and many more. Its main tasks include an automated vehicle that increases safety and reduce road accidents, and thus saving lives. In days to come many more complex and challenging tasks of road lane detection or boundaries detection related to self driven vehicles could be resolved. [3]

II. LITERATURE OVERVIEW

OpenCV refers to "Open Source Computer Vision" which is a library for computer vision and machine learning software's invented by Intel in 1999. OpenCV consists of C++, Python, Java and MATLAB interfaces and is supported by Windows, Linux, Android, and Mac OS.

Computer Vision is an allied field of Artificial Intelligence which trains the machines to interpret and understand the real-world scenarios. Its combines various techniques like cameras, videos along with deep learning models machines that could accurately identify, classify objects automatically and then decisions considering the data interpretation. Today computer vision technology is used for various purposes like Image segmentation, object detection, Edge detection, Facial recognition, Pattern detection etc.

Lane detection Implementation undoubtedly Lane lines are important part of indicating a traffic flow while driving any vehicle. To avoid accidents it is essential to remain in a single lane and to avoid crossing lanes. OpenCV and Python are good starting point for building your own self driving to design lane detection cars. Refer figure 1

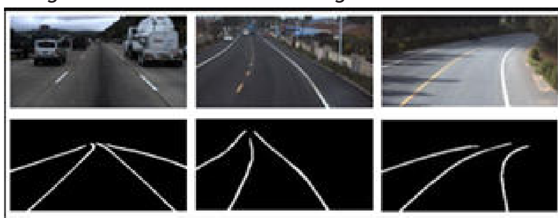

Figure 1: road images (first row) along with their detected lanes (second row):

I. RESEARCH OVERVIEW

There are multiple ways to perform lane detection either by using Learning-based approaches, such as training a deep learning model on an annotated video dataset or use a pre-trained model Simple Lane Detection is an detection technique which detects straight lane. We will be using "Atom" text editor or "Sublime" whatever you like working with. The purpose of this is to develop a program that can identify lane lines in a picture or a video. Here's the structure of our lane detection pipeline.

Step 1: Reading Images: in general, when humans drive, they use eyes and common sense to drive. These inputs can easily identify the lanes on the roads and do the steering along the road. But for machines, it's a difficult task hence here computer vision comes into picture.

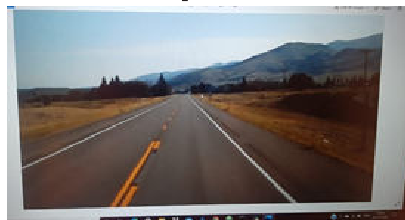

Figure 2: frame from the video

Step 2: Generally we have four lanes separated by white-colored lane markings. So, to detect a lane, we must detect the white markings on either side of that lane. Important question is how to detect the lane markings?

Step 3: There can be so many other objects in the scene apart from the lane markings like, vehicles on the road, road-side barriers, street-lights, etc. Also in real time scenarios/ video, a scene changes at frame by frame.

Step 4: Hence, before solving the lane detection problem solution for finding ways to ignore the unwanted objects from the driving scene had been determined.

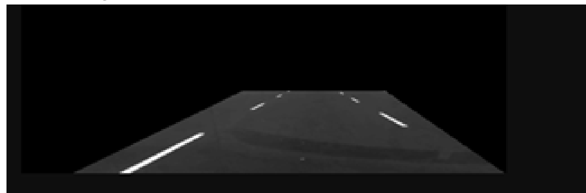

Figure 3

Step 5: This can be done by narrowing down the area of interest. Instead of working with the entire frame, we can work with only a part of the frame. In the image below, apart from the lane markings, everything else has been hidden in the frame. As the vehicle would move, the lane markings would fall more or less in this area only[4]

Reduce Noise and Smoothen Image

Images consist of Noise which causes blurring, so this need to be removed. Image with noises creates false edges and can ultimately impact the edge detection which is a very crucial step in lane line detection. We will use the Gaussian Filter to blur the image. A Gaussian filter non-uniform low pass filter.

