David Clark
2/22/2017

P1 Reflection - Udacity Self-Driving Car Nanodegree

Overview

Project #1 is about creating a pipeline that detects lane lines in images and draws them onto the original image. While the pipeline is created for a single image, it can be applied to video footage by breaking the video down into frames, passing the frames through the pipeline, and then reconstructing the video.

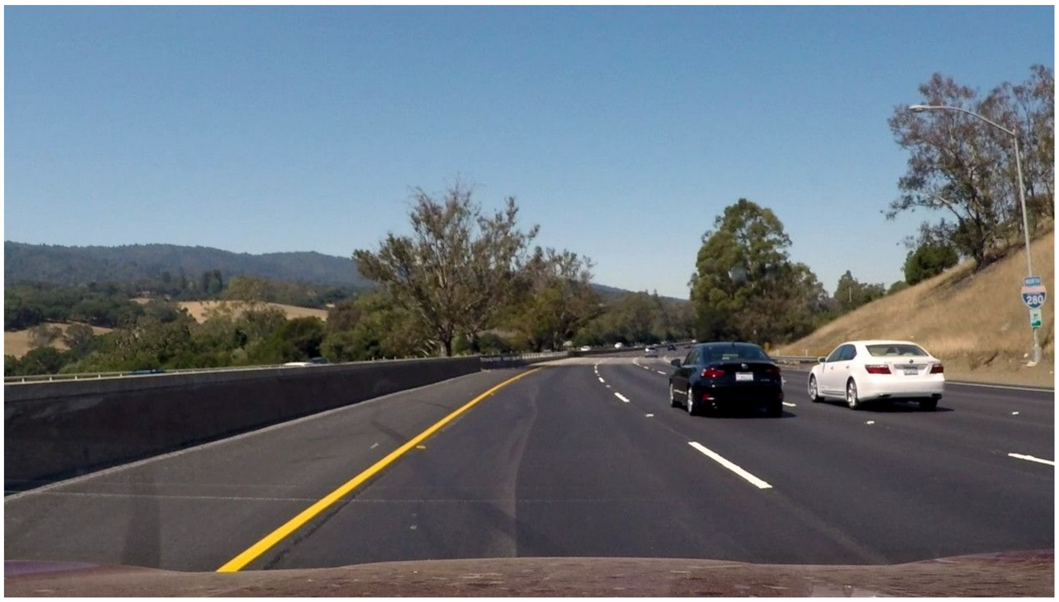
My code for this project is publicly available and can be found here:
https://github.com/SealedSaint/CarND-Term1-P1

Describe your pipeline. Explain how you modified the draw_lines() function.

My pipeline consists of the following six major steps:

1. Grayscale
2. Gaussian Blur
3. Canny Edge Detection
4. Region of Interest - Create Vertices
5. Hough Transform - Average Lines
6. Draw Lines

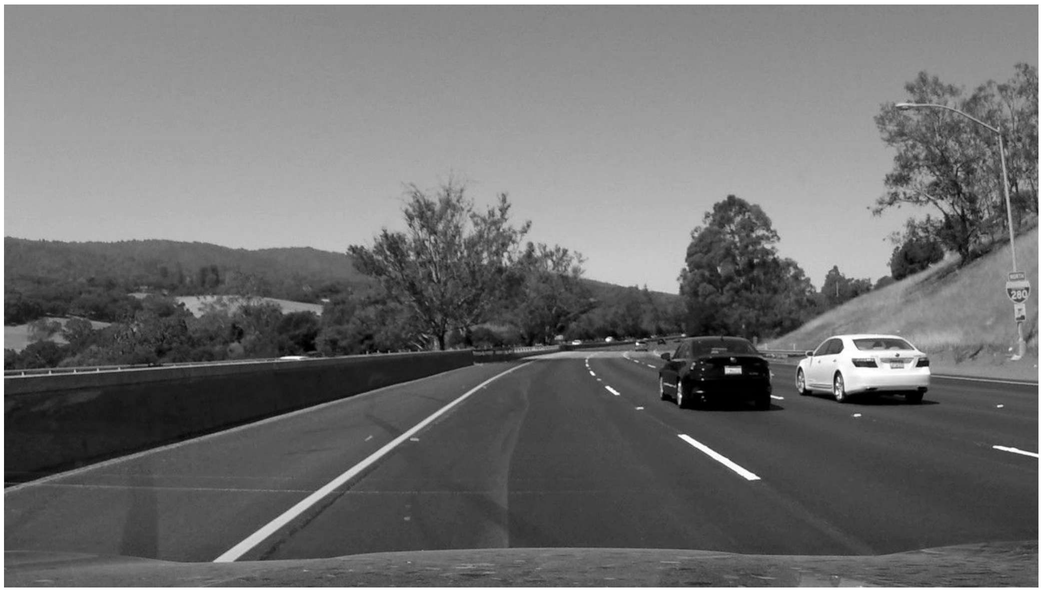
Let's use the image below as our original:



