Finding Driving Lane Line live with OpenCV

Percy Jaiswal Follow
Jul 14 · 8 min read

Hello, in this project I will attempt to find lane lines from a dash cam video feed. Once we detect lane lines, we will mark them on the original video frame and play it back. This all will be done online and without any lag using OpenCV functions.

Our approach here would be to develop sequence of functions to detect lane lines. We will use a 'sample' image while writing this functions, and once we are able to detect lane lines on few 'sample' images successfully, we will club complete program into a function which can accept live feed image and return the same image frame with lane lines highlighted. So without much delay, let's get started.

Sample Image

First we input our sample image frame. This line will be commented in final code, where 'image' will be frame sent by video capture.

```
image = cv2.imread('test_images/whiteCarLaneSwitch.jpg')
```

Greyed Image

To lessen the burden on our processor (which is very scarce resource in embedded systems), we will do all image processing in 'Greyscale' version of the image instead of original colored version. This helps execute our program faster with less resources. Below function converts color image to its greyscale version

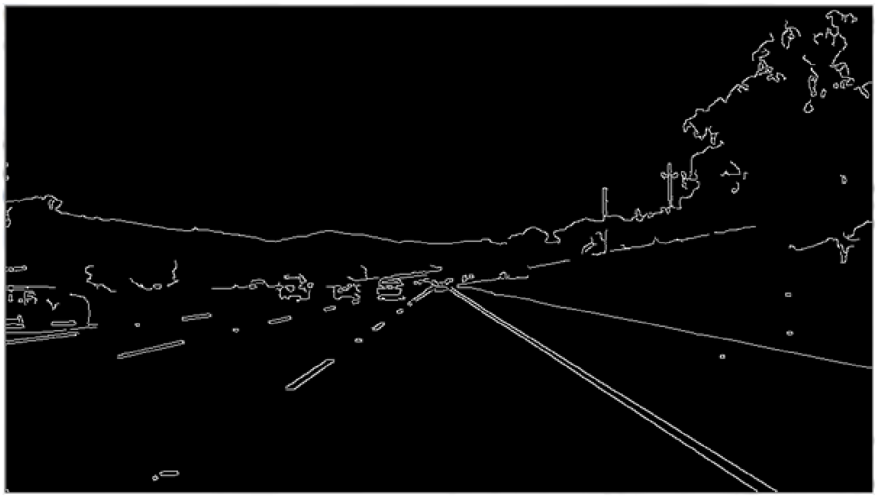
```
grey_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Image Blurring (Image Smoothing)

Next we will remove noise from our image by blurring it. Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noises. It actually removes high frequency content (eg: noise, edges) from the image. So edges are blurred a little bit in this operation. OpenCV provides 4 different types of blurring techniques, with Gaussian blur being the most popular one.

We can select different kernel sizes, wherein the resultant filter will simply take the average of all the pixels under kernel (a row x column matrix of kernel size) area and replace the central element with average value. Again 5 is a fairly standard value and has worked for me.

```
kernel_size = 5

blur_gray = cv2.GaussianBlur(grey_image,(kernel_size,
kernel_size),0)
```

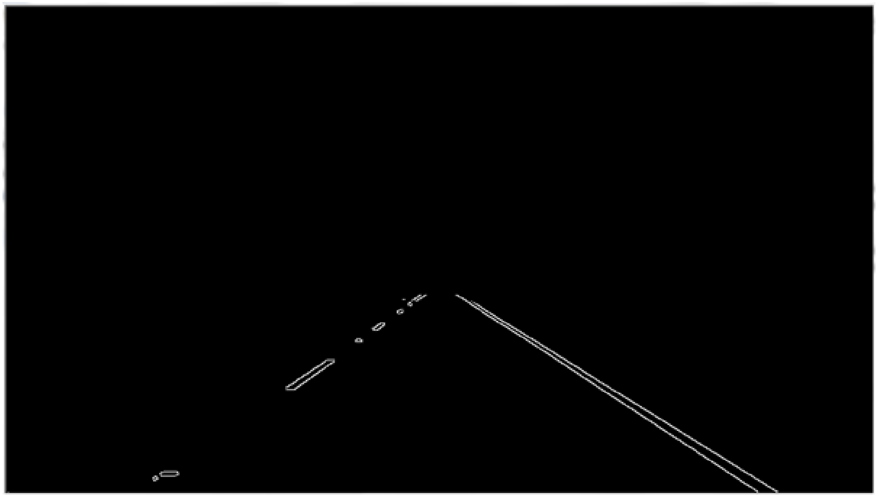

Canny Edge Detection

Canny Edge Detection is a popular edge detection algorithm. In fact Canny edge function also implements a 5x5 kernel Gaussian filter that we had used in previous steps, but in many of the literature that I had come across, it's always recommended to implement your own blurring before canny edge detection. The basic theory behind edge detection is, wherever there is an edge, the pixel on either side of the edge have a big difference (also called gradient) between their intensities. First the input image is scanned in both horizontal and vertical direction to find gradient for each pixel. After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighborhood.

low_threshold and high_threshold determine how strong the edge must be to be detected. A gradient is considered to be part of an edge, if its gradient is higher than 'high_threshold'. But once an edge is detected, the next pixel is included in the edge even if it's greater than just 'low_threshold'.