IV. APPLICATION DESIGN

To understand the application design process refer following steps in detail:-

1) Compute the camera calibration matrix and distortion coefficients.

All cameras use lenses and main problem with these lenses is that they have some radial distortion. To remove this distortion, we used OpenCV functions on chessboard images to calculate the correct camera matrix and distortion coefficients. This can be achieved by finding the inside corners within an image and using that information to un-distort the image. Fig 4 shows the chessboard image on the left and the inside corners within this image detected on the right. [5]

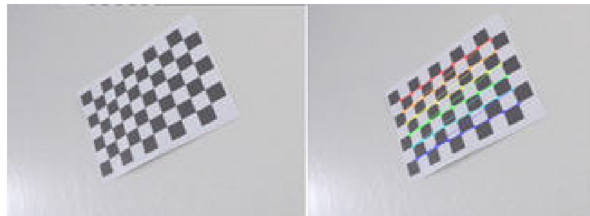

Figure 4: Calculating the camera matrix and distortion coefficients by detecting inside corners in a chessboard image (Source: Udacity)

The distortion matrix was used to un-distort a calibration image and provides a demonstration that the calibration is correct. An example shown here in Fig 2, shows the before/after results after applying calibration to un-distort the chessboard image. [5]

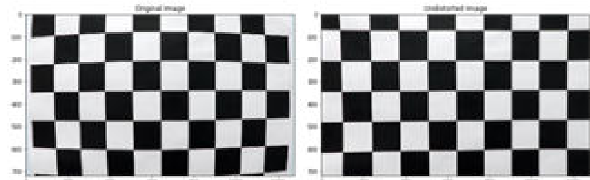

Figure 5: Before and after results of un-distorting a chessboard image (Source: Udacity)

2) Apply a distortion correction to raw images.

The calibration data for the camera that was collected in step 1 can be applied for raw images to apply distortion correction. An example image is shown here in Fig 6. It may be harder to see the effects of applying distortion correction on raw images compared to a chessboard image, but if you look closer at right of the image for comparison, this effect becomes more obvious when you look at the white car that has been slightly cropped along with the trees when the distortion correction was applied. [5]

Figure 6: Before and after results of un-distorting an example image (Source: Udacity)

3) Use color transforms, gradients, etc., to create a thresholded binary image.

The idea is to create an image processing pipeline where the lane lines can be clearly identified by the algorithm. There are a number of different ways to get to the solution by playing around with different gradients, thresholds and color spaces. We experimented with a number of these techniques on several different images and used a combination of thresholds, color spaces, and gradients. Worked on the following combination to create my image processing pipeline: [5]

'S channel' thresholds in the HLS color space and 'V channel' thresholds in the HSV color space, along with gradients to detect lane lines. For example, a final binary thresholded image is shown in Fig 7, where the lane lines are clearly visible. [5]

Figure 7: Before and after results of applying gradients and thresholds to generate a binary thresholded image (Source: Udacity)

4) Apply a perspective transform to generate a "bird's-eye view" of the image.

Images have perspective which causes lanes lines in an image to appear like they are converging at a distance even though they are parallel to each other. It is easier to detect curvature of lane lines when this perspective is removed. This can be achieved by transforming the image to a 2D Bird's eye view where the lane lines are always parallel to each other. Since we are only interested in the lane lines, I selected four points on the original un-distorted image and transformed the perspective to a Bird's eye view as shown in Fig 8 below. [5]

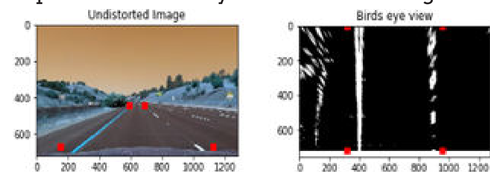

Figure 8: Region of interest perspective warped to generate a Bird's-eye view (Source: Udacity)

5) Detect lane pixels and fit to find the lane boundary.