David Clark
2/22/2017

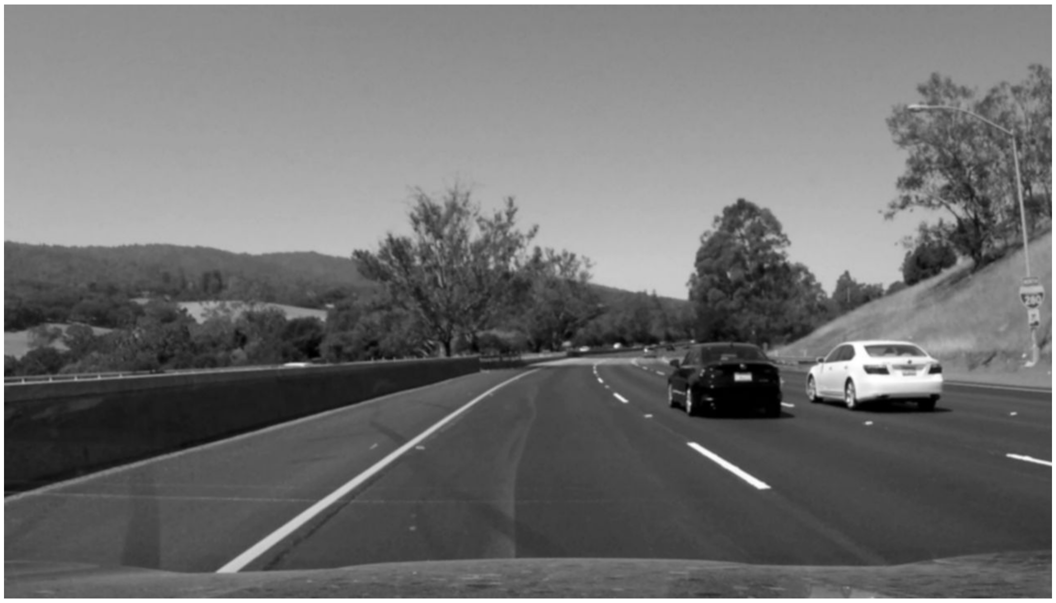
Grayscale

The first thing we do with the raw image is convert it to grayscale. Not only does this reduce the amount of data and complexity we are dealing with, but it also benefits other steps like step #3, Canny Edge Detection.



Gaussian Blur

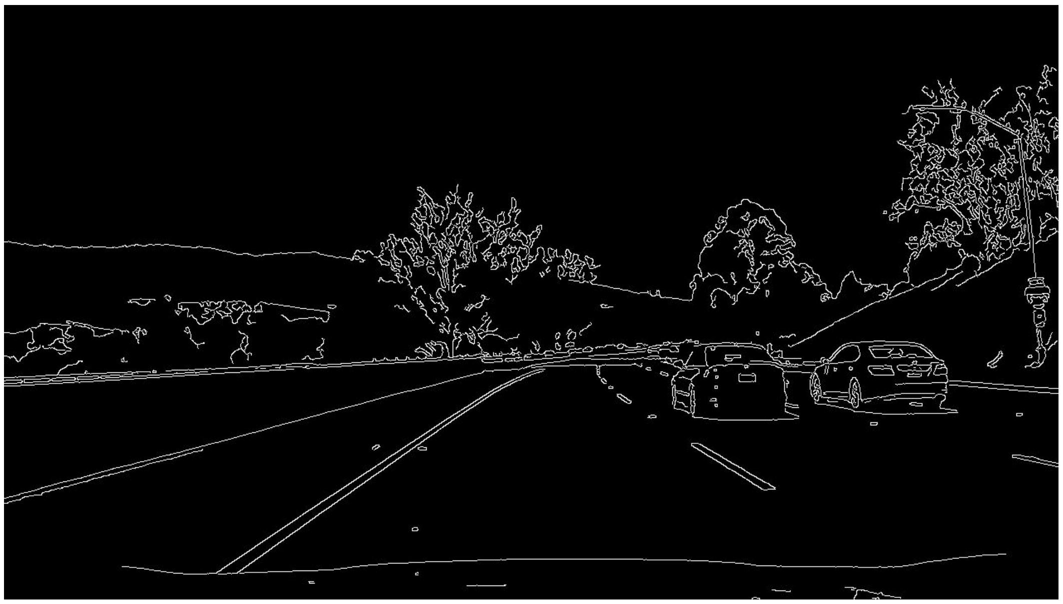
Before edge detection, we want to smooth out the image some. Applying a Gaussian Blur will cut down on visual noise and ensure that only the sharpest edges get through.



