Lane detection for a self-driving car using OpenCV

Dhruv Pandey · Follow
Published in Analytics Vidhya
7 min read · Apr 20, 2021

To all the people who are wondering how can this concept be covered inside an article, I would say things sound complicated until you have explored the depths of it. I will not say that the article would be very easy but yes it would be built on top of pretty basic computer vision concepts. Please do not presume that this is exactly what Tesla would be using in their cars, but it might be something similar.

What are the prerequisites? Some basic knowledge of OpenCV would be good. If not, don't worry I will try to explain the OpenCV functions that I will use and also put references for you to check them in more detail.

Each section of this article would cover a function that would finally be used in the main part of the program. Also, in this article, I will demonstrate everything using an image. You can reuse the same code to use on a video(since the video is just a collection of images).

The image that I will use for this article

Step 1: Edge Detection

We would be using Canny edge detection. If you are not sure what is this, check out my previous article that explains it in a practical way.

```
def canyEdgeDetector(image):
    edged = cv2.Canny(image, 50, 150)
    return edged
```

This is how the output looks after we applied canny edge detection

Output after Canny edge detection

Step 2: Define ROI(Region of Interest)

While driving, to keep a car on the lane you focus only on the next 100meteres of your current road. Also, you don't care about the road on another side of the divider. This is what is our *region of interest*. We hide the unnecessary details from the image and show only the region which would help us in finding the lane.

The Red triangle shows our Region of Interest

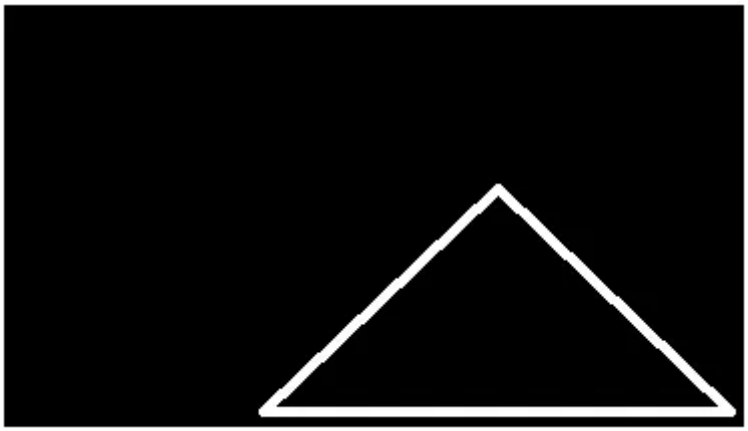
```
def getROI(image):
    height = image.shape[0]
    width = image.shape[1]
    # Defining Triangular ROI: The values will change as per your
camera mounts
    triangle = np.array([[(100, height), (width, height), (width-
500, int(height/1.9))]])
    # creating black image same as that of input image
    black_image = np.zeros_like(image)
    # Put the Triangular shape on top of our Black image to create a
mask
    mask = cv2.fillPoly(black_image, triangle, 255)
    # applying mask on original image
    masked_image = cv2.bitwise_and(image, mask)
    return masked_image
```

1. We have defined the triangular ROI, whose coordinates will change depending on where you have mounted the camera on your car(try to have only that part of the image which will actually contribute towards lane detection).

