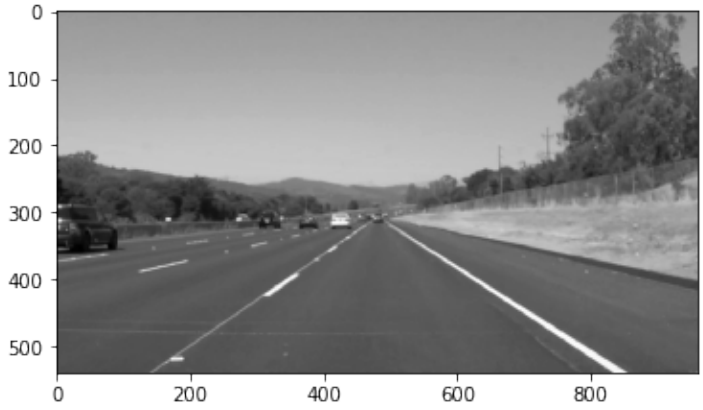
CarND-Lane-Line-Detection

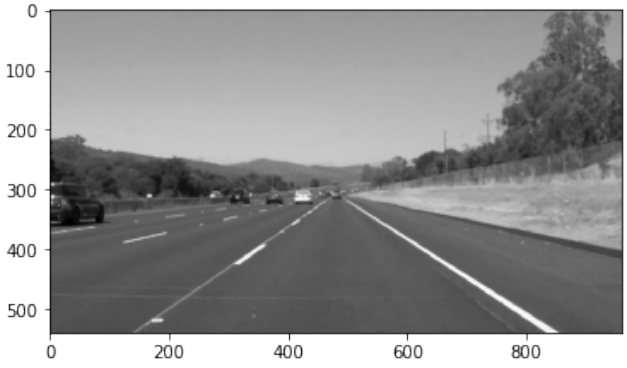
1.Pipeline Description

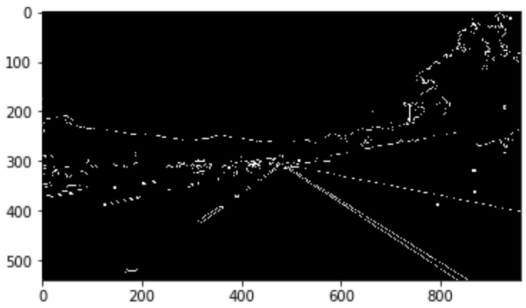
The Pipeline consists of 6 steps.



First the incoming image is grayscaled.

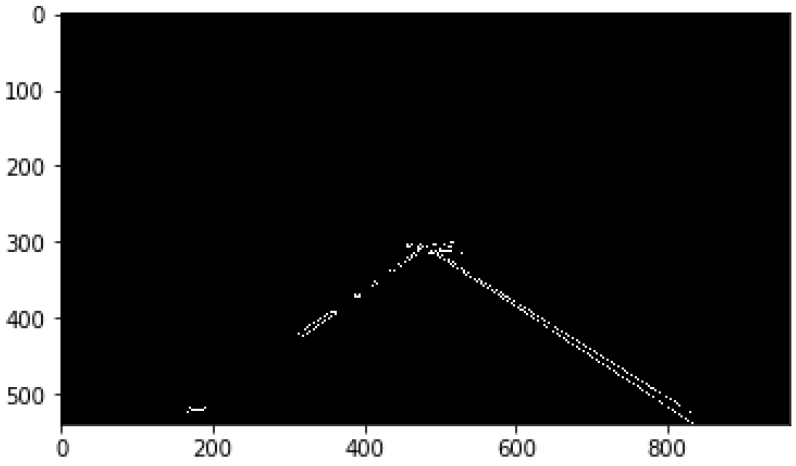
As the second step the grayscaled image was smoothed with the gaussian blur algorithm with a kernel_size of 5. This is not a must do, but it helps smoothing the image and makes it easier to apply the canny edge detection later to that image.





Then we apply the canny edge detection on the grayscaled and smoothened image to be able to detect our lines in the image.

Followed by the creation a masked image, turning the „uninteresting“ part of the image to black.



This shall reduce the needed computation power and later prevent detecting lines which are not important for our car to stay on its side of the road. Since for this example we are testing this with a steady camera, this is easily achieved by a fixed size for the region of interest.

On the next part of the pipeline we need to detect the lines of the previous image. For this I have chosen the following parameters:

rho=2, theta=np.pi / (180), threshold=40, min_line_len=30, max_line_gap=10

Rho and theta were just copied from the udacity exercises. As threshold I have chosen a value of 40, min_line_len 30px and max_line_gap 10px, this did result in the best line detection for me. Those values were adjusted with various runs and are more a result of trial and error. The final and most crucial and hardest part was the extension of the draw_lines function inside of hough_lines.