David Clark
2/22/2017

Canny Edge Detection

Personally, I find the result of this step really neat. Canny edge detection is an algorithm that measures the change in connected pixel values (the gradients). We use it to turn our image into pure black and white where white represents the largest gradients - the most drastic changes in connected pixel values. These represent our "edges" in the original image.

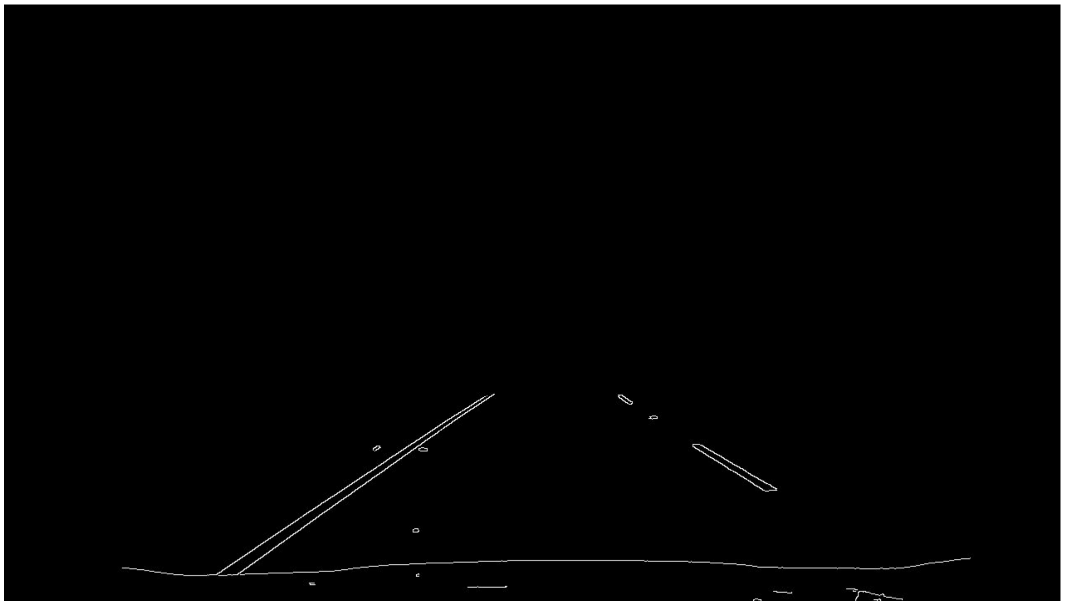


Region of Interest - Create Vertices

If we are only looking for lane lines, we certainly don't care about all of these edges. We apply a region of interest that allows us to ignore white pixels where lane lines shouldn't be (in the top of the image, for example).

The region of interest function calls another important function - create vertices. I wrote the create vertices function as an alternative to hard-coding the coordinates for the region of interest. If the car is cresting a hill or coming out of a valley, the horizon of the road will change vertically. Similarly, if the road is turning, the lane lines will veer to the right or left. The create vertices function will dynamically determine the region of interest on a per-image basis, accounting for these varying conditions. In a nutshell, it works by searching from the bottom-center of the image (where the lane will be) outward until it encounters lines. The region is then set at these discovered points, plus a little bit of buffer to include the lines.