```
low_threshold = 50

high_threshold = 150

edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
```

With reference to our sample image, it's clear that at edge conditions, especially where lane lines are, there is vast contrast difference between adjacent pixels, with lane line being white and adjacent road pixel being dark.


Region of Interest

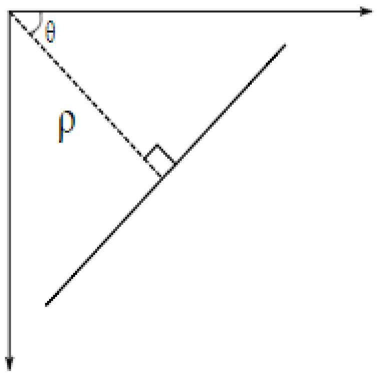
One thing to consider is, we don't want to find all the edges in the image. We are just interested in finding the lane around the center area of our image. Intuitively it makes sense as in an edge in top right / left portion of the image is highly unlikely to be a driving lane. Looking at out sample image, we can safely say that lane lines should be inside a trapezoidal area with broader edge at bottom of image and with edges becoming narrower as we go towards top portion of the image.

Following four lines mark the region of interest in our edge detected image. First, we find out size of the image, second we create four corners of trapezoid (this step, as many others is bit of an iterative process, wherein we need to try with different values to find out best case). Third, we create the trapezoid with above vertices and finally we do a bitwise operation, so that only pixel which are inside region of interest and are classified as edge are marked as 1.

```
imshape = image.shape

vertices = np.array([[(0,imshape[0]),(450, 320), (500, 320),
(imshape[1],imshape[0])]], dtype=np.int32)
```

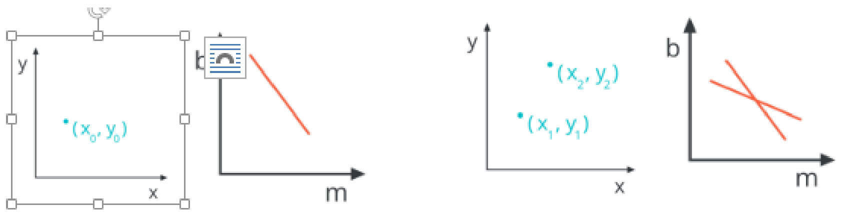
```
cv2.fillPoly(mask, vertices, ignore_mask_color)

masked_edges = cv2.bitwise_and(edges, mask)
```

Next we perform Hough Line Transform in order to detect a line from the above edge detected image. Remember an edge could be a circular edge too, but the edge we are interested in our application is line edge of a driving lane.

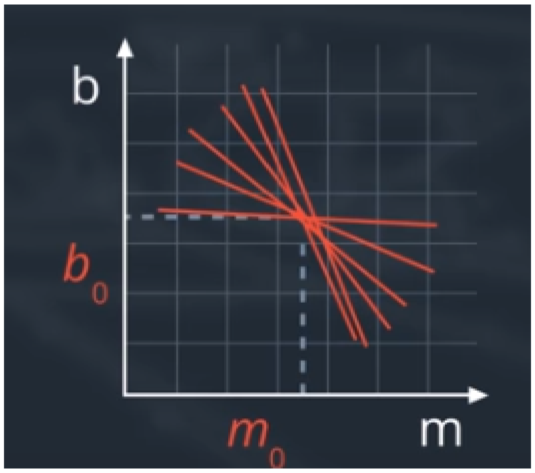


Hough Line Transform

In short, Hough's transform method transforms a line from its traditional y = mx + b form to rho = x *cos (theta) + y * sin (theta) where rho is the perpendicular distance from origin to the line, and theta is the angle formed by this perpendicular line and horizontal axis.

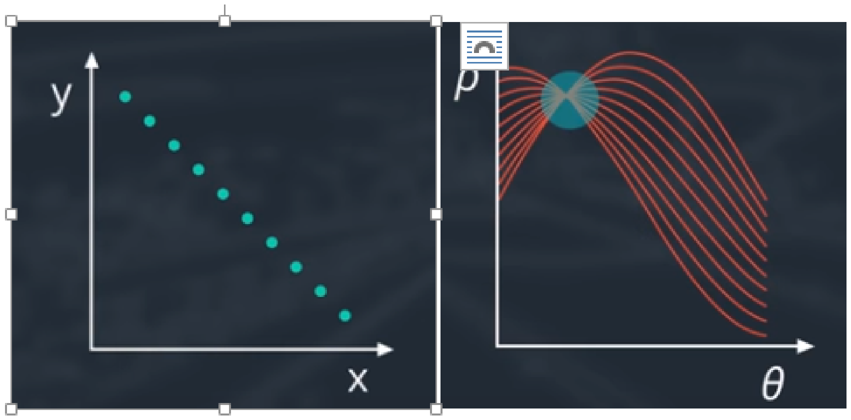


We know that a line (y = mx + b) when represented in m vs b graph is just a point, and a point in x, y frame is represented as a line in m vs b frame.



So our strategy to find lines in image space will be to find intersecting lines in Hough space. We do this by dividing up our Hough space into a grid and define intersecting lines as all lines passing through a given grid cell. And where many lines in Hough space intersect, we declare we have found a collection of points that describe a line in image space.

We have a problem though, vertical lines have infinite slope in m vs b representation and that's where need for rho vs theta parameterization comes into picture.



Now each point in image space corresponds to a sine curve in Hough space (rho vs theta). And if we take a whole line of points, it translated into whole bunch of sine curves in Hough space. You can consider a sine curve in Hough space as equivalent to a line in m vs b space which is representation of a point in image space. And again, the intersection of those sine curves in Hough space gives the representation of the line.

Coming back to our code, we first define parameters for Hough Transform, and then call the function itself.