2. We have created a black image of the same shape as that of our original image:

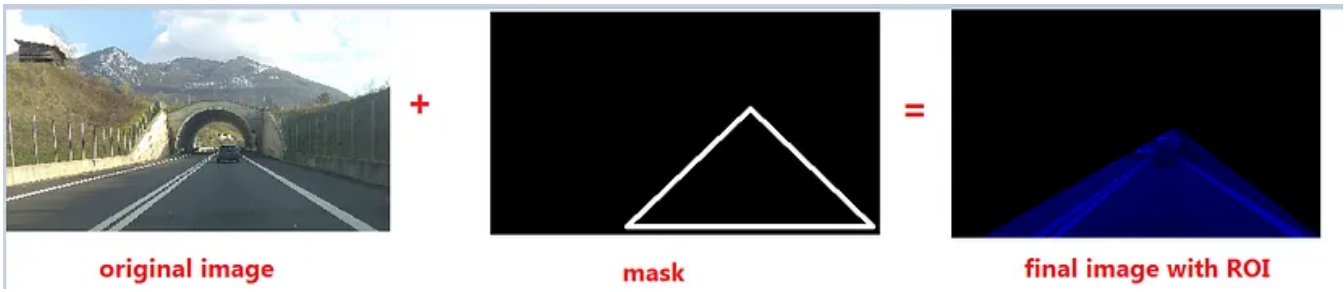
Creating a black image of the same shape as the original image

3. **Create a Mask:** We then use *cv2.fillPoly()* to put our triangle shape(with white-colored lines)on top of our black image to create a mask.



