Finding Lane Lines - Self Driving

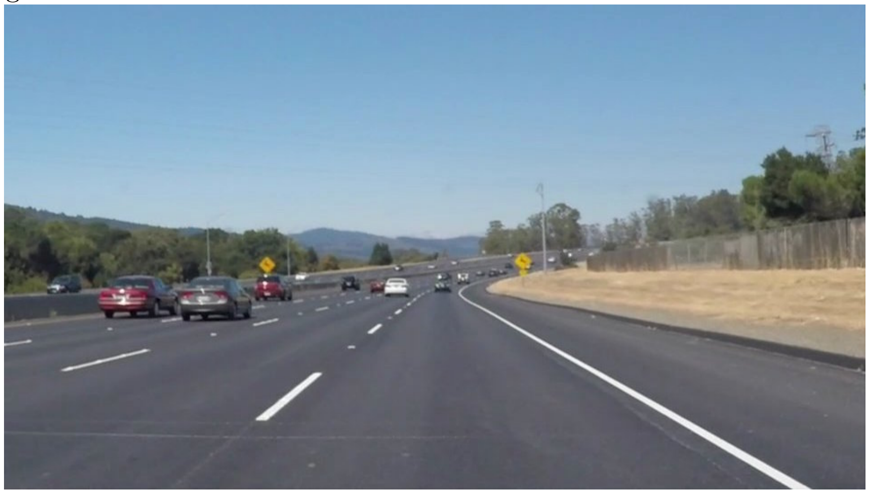
Tianyun Shan

<2017-01-29 Sun>

Contents

1 Detect Lane Lines On Still Image

The first step is detecting lane lines on a still image. Here is an example image that we use to detect the lane lines.



1.1 Canny Edge Detection

First read in an image and convert to grayscale.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2  #bringing in OpenCV libraries
image = mpimg.imread('exit-ramp.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY) #grayscale conversion
# print out image
plt.imshow(image) #original
plt.imshow(gray, cmap='gray') #grayscale image
```

Now let's try the Canny edge detector. We are applying Canny to the image. The algorithm first detect strong edge (strong gradient) pixels above `high_threshold` and reject pixels below `low_threshold`. the ratio of `low_threshold` to `high_threshold` is recommended to be 1:2 or 1:3.

The course recommend we include Gaussian smoothing before running `Canny`. Gaussian smoothing is essentially a way of suppressing noise and spurious gradients by averaging (Here - OpenCV Doc). The `kernel_size` for Gaussian smoothing to be any **odd number**. A larger `kernel_size` implies averaging or smoothing over a larger area.

```
# Do all the relevant imports
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2

# Read in the image and convert to grayscale
# Note: in the previous example we were reading a .jpg
# Here we read a .png and convert to 0,255 bytescale
image = mpimg.imread('exit-ramp.jpg')
gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

# Define a kernel size for Gaussian smoothing / blurring
kernel_size = 5 # Must be an odd number (3, 5, 7...)
blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)

# Define our parameters for Canny and run it
low_threshold = 50
```

```
high_threshold = 150
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)

# Display the image
plt.imshow(edges, cmap='Greys_r')
```

More details, this Introduction to Computer Vision course on `udacity` helps.

1.2 Hough Transform

At this point, we have the image applied Canny edge detection. In order to detect lines, we use Hough Transform on top of the Canny image. To do this, we will use an OpenCV function called `HoughLinesP` that takes several parameters.

If you want to know how Hough Transform is implemented in the first place, take a look at this blog.

Here is the complete source.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2


# Read in and grayscale the image
# Note: in the previous example we were reading a .jpg
# Here we read a .png and convert to 0,255 bytescale
image = mpimg.imread('exit-ramp.jpg')
gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

# Define a kernel size and apply Gaussian smoothing
kernel_size = 5
blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)

# Define our parameters for Canny and apply
low_threshold = 50
high_threshold = 150
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
```

```
# Next we'll create a masked edges image using cv2.fillPoly()
mask = np.zeros_like(edges)
ignore_mask_color = 255

# This time we are defining a four sided polygon to mask
imshape = image.shape
vertices = np.array([[(0,imshape[0]),(450, 290), (490, 290), (imshape[1],imshape[0])]],
# vertices = np.array([[(0,imshape[0]),(0, 0), (imshape[1], 0), (imshape[1],imshape[0])
cv2.fillPoly(mask, vertices, ignore_mask_color)
masked_edges = cv2.bitwise_and(edges, mask)

# Define the Hough transform parameters
# Make a blank the same size as our image to draw on
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 = 20    # maximum gap in pixels between connectable line segments
line_image = np.copy(image)*0 # creating a blank to draw lines on

# Run Hough on edge detected image
# 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)

# Create a "color" binary image to combine with line image
color_edges = np.dstack((edges, edges, edges))

# Draw the lines on the edge image
lines_edges = cv2.addWeighted(color_edges, 0.8, line_image, 1, 0)
plt.imshow(lines_edges)
```

For more details on `HoughLinesP` API:

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

1. `edges` - the output image from `Canny`

2. `rho` and `theta` - distance and angular resolution of our grid in Hough space. Remember that in Hough, we have a grid laid out along the ($theta, $rho) axis

3. `threshold` specifies minimum number of votes (intersections in a given grid cell).