```
rho = 2 # distance resolution in pixels of the Hough grid

theta = np.pi/180 # angular resolution in radians of the
Hough grid

threshold = 15     # minimum number of votes (intersections
in Hough grid cell)

min_line_length = 40 #minimum number of pixels making up a
line

max_line_gap = 30    # maximum gap in pixels between
connectable line segments

line_image = np.copy(image)*0 # creating a blank to draw
lines on
```

Output "lines" is an array containing endpoints of detected line segments.

```
lines = cv2.HoughLinesP(masked_edges, rho, theta, threshold,
np.array([]),min_line_length, max_line_gap)
```

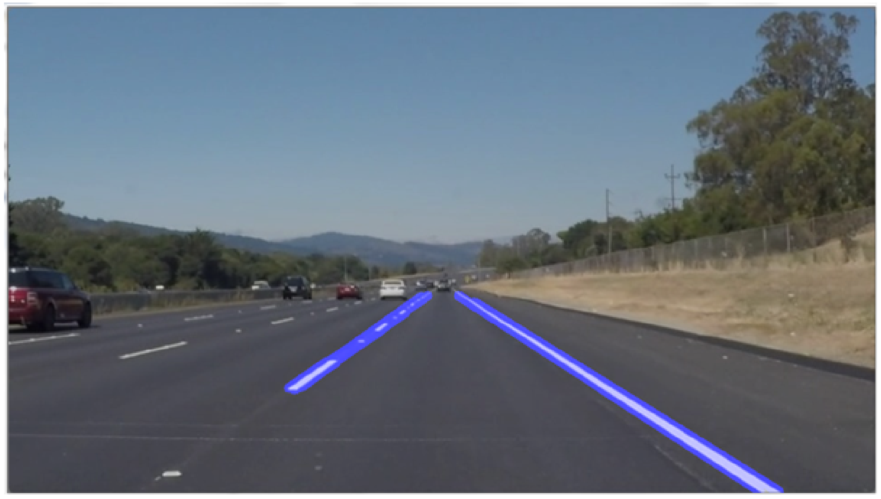
Iterate over the output "lines" and draw lines on a blank image.

```
for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(line_image,(x1,y1),(x2,y2),
(255,0,0),10)
```

Draw the lines on the original image and return it.

```
lines_edges = cv2.addWeighted(image, 0.8, line_image, 1, 0)
```

```
  return lines_edges
```


Lane Lines on a single frame

After successfully detecting lane lines on an sample image, we will take a video as an input, detect lane lines and play back the video. Note that here I am processing a previously recorded video, but this can very easily be applied to a live video using same cv2.VideoCapture function.

First we create an object of our video using cv2.VideoCapture command. While 'video_capture' is running, we read it. Read() function will return two variable, where first variable is a Boolean value indicating success or failure of read operation with a true or false value respectively and second object is the captured frame itself. So whenever 'ret' is true, we take the frame and simply pass it to processImage function which we just built above. The output received from processImage is displayed showing lane marking on top of captured frame.

```
video_capture =
cv2.VideoCapture('test_videos/solidWhiteRight.mp4')


while (video_capture.isOpened()):
    ret, frame = video_capture.read()
    if ret:
        output = processImage(frame)
        cv2.imshow('frame',output)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break
```

```
# Release everything if job is finished
video_capture.release()
cv2.destroyAllWindows()
```

And that's about it. Hope this article was written in good enough manner for you to have a pleasant read and hopefully learn a thing or two. Obviously there are many more improvements which can be implement in above program, for e.g. checking slope of detected line to check whether detect line is in fact consistent with lane line and remove outliers. Etc. Feel free to suggest any improvements and suggest which can help me grow too. Complete code with sample image and video can be found here

If you liked this post, Follow, Like, Retweet it on Twitter or Claps, Likes here on Medium will act as encouragement for writing new posts as I continue my journey into Blogging world.

Till next time….cheers!!