Michael Eliot

1) Describe your pipeline

My apply_pipeline works similar to how is described within the course material. I first convert the image to grayscale. Next, I apply a canny transformation to get the outlines of various objects within the image. This gets me all locations where line segments may appear, so I create a mask on the canny transformation that essentially crops the lines down to the area of most interest, i.e. in front of the car. I further cut down my image by applying the hough transformation, identifying the areas with actual lines in the image. This helps separate random points from actual full lines. Here is where draw_lines comes into play, which I will touch on in the next paragraph. Lastly, I create a color binary image, and draw the lines from the hough transformation on this color binary.

Draw_lines is the most difficult part of the project. What I do is sum each value of the endpoints of the lines together, and then average them. For example, I take all the left-most x values for the right lines, sum them up, and then average them out to get one x value for the first point of the final right line. I then do that with the rest of the values for the rest of the points, separating it by left and right line. I then extrapolate these newly created lines by selecting a topmost and bottom most y value from the image, and applying my line equation found previously to these different points. I conclude by capping the max distance a right line can be to the left, and a left line can be to the right.

2) Identify Potential shortcomings

The algorithm works well overall, but there is definitely room for improvement. The criticism is mainly weighed against the capping system. There may be instances where the right lane should go above the cap, and this algorithm wouldn't detect it. It's a tradeoff between assurance of accuracy, i.e. not allowing stray points to pull the line awkwardly, and considering potential edge cases within line movement.

Another shortcoming could be the need for averaging amongst line segments. In some instances, these line segments may be spread apart or curved, and thusly an averaged set of these points may not turn out as accurate.

I will conclude by indicating I had some trouble with the line suddenly jumping askew, and then snapping back to its predicted place for a frame or two. One could either expand the capping system, which has its tradeoffs, or expand how to remove unnecessary points.

3) Possible improvements

The first improvement is to continue fiddling with certain variables, or even dynamically adjusting these variable to certain input. For example, the mask is simply a hardcoded trapezoid in front of the car made through guess and check. These points not only should continue to be

optimized, but we also might adjust the trapezoid itself under certain criteria like turning left or right.

Next is changing draw lines. I feel like the draw lines currently presented is very strong; however, I think improvements can be made. The summation through the various lines feels crude, but I can't think of a better way to handle the conglomeration of the data. Additionally, the extrapolation is hardcoded in right now, and should be optimized beyond just me eyeballing it. Lastly, is the fluidity issue. Because lines don't change significantly, it would be cool to try and use previous lines to inform our current prediction of the line. This would enhance the fluidity of the line searching, prevent sudden twitches, and minimize the need for a capping system.

After giving it my try, I looked at other ideas posted around like https://github.com/naokishibuya/car-finding-lane-lines and https://github.com/srikanthpagadala/udacity/tree/master/Self-Driving%20Car%20Engineer%20Nanodegree/LaneLines-P1 which highlight some of the differences one could make. However, I did not use any of their code/ideas within my own work.