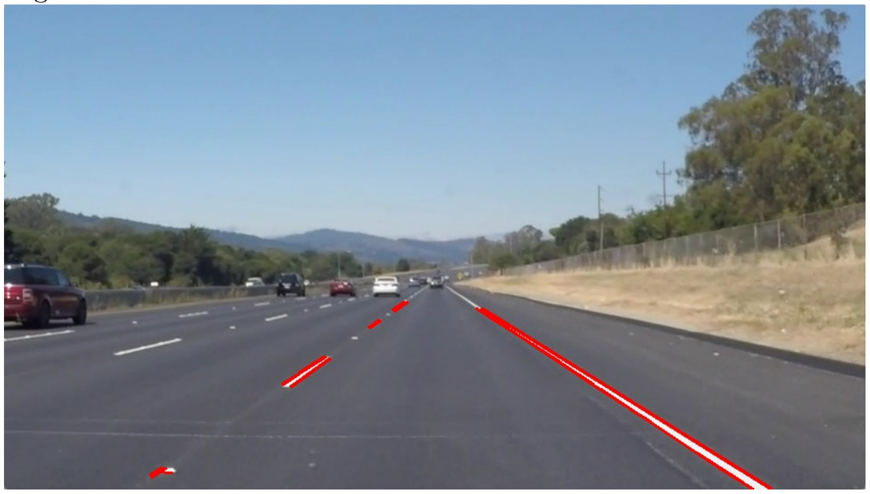
4. `np.array([])` is just a placeholder, no need to change.

5. `min_line_length` is minimum length of a line that you will accept in the output

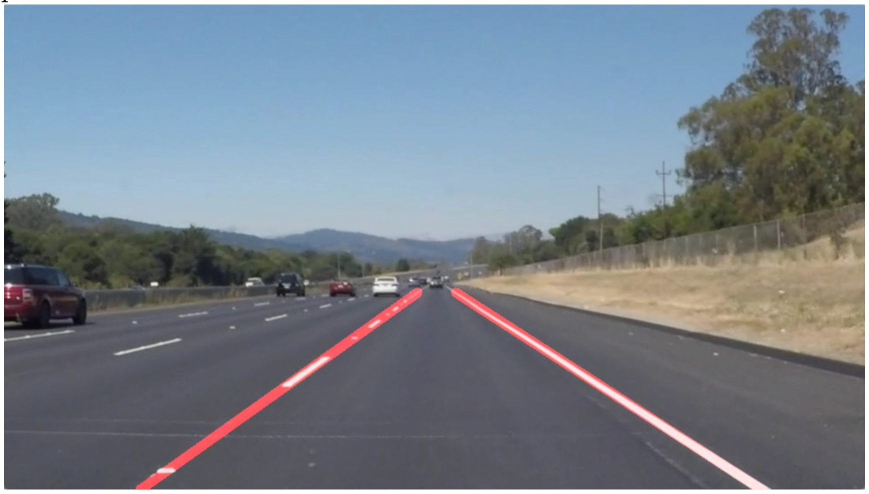
6. `max_line_gap` is maximum distance between segments that you will allow to be connected into a single line

2 Detect Lane Lines On Video (project)

Process on video is similar to process on still images. What we do is to consider write a pipeline process on still images and treat the video as a list of images and apply the pipe line on it. We already have learned how to detect lane lines on a still image. Here is the difficult part to what we learned. Previously we use `hough transform` to detect lines, now we need to only draw on solid line for left and right lane. that solid line should be connect to the bottom edge so we can detect where the lane starts while driving.



The output should look something like above after detecting line segments, and the goal is to connect/average/extrapolate line segments to get output like below.



2.1 Pipeline on Still Image

For detail on how to detect lines, please see `Canny Edge Detection` and `Hough Transform`. Previously, we print lines on map using the following method.

```
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(img, (x1, y1), (x2, y2), color, thickness)
```

That method draws all the lines we can find, but now we want only two lines. One on the left and one on the right. How am I going to tweak this method and make it draw two lines left and right?

Here is how I did. It must not be the best plan but it is the one I use in the project. First, we can see that the slope for left line is negative and the slope for the right line is positive. (When x increase and y increase, the slope is positive).

So I loop all the lines and find one with positive slope and one with negative slope.

Now here are two ways I can do. One is to just loop though all the lines and find all the slopes, make an array of left slops and right slops put the positive and negative number into the correct array, and calculate left and right slope average. But this is not what I did. Why?

When I calculate the slope, I found out that some line I detected does not belongs to left or right, so when I get one sample of left and right slope I calculate if the slope is 0.1 difference to the left or right slope. If this close to neither left nor right, I ignore this line. The rest the similar, I put slope of the line I want in to left array and right array, calculate the average.