David Clark
2/22/2017



Hough Transform - Average Lines

After we have our regioned-out edges from the Canny algorithm, we pass the image through a Hough Transform. In a basic sense, the Hough Transform will scan the image and detect where pixels form lines. If we set our parameters correctly, this will return our lane lines.

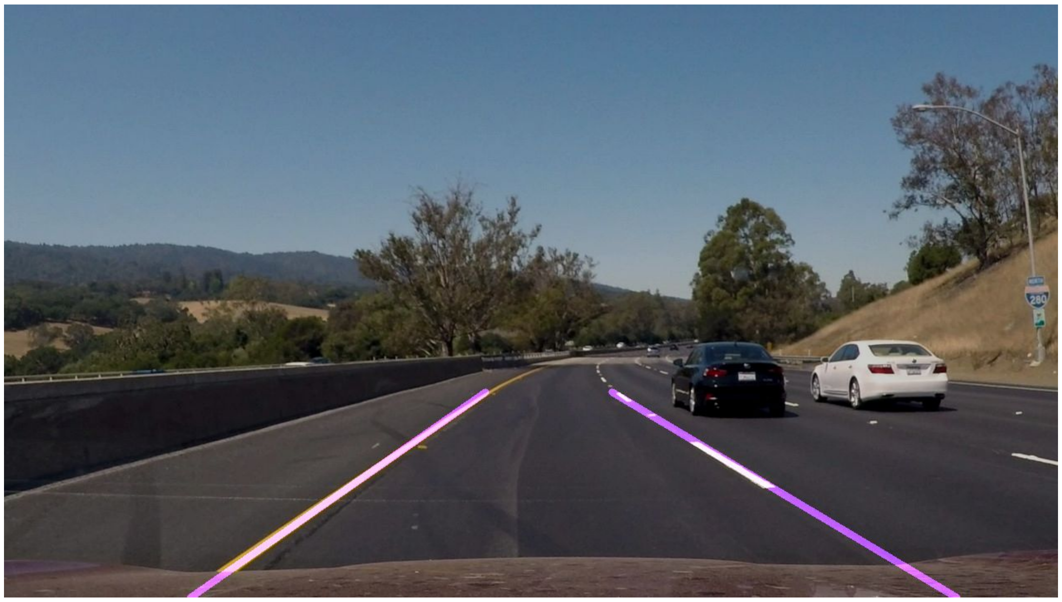
The reality is it's hard to get single solid lane lines out of a Hough Transform, especially when one of the lines is dashed. Even the solid lane lines are often made of multiple Hough lines. I created the average lines function to take the line information output by the Hough Transform and reduce it down to two lines - our ideal lane lines. This mostly works by comparing the slopes of the lines. We ignore lines with slopes too horizontal - those won't be lane lines. We then group the lines by positive and negative slopes - our right and left lane lines. From that separate line information we average the points together and extrapolate out to the top of our region of interest and the bottom of the screen. This gives us two solid lane lines (per project specs).

(My average lines function is my version of altering the draw_lines() function. I left the draw function as-is because it made sense to keep it only as a drawing function.)

Draw Lines

The last step is simple and boring - we simply draw the final lines onto the original image. This pipeline draws lane lines onto a single image, so for videos each frame is passed through this pipeline and returned - forming a new video with marked lane lines.

David Clark
2/22/2017



Identify potential shortcomings with your pipeline.

I did my best to reduce specific dependencies in my pipeline (such as certain image resolutions). One of the ways I did this was to include the create vertices function as part of the region detection process. Ironically, while create vertices is great at dynamically handling various situations, it’s also one of the weakest points of my pipeline. The algorithm isn’t super smart, so the smallest of edges in the middle of the road (if detected through Canny Edge Detection) will severely limit the region of interest, usually causing lane lines to go completely undetected.

Suggest possible improvements to your pipeline.

Given that my create vertices function which dynamically determines the region of interest is probably my pipeline’s weakest point, I think something different should be done there. With some more data, I think hard-coding a larger region of interest to account for various driving/terrain scenarios might be appropriate. Or perhaps a simpler approach of altering a standard region of interest based on driving data (like steering angle) might work well.

I also wonder if filtering the original image down to white and yellow at the very beginning would improve the process a lot. If we can make the assumption that lane lines are always white or yellow this could help quite a bit, it seems.