 # **Finding Lane Lines on the Road**

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file. But feel free to use some other method and submit a pdf if you prefer.

****Finding Lane Lines on the Road****

The goals / steps of this project are the following:

*** Make a pipeline that finds lane lines on the road**

*** Reflect on your work in a written report**

[//]: # (Image References)

[image1]: ./examples/grayscale.jpg "Grayscale"

Reflection

1. Describe your pipeline. As part of the description, explain how you modified the draw_lines() function.

First, I converted the images to grayscale, then I used the parameters that I used in the lesson leading up to the project as magic numbers for gaussian_blur.

The gaussian_blur is used, so that the noise in terms of too many edges would otherwise be detected by the canny function.

The canny edge detection algorithm picks out the parts in the picture that have a big contrast in brightness, as usually occurs in edges between objects. Like a black headphone cable will be a clear contrast to a white kitchen surface for instance. These contrasts are detected and the canny function will essentially draw the dots, of where the contrast is on a plane "canvas" the same size as the original input image. Of course in reality its just an array of numbers that can than be interpreted and displayed as a picture by the computer. The blank canvas is defined at np.zeros_like(img). This means that numpy fills an array in the shape of the image (same size) with 0s. Due to the cv2.fillPoly function only the polygon sized area of that canvas will be filled with non0s, because the cv2.bitwise_and function will return a masked image, or in other words an image where only the region of interest is filled. It takes in the polygon defined nonzero area and overlays the picture. But the part where its 0 stays masked.

Now that there is a picture with the edges and only the edges in the region of interest, the stream of dots has to be converted into lines again.

So after there is the preselected and masked area, I fed it into the Hough_lines function. The function makes use of the Hough algorithm that can detect a series of dots and converts them into lines again.

But before I had to define some parameters again.

In order to do that successfully I first attempted the same values that were successfull in the lessons. But I realized immediately that those didn't help too much here.

I used the matplotlib library in order to display the original picture and the picture with the lines on, in order to do that.

For instance I tried to fix the non-detection of small lines in the further distance in the picture with a smaller max_line_gap. Also the threshold and the min_line_len were tested again and again till I thought that I had sufficient values.

But before I could experiment with the pipeline, I had to build it to the end:

Now I used the weighted_image function in order to overlay the drawn lines onto the original picture. Inside the function it was using the cv2.addWeighted function.

In order to draw a single line on the left and right lanes, I modified the draw_lines() function by ...

1. Creating two channels: one for the left and one for the right border of the lane

I was going with a negative slope (m) originally as I assumed that would differentiate between the two enough.

I went on to notice that it was a bit harder in practice than originally anticipated to find the right values.

2. I made the differences visible by choosing different colors for the right and left lane in order to track if I am only imagining the code working where I want it to work or not.

I than again chose another two colors later when tackling the challenge video in order to see if my changes in the code have the desired effect.

Originally they didn't but making this visual helped me a lot to notice and pin down what is actually going on.

3. I went on to insert "imshape = image.shape" from the pipeline into the draw_lines function as this forced the values to be correct inside the function that I chose to extrapolate extend the lines till the top part of the polygon from the pipeline above and to the bottom of the picture.

4. I later went on to improve on this for the challenge video and tried not only by narrowing down the angle of lines to be drawn (which got rid of the noisy random lines for example in the shadows and other places in the challenge video..) -- in other words I only drew the lines if the slope m was in a certain range -- The display of colors etc. helped me a lot to find the best range.

5. I averaged all right lines and left lines out to be one line only, in order to meet the requirement of drawing one thicker line instead of many lines.

5. In the end I tried to tackle the following problem, that the challenge video had due to the bend the car was driving in the lines going way too far into the picture and even intersecting with each other

-- > I did that by checking the xvalues of the top points of the lines to be drawn

if the left lines xvalue was higher than the right lines value it would obviously going to be intersecting.

So if that was the case I tried to:

a) Detect intersecting lines.

b) Change the parameters in the draw_function only for this

case, so that they would not intersect.

--> This was achieved by taking the difference delta.

Between the x points and halving the difference.

Adding/subtracting a few more pixels so that the left line and right line does not only not intersect any longer but also doesn't even touch in the middle any longer.

c) While doing b) I came acrosse a couple of division by zero errors, that hadn't been an issue before, despite me trying all three videos a couple of times already.

So i prevented those lines to be taken into consideration, in the end of the day if the slope is 0, which was usually causing the division by 0, the line is not interesting i.e. noise anyway... so I let go of it.

d) In order to not loose out on the other lines that might not fall into the categories above I implemented it in such a way that all lines.. (except slope 0 lines that are coming through the range filters)

..will be drawn --> In programming terms: I used if else clauses.

In order to extrapolate all lines in the way it was best for the specific line.

(e) I wrote a few subfunctions within draw_lines function only to keep the code a little less messy and not to duplicate too much...)

(f) I hope that the comments in the code make it easy enough to follow, the draw_function got more and more complex, while trying to tackle the challenge video)

Therefore in a nutshell:

For every line in lines:

The slope m is calculated.

The preselection for left and right lines through slope ranges happens here.

The preselected lines will be averaged out, to only have one line on each side (left and ride).

These remainder lines will be filtered again through the slope range filters just to be on the safe side and partially to avoid the 0 division error.

The left and right lines are checked for intersecting.

If they don't: extrapolate (Drawing an extended version of the line that goes to top and bottom of polygon.)

If they do intersect: The top points for line drawing are adjusted, so they don't intersect and don't touch.

2. Identify potential shortcomings with your current pipeline

One potential shortcoming would be what would happen when ...

the lines go out of the predifined radius. They will essentially disappear right now.

Another shortcoming could be ...

the potentioal misread of a car with similar color and edge (or any other object for that matter, maybe a huge building in the background with white paint and edges somewhat parallel to the road...), even only for a moment for the cameras view.. This might distort the line finding process.

3. Suggest possible improvements to your pipeline

A possible improvement would be to ...

have the algorithm extend and extrapolate the lines into a bend. So that the lines actually follow a bend and bend with the curve the car is driving on.

Another potential improvement could be to ...

try and identify other roadmarkers, like a solid line that is perpendicular to the driving lane, like it is often found in front of/ next to stop signs. And have that thrown into the entire gaussian probability pipeline as an indication to identify a Stop Sign, or other situation when the line indicates a (temporary) stop..