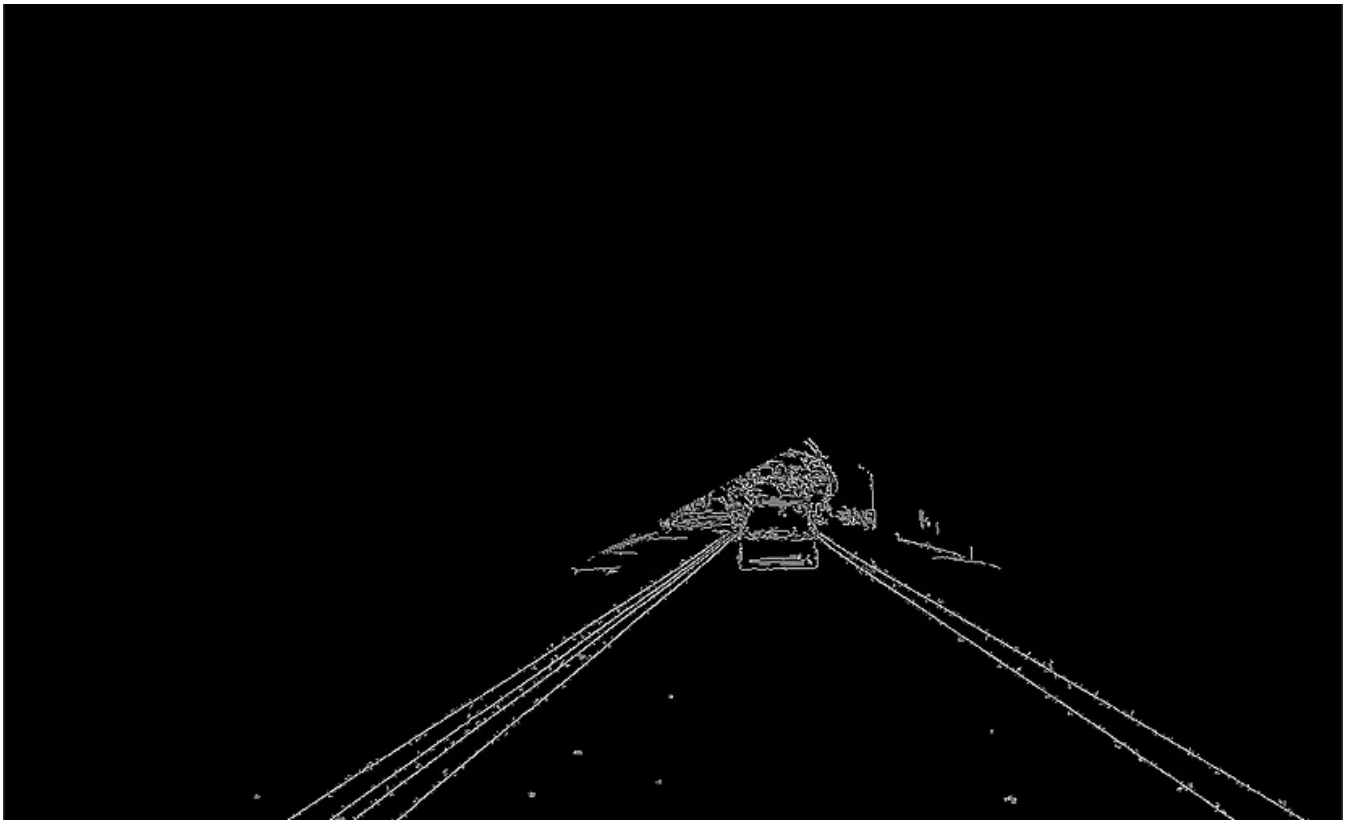
Create a mask

4. Apply the mask on our original image to get cropped image having only our ROI.



Original Image + Mask = Final Image with ROI

The output of this step would be something like:



Output after getROI()

> *It is important to apply edge detection before getting ROI otherwise edge detection will detect boundaries of our ROI too.*

Step 3: Get Lines

The next step would be to pass the ROI to get all the straight lines in the image. The *cv2.HoughLinesP()* helps you achieve this. This function returns a list of all the straight lines that it can find in the input image. Each line is represented by [x1, y1, x2, y2].

Now, this may look very simple but the underlying working principle of Hough Lines detection is a little bit time taking to explain. So I would not cover it in this article. Instead, I suggest you have a look at this tutorial (#28, #29, #30 should be sufficient to understand Hough Lines Principle).

```
def getLines(image):
    lines = cv2.HoughLinesP(image, 0.3, np.pi/180, 100,
np.array([]), minLineLength=70, maxLineGap=20)
    return lines
```

The parameters of the cv2.HoughLinesP() must be tuned as per your requirement(try to change and debug what works out best for you). But I think the ones above should work in most of the cases. This is how the output of this step looks like:

3 Lines detected in the image. There can be hundreds of lines that may be detected in your image. So tune your parameters to obtain as fewer lines as you can get

Step 4: Some utility functions

The following utility function takes an image and list of lines and draws the lines on the image. (This step is not taking any input from Step3. Instead, this is just a utility step that would be called from Step5, so you make first have a look at Step5 and visit this step when needed).

```
def displayLines(image, lines):
    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line.reshape(4) #converting to 1d array
            cv2.line(image, (x1, y1), (x2, y2), (255, 0, 0), 10)
    return image
```

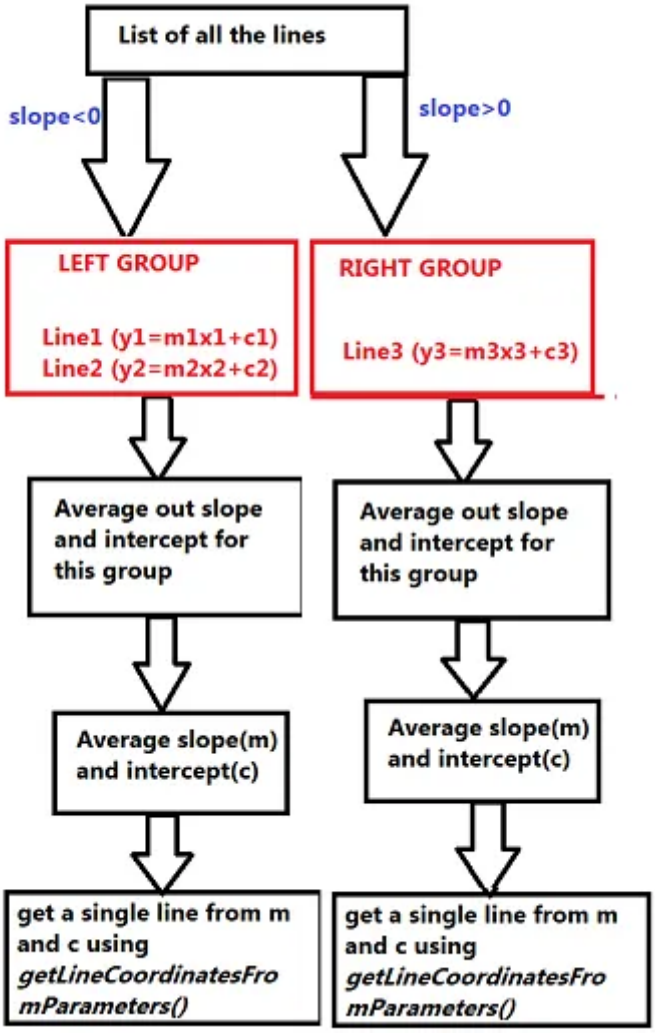
We define another utility function to get line coordinates from its parameters(slope and intercept). Keep in mind that a line is represented by $y=mx+c$ where m is the slope and c is the intercept.

```
def getLineCoordinatesFromParameters(image, line_parameters):
    slope = line_parameters[0]
    intercept = line_parameters[1]
    y1 = image.shape[0]  # since line will always start from bottom
of image
    y2 = int(y1 * (3.4 / 5))  # some random point at 3/5
    x1 = int((y1 - intercept) / slope)
    x2 = int((y2 - intercept) / slope)
    return np.array([x1, y1, x2, y2])
```