Note: when I was looping though the lines and calculate the slope, I also need to calculate the y-intersection points. See formula below to get all the y-intersections for the line and also calculate the average.

m $= (\mathrm{y}_2 - y_1)/(x_2 - x_1)b = y - m * x$

From this point, I have both slope and y-interactions for left line and right line. There is one more thing we need to do. The line we draw must starts at the bottom edge of the image. However, the min points we get may not 100% starting from the edge, so we need to calculate the point value ourselves. How? $y = mx + b$, we have `m` and we have `b`, the y is the image height, so we can get x.

The other point will the point with the minY value. filter all the points for both left line and right line and find the minimum y-value point.

Below is all the codes I use to draw the line. Since I do not have too much time on this project, the code is a bit mass here. Its only for my own reference.

```
leftM = 0
rightM = 0
findSample = 0 # find flag
slopeDifference = 0.2
for line in lines:
    for x1, y1, x2, y2 in line:
        for newline in lines:
            for a1, b1, a2, b2 in newline:
                if x1 == a1 and y1 == b1 and x2 == a2 and y2 == b2:
                    # all the same, ignore
                    continue
                m1 = ((y2-y1)/(x2-x1))
                m2 = ((b2-b1)/(a2-a1))
                if m1 > 0 and m2 < 0:
                    leftM = m1
                    rightM = m2
                    #print("leftM = %f, rightM = %f" %(leftM, rightM))
                    findSample = 1
                if m1 < 0 and m2 > 0:
                    leftM = m2
                    rightM = m1
                    #print("leftM = %f, rightM = %f" %(leftM, rightM))
                    findSample = 1
                else:
                    continue
                if findSample == 1: break
            if findSample == 1: break
        if findSample == 1: break
    if findSample == 1: break

leftLinePoints = []
rightLinePoints = []
for line in lines:
    for x1, y1, x2, y2 in line:
        m = ((y2-y1)/(x2-x1))

        if abs(m - rightM) <= slopeDifference:
            #right line points
            rightLinePoints.append((x1, y1))
            rightLinePoints.append((x2, y2))
        elif abs(m - leftM) <= slopeDifference:
            #left line points
            leftLinePoints.append((x1, y1))
            leftLinePoints.append((x2, y2))
        else:
            continue

# find smallest points in left
# get array of y points for leftPoints
leftYPoints = list(map(lambda x: x[1], leftLinePoints))
minLeftYPoints = min(leftYPoints)
maxLeftYPoints = max(leftYPoints)

minLeftPoints = list(filter(lambda x: x[1] == minLeftYPoints, leftLinePoints))[0]
maxLeftPoints = list(filter(lambda x: x[1] == maxLeftYPoints, leftLinePoints))[0]

# get array of y points for rightPoints
rightYPoints = list(map(lambda x: x[1], rightLinePoints))
minRightYPoints = min(rightYPoints)
```

```
maxRightYPoints = max(rightYPoints)

minRightPoints = list(filter(lambda x: x[1] == minRightYPoints, rightLinePoints))[0]
maxRightPoints = list(filter(lambda x: x[1] == maxRightYPoints, rightLinePoints))[0]

# calculate final slope
final_left_m = (maxLeftPoints[1] - minLeftPoints[1]) / (maxLeftPoints[0] - minLeftPoint
final_left_b = (minLeftPoints[1] - (final_left_m * minLeftPoints[0]))
final_right_m = (maxRightPoints[1] - minRightPoints[1]) / (maxRightPoints[0] - minRight
final_right_b = (minRightPoints[1] - (final_right_m * minRightPoints[0]))

height, width, channels = image.shape
final_left_point = (height - final_left_b) / final_left_m
final_right_point = (height - final_right_b) / final_right_m
cv2.line(line_image, minLeftPoints, (int(final_left_point), height), line_color, 10)
cv2.line(line_image, minRightPoints, (int(final_right_point), height), line_color, 10)
```

2.2 Use Pipeline On Video

To use it on a video, we have all the sample codes provided. All we do is to wrap our pipe line into a `def process_image(image)` function and to apply this function to every frame of the video.

```
def process_image(image):
    # put the pipe line code here.
    # image is the input image
    # result is the output image
    return result

white_output = 'xxx-output.mp4' #video output name
clip1 = VideoFileClip('xxx-input.mp4') #video output name
white_clip = clip1.fl_image(process_image)
%time white_clip.write_videofile(white_output, audio=False) #write to the output video
```

3 Improvement

How to make the algorithm more robust? Currently the algorithm only detect straight line because I am using the linear equation. So in situation when the road is not straight, this algorithm may fail. In order to make it better, instead calculating and drawing the straight line, we can draw the curve. I think drawing curve is not that as easy as drawing the line so another work round may be calculating multiple slope and drawing many lines to form a curve.

Above are just my thoughts on how to make improvements.