To detect the lane lines, we used convolution which is the sum of the product of two separate signals: the window template and the vertical slice of the pixel image. Also, the sliding window method to apply the convolution will maximize the number of hot pixels in each window. The window template is slid across the image from left to right and any overlapping values are summed together, creating the convolved signal. The peak of the convolved signal is where the highest overlap of pixels are and it is the most likely position for the lane marker. Methods have been used to identify lane line pixels in the rectified binary image. The left and right lines have been identified and fit with a curved polynomial function. Example images with line pixels identified with the sliding window approach and a polynomial fit overlapped are shown in Fig 9. [5]

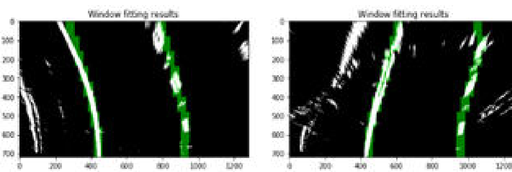

Figure 9: Sliding window fit results (Source: Udacity)

6) Determine the curvature of the lane and vehicle position with respect to the center of the car.

The measurements of lane lines were taken to estimate extend of the road curve, along with the vehicle position with respect to the center of the lane. It was assumed that the camera is mounted at the center of the car. [5]

7) Warp the detected lane boundaries back onto the original image and display numerical estimation of lane curvature and vehicle position.

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. Fig 10 demonstrates that the lane boundaries were correctly identified and warped back on to the original image. [5]

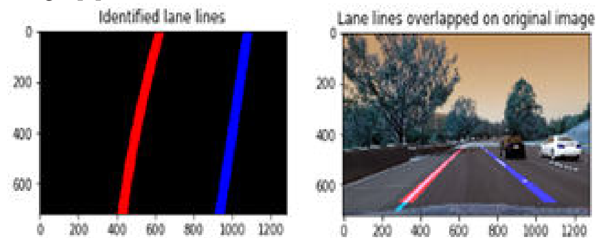

Figure 10: Lane line boundaries warped back onto original image (Source: Udacity)

An example image with lanes, curvature, and position from center is shown in Fig 11.

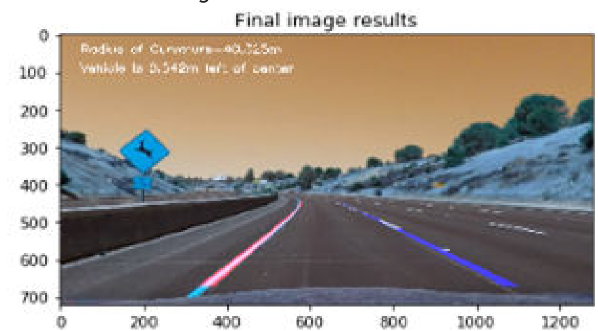

Figure 11: Detected lane lines overlapped on to the original image along with curvature radius and position of the car (Source: Udacity)

The above process was applied to each frame of a video and the end results with the key steps are useful to predict model across different frames. [5]