Notice how did we select values for y1 and y2

Step 5: Getting Smooth line

Once we have obtained lines from Step 3, in this step we are dividing those lines into 2 groups(left and right). If you notice the output image from Step3, this step will put Line1 and Line 2 into the *left group* and Line3 into the *right group*.



How to get a common line for the left and right side of the lane

Once grouped, we find the average slope(m) and intercept(c) for that group and try to create a line for each group by calling getLineCoordinatesFromParameters() and passing average m and average c.

Here is the function that does all of this:

```
def getSmoothLines(image, lines):
    left_fit = []  # will hold m,c parameters for left side lines
    right_fit = []  # will hold m,c parameters for right side lines

    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)

    # now we have got m,c parameters for left and right line, we
need to know x1,y1 x2,y2 parameters
    left_line = getLineCoordinatesFromParameters(image,
left_fit_average)
    right_line = getLineCoordinatesFromParameters(image,
right_fit_average)
    return np.array([left_line, right_line])
```

This is how the image looks after the lines have been grouped:

Output after lines grouping

The Main Code(calling above steps one by one)

Once we have the individual functions ready, we just have to call them in our main code and you would have lanes detected in your image.

```
image = cv2.imread("3.jpg") #Load Image

edged_image = canyEdgeDetector(image)   # Step 1
```

```
roi_image = getROI(edged_image)          # Step 2

lines = getLines(roi_image)              # Step 3

smooth_lines = getSmoothLines(image, lines)    # Step 5
image_with_smooth_lines = displayLines(image, smooth_lines) # Step 4

cv2.imshow("Output", image_with_smooth_lines)
cv2.waitKey(0)
```

The output will look something like this:

The final output with the determined lane

Final Words

You made it to the end of the article. Once things are sorted and works fine for the image, you know how to get it working on a video. You may have realized how you can use the very basic computer vision operations cleverly in order to achieve something this useful. Again I will say do not try to compare this work with what bigger companies like Tesla have done(the base for them is also something similar). Use this as motivation instead and maybe at some point, you are able to achieve something similar.

Till then keep exploring :) You can check the link to the Github source code for this article here.

Sample output after detecting lanes on a video(the frames are slow because of the video recorder):

Detecting lanes on a video

Opencv | Self Driving Cars | Lane Detection | Line Drawing | Hough Transform

Follow

Written by Dhruv Pandey

47 Followers · Writer for Analytics Vidhya

A machine learning and computer vision enthusiast working as a web developer in Finland.