FINDING LANE LINES ON THE ROAD

Author: Jumana Mundichipparakkal

Reflection

1- PIPELINE DESCRIPTION

I created a lane processing pipeline function which goes through the following steps sequentially.

1- **Convert the image into grayscale** for making the detection of potential white lane points easier, using Gaussian filtering and Canny edge detection to follow, making pixels one dimensional from three dimensional colored values and we aim at detecting white points on black background.

2- **Apply Gaussian filter** on the gray image to smoothen the image filtering noise. For smoothening, the filter requires a kernel size parameter which I have chosen to be 3. The larger the size, blurs the image a lot. In our case of images with white lanes, 3 proved to be an optimum value.

3- **Apply Canny Edge Detection** algorithm on the smoothened image to detect the edges in the image. This will help to identify all the white points in the black image. Threshold values chosen are 50 for low and 150 for high, considering that Canny suggests a ratio of 1:3 for threshold values. These values help to identify the transition pixels from black pixel(pixel value: 0) to white pixel(pixel value: 255) and vice versa decently.

4- **Identify edge pixels in our region of interest.** Looking at the lane images, from the angle of camera, white lanes are always placed within a specific region of the entire image. This helps to filter out all other edges detected in the entire image. Working with all input images and videos, I figured out that a polygon of four vertices with following values work decently enough.

   x and y are image sizes and following are the coordinates of the polygon.
   top_left = (int(x*25/100), int(y*65/100))
   bottom_left = (int(x*10/100), int(y*90/100))
   top_right = (int(x*75/100), int(y*65/100))
   bottom_right = (int(x*85/100), int(y*90/100))

   Currently the *percentage values of the x size and y size* for deciding the polygon vertices are determined by manual experiments. I think there should be a way to determine these from the camera positioning.

5- **Use Hough transform to identify potential line points** in the image. Hough transform helps to identify all lines in the image space as points. It converts image into polar coordinates, which require rho and theta parameters. I chose rho as 2 pixels and theta as 1 radian. On the top, to decide potential lanes, we have other parameters to tweak.

threshold parameter choses how many point of intersections in Hough space are minimum needed to consider as line of interest. This will help to eliminate one line identified here and there as noise.
min_line_length determines minimum number of pixels that can form a line of interest. This helps to eliminate small lines formed from noise as compare to the real lines of interest in the lane we are looking for.
maximum_line_gap is the allowed gap between two pixels to form a single line. This helps to join dotted white lines, but remove the dots that actually are noise and have wider gaps than expected.

Chosen values are:
threshold = 20
min_line_length = 40
max_line_gap = 20

6- **Draw the Hough detected line segments on top of the image** which should ideally have detected the white lines on the lane. For this, we need to determine the correct line segment of our interest from all the returned line segment points from the above step. For this, strategies used are:
   a. Slope values of the lines can be limited to a range for both lanes. Slope is positive for right lane and negative for left lane. From statistical analysis, I have found that value lie between 0.5 and 0.8 for right lane and -0.5 and -0.8 for left lane. I eliminate the points that don't have slope in this range after computing slope. Passed line points are added to a list and accumulated.
   b. From the accumulated line points, I use *np.polyfit* to find a line segment that fits all these points. This return the slope and coefficient of the most fitting line segment.
   c. We can find the x coordinates of both the left lane and right lane by feeding the y values to the line equation, *y = mx +b.*
   Bottom y value is the image y size and top y value is the 65% of the y size.

2- POTENTIAL SHORTCOMINGS IN PIPELINE

1- **Gaussian filter:** The kernel size is chosen based on the test images, it could be possible that this value can smoothen out a lot in less intensive image (less contrast).
2- **Canny Edge Detection:** The threshold values chosen for edge detection are optimized for the test inputs. It might not work well in less intensive images like in the middle point of the challenge video. It can also be a problem in different colored lanes or in images with lot of light where contrast is very less.
3- **Region of Interest:** I have chosen the vertices approximating to the test input images. This should be done based on the position of the camera.
4- **Hough Transform Line detection:** The poly fit algorithm is good at detecting linear lanes and it would work poorly for curved lanes.
5- **Drawing lines using slope identification:** I have statistically found the slope value ranges now. I think there should be a better way to learn the slopes either from previous slopes of lines detected and position of camera.

3- POSSIBLE IMPROVEMENTS IN PIPELINE

From the shortcomings stated in the previous section, possible improvements in the pipeline could be:

1. Finding better algorithms that work in images taken in any intensity of light, which will help to compute the canny edge detector threshold values.
2. Get the camera position parameters integrated to compute the vertices of the polygon that covers the area of lanes in the car.
3. Currently, we perform lane detection on grey scale image. Introducing colour detection should help in filtering lines more robustly. Similarly, some other feature detection algorithm integrated with line or curve detector should help.