I needed far more time for this project than I would like to admit. The sad thing about this is, that the problem occured because I have overwritten the original input video with drawn lines on it without noticing… The math behind this project, was developed by the help of my math teacher at the university and by myself. My first attempt started by creating a small function which should help me reduce the error rate by calculating the slope, was to define a get_slope function. This would take 4 parameters (x,y, x1,y1) representing each pair of (x,y) values a single point. Then I simply used the formula given by udacity in the comments to calculate the slope for the given points. Using this function prevented me from writing one big line to calculate the slope each time I needed it. Following the next step was the hard part of this project, modifying the draw lines function to connect from the bottom of the line to the top of the detected line.

First my problem was understanding, how to actually achieve this, I already had the idea to average the lines and plot them in some way. Averaging was actually a right guess, but it was not everything which had to be done. As I already mentioned the math behind this was created with help of the math teacher at my university, so forgive me if this explanation of how I achieved it is not the actual math explanation.

The first part of this task, was knowing which values from the lines array, where actually from the left or from the right side of the street. This was fairly easy, as one could imagine a „coordinates system“ (probably not the correct English term for this) where the slope of the x and y values is the actual line. By plotting that line on the „coordinates system“ and dividing it through the y-axis, if the slope plots from the negative side, it is obviously the left lane, as the slope with a positive value is plotted on the right. As already mentioned, I do not know if this is really correct in math, but for me it made sense and it worked. So by first calculating the slope, I determined if the x and y values where from the left or from the right side of the lane. Since I wanted to calculate the mean of my values I created for each x1,x2 and y1,y2 value a list. The values which where determined to be part of the left lane got appended to the left x1 and left x2 and so on. Same goes for the values which where supposed to be on the right side of the lane. Those were appended to the right x1,x2, … values.

Now follows the second part of the task. The lists I had now, where used to calculate the mean of each value. So I had the average of x1,x2,y1 and y2. Not having thought my idea till the end, I did not notice that I had to use int values (images use int values), after a short try of drawing those values on the image, I noticed and could correct that by casting all the mean values to numpy.int.t

Having the averages I was able to calculate an approximation to the slope of the each of the lines (left and right), thanks to the previously created function this was done by calling get_slope and passing in all averaged values for the right line. The output was the slope for the right lane. But the average values itself, where not the values who could be used to draw the lines.

This is where the first problem of this projects solution occurs. To draw the line of the road, I used fixed values not only in the vertices for the region of interest, but aswell in the Y-values. For the right side of the lane, y1 and y2 are fixed values (320, 540). 540 being the beginning of the lane and 320 being the furthest point of the right lane side. Changing the position of the camera or the image shape, would destroy the existing pipeline which of course is bad. But since it does get the job done for this project, and people from computer science are supposed to be lazy, why not?

Now comes the tricky part, this is where my math prof. did help me (thanks to that prof. Todorov) we already have 2 points of the 4 we do need. Both are y-values.
So we need to determine the first x-value (x1), we need to plug it into our formula.
Since we already got the averaged values and the fixed values for y1 and y2. We simply need to fill in the corresponding values.

The formula looks like this:
x1 = right_average_x1_value + (right_y1 - the right_average_y1) / right_calculated_slope

The result was needed to be casted again to a numpy.int.

Same goes for the X2 value, the only difference is that now the right_y1 value is changed with the right_y2 value.
Now we have our 2 Points calculated and only need to draw them with the cv2 function onto the image. This step is repeated and the values are swapped for the values from the left line. There was still one Problem, sometimes there wrong lanes detected. Some detected Points in the image where not part of the lane („line“), so I started testing how to define a correct line. While debugging I noticed that the fastest way to determine an atleast possible lane. While trying out many values I got to the average value of -0.8 and -0.8. Our average lane is around 0.7 and -0.7

As last step, the predefined function „weighted_img“. Here we take the original image and draw our lines from the hough_transform + draw_lines on it.

Possible Shortcomings

As already mentioned, this project was solved using fixed y-values. Having another camera angle or simply another lane with a slight curve would destroy the drawing of the lines. Even worse curved roads would absolutely mess up this pipeline. So as introduction this is fine, and it works with this kind of sample pictures/videos. Not only the line detection will be a problem for curved lines, the region of interest or masked image. This one is also, created with fixed sizes which only work for the given image size. Another image shape, or again tilted / changed camera all this would affect this pipeline incredibly hard, making it useless for pictures and videos not being a close enough copy to the given data.

Possible Improvements

Transforming the images from the road into the third person giving a clearer image of the 2 lines in which the car needs to be. Then declaring the zone inside of both lanes. Then we could compute the steering angle for the car to hold. With cv2 it could also be possible to determine the position of the car in its environment.