Project 4 - Advanced Lane Lines
Udacity Self-Driving Car Nanodegree
April, 2017
By: William J. Keller

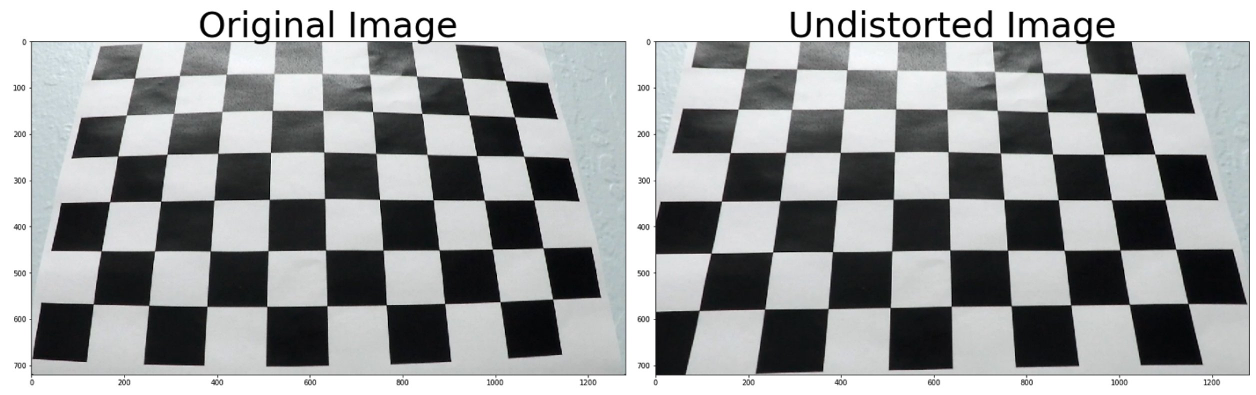
Writeup / README

This is my project writeup.

Camera Calibration

This is contained in the first two cells of my jupyter notebook. The first step is to read in a collection of chessboard images with which to calibrate the camera. For each image, I used the “findChessboardCorners” function to find the interior corners on the image; those were then appended in an array while a companion set of known “real world” points were appended to a second array.

The second calibration cell takes the corner points found above and uses the “calibrateCamera” function to undistort one of the calibration images for inspection. The results are below:



Pipeline (single images)

1. Provide an example of an undistorted image
   This was done using the same process as was used for the inspection chessboard above. The code specific to this image does not exist in the notebook I’m submitting as I undistorted the individual images in a previous iteration. The third cell, however, contains the function my pipeline uses on the frames from the video.

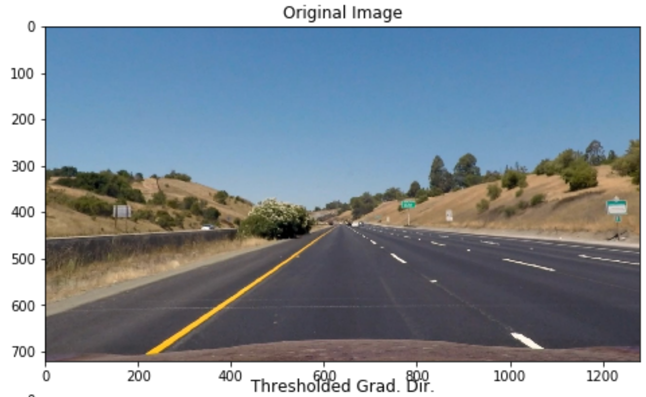
2. Threshold Image Processing
   This is done in the fourth, fifth, and sixth cells in my notebook. The fourth applies the Sobel algorithm, and then finds the direction of the gradient then returns the binary image similar to the example at the bottom of this section.

   The fifth cell applies the Sobel algorithm and returns a binary image.

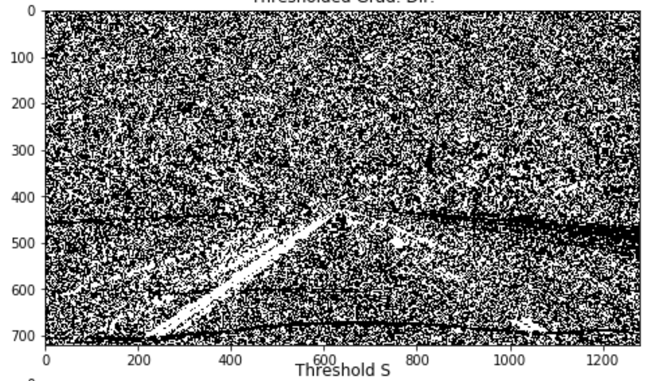
   The sixth cell converts the image from RGB color to HLS color. It then applies a threshold to the S color channel and returns the resultant binary image

   Within the main "process_image" function in my pipeline (ninth cell), the three binary images are combined into a single image reflecting pixels in both the Sobel and direction gradient images or in the S-threshold image. I experimented with a couple different combinations but found this to highlight the lane markers most successfully. (I realize the images are not in the same order as the above paragraphs. I have the paragraphs in the order they appear in the final pipeline, but the images reflect an earlier iteration of development.)