V. CODING

```
import cv2
import numpy as np
def make_coordinates(image,line_parameters):
  #slope,intercept=line_parameters
  try:
   slope,intercept = line_parameters
  except TypeError:
   slope,intercept = 0,0
  y1=image.shape[0]
  y2=int(y1*(3/5))
  x1=int((y1-intercept)/slope)
  x2=int((y2-intercept)/slope)
  return np.array([x1,y1,x2,y2])
def average_slope_intercept(image,lines):
  left_fit=[]
  right_fit=[]
  for line in lines:
    x1,y1,x2,y2=line.reshape(4)
    parameters=np.polyfit((x1,x2),(y1,y2),1)
    slope= parameters[0]
    intercept=parameters[1]
    if slope < 0:
    left_fit.append((slope,intercept))
    else:
  right_fit.append((slope,intercept))
  left_fit_average=np.average(left_fit,axis=0)
  right_fit_average=np.average(right_fit,axis=0)
  left_line=make_coordinates(image,left_fit_average)
  right_line=make_coordinates(image,right_fit_average)
  return np.array([left_line,right_line])
  def canny(image):
  gray=cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
  blur= cv2.GaussianBlur(gray,(5,5),0)
  canny=cv2.Canny(blur,50,150)
  return canny
  def display_lines(image,lines):
  line_image=np.zeros_like(image)
  if lines is not None:
  for x1,y1,x2,y2 in lines:
  cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)
  return line_image

def region_of_interest(image):
  height=image.shape[0]

polygons=np.array([[(200,height),(1100,height),(550,250)]]
)
  mask=np.zeros_like(image)
  cv2.fillPoly(mask,polygons,255)
  masked_image= cv2.bitwise_and(image,mask)
  return masked_image
#image =cv2.imread("test_image.jpg")
#lane_image=np.copy(image)
#canny_image=canny(lane_image)
#cropped_image=region_of_interest(canny_image)
#lines=cv2.HoughLinesP(cropped_image,2,np.pi/180,100,n
p.array([]),minLineLength=40,maxLineGap=5)
#averaged_lines=average_slope_intercept(lane_image,line
s)
#line_image=display_lines(lane_image,averaged_lines)
#combo_image=cv2.addWeighted(lane_image,0.8,line_im
age,1,1)
#cv2.imshow("result",combo_image)
#cv2.waitKey(0)
cap=cv2.VideoCapture("test2.mp4")
while(cap.isOpened()):
  _,frame=cap.read()
  canny_image=canny(frame)
  cropped_image=region_of_interest(canny_image)
 lines=cv2.HoughLinesP(cropped_image,2,np.pi/180,100,np
.array([]),minLineLength=40,maxLineGap=5)
  averaged_lines=average_slope_intercept(frame,lines)
  line_image=display_lines(frame,averaged_lines)

combo_image=cv2.addWeighted(frame,0.8,line_image,1,1)
  cv2.imshow("result",combo_image)
  if cv2.waitKey(1) == ord("q"):
    break
cap.release()
cv2.destroyAllWindows()
```

I. CONCLUSION

This paper intends to design an experimental Lane detection system by using computer vision-based technologies, which could efficiently detect the lanes on the road. It can be synchronized with different techniques like *preprocessing, thresholding, perspective transform*, etc that could be fused together in the proposed lane detection system by detecting the the lane line in form of binary images.

To summaries the programming the Sliding window search is used to recognize the left and right lane on the road. The cropping technique worked only the particular region that

consists of the lane lines.

Hence, from the experimental results, it can be concluded that the system detects the lanes efficiently with any conditions of the environment. The system can be applied to any road having well-marked lines and implemented to the embedded system for the assistance of Advanced Driver Assistance Systems and the visually impaired people for navigation to keep them in proper track. [1]

II. REFERENCES

[1] M. Rezwanul Haque, M. Milon Islam, K. Saeed Alam, and H. Iqbal, "A Computer Vision based Lane Detection Approach," *Int. J. Image, Graph. Signal Process.*, vol. 11, no. 3, pp. 27–34, 2019, doi: 10.5815/ijigsp.2019.03.04.

[2] Y. Ojha, "Self Driving Cars— a Beginners guide to Computer Vision — Finding Simple Lane Lines using Python and OpenCV | by Yogesh Ojha | Medium," 2019. https://medium.com/@yogeshojha/self-driving-cars-beginners-guide-to-computer-vision-finding-simple-lane-lines-using-python-a4977015e232 (accessed Nov. 08, 2021).

[3] D. Shrivastava, "SELF-DRIVING CARS USING OPEN CV," *Madras Scientific Research Foundation*. https://www.madrasresearch.org/post/self-driving-cars-using-open-cv (accessed Nov. 03, 2021).

[4] P. Joshi, "Hands-On Tutorial on Real Time Lane Detection using OpenCV," 2020. https://www.analyticsvidhya.com/blog/2020/05/tutorial-real-time-lane-detection-opencv/ (accessed Nov. 03, 2021).

[5] R. Uppala, "Advanced Lane Detection for Autonomous Vehicles using Computer Vision techniques | by Raj Uppala | Towards Data Science," 2017. https://towardsdatascience.com/advanced-lane-detection-for-autonomous-vehicles-using-computer-vision-techniques-f229e4245e41 (accessed Nov. 08, 2021).