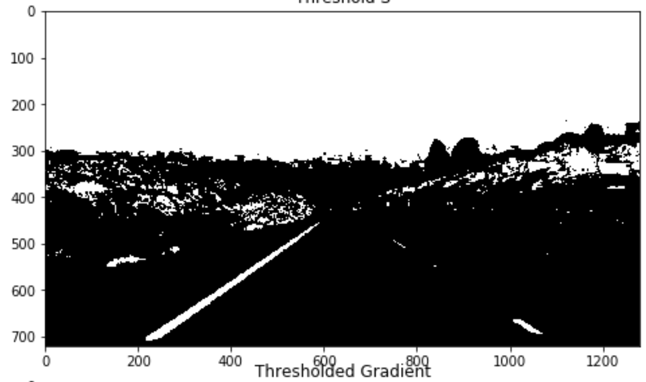
Original Image



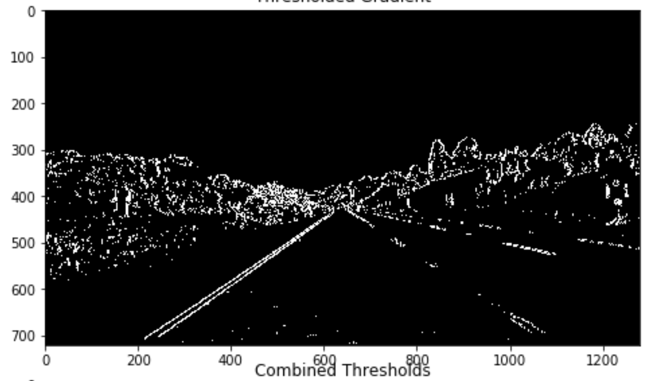
Thresholded Grad. Dir.



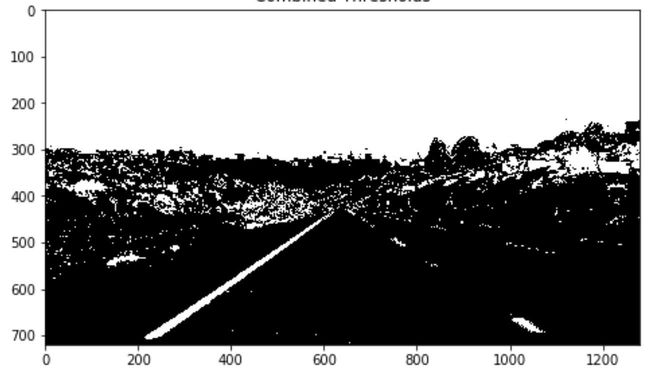
Threshold S



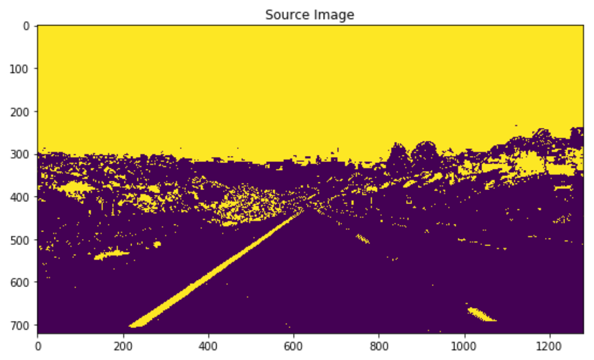
Thresholded Gradient

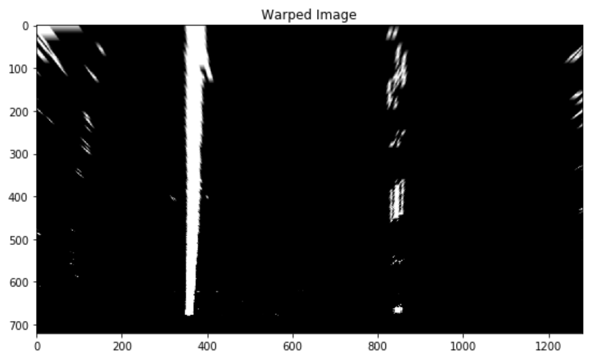


Combined Thresholds



3. Perspective Transform
   The seventh cell in my pipeline contains the “warp” function which transforms the image from the real-world looking ahead perspective to a simulated “bird’s eye view” of the road ahead. During the development process, when I was fine-tuning with individual images, I found the points used for the perspective transform, which are then hard coded into the pipeline. (The RGB is wrong in this example, but it’s the same image used above, only in BGR.) (I fixed this in response to some feedback on my first submission suggesting I could get the lines a little closer to perfectly parallel.)





4. Lane Line Pixels and Fit with Polynomial
   The eighth cell in the pipeline contains the bulk of the processing to find the actual lines. It begins with a histogram of the bottom half of the image to find the nexus of pixels likely representing the starting points of the two lines. I then used a couple of if statements to try to eliminate wild jumps in the lines to the far left or far right. After that, the code compares the computed center of the lane with the center of the image to determine where in the lane the vehicle is traveling. Then we split the frame into nine rows and go through each row looking for a high number of pixels to indicate the lane line. The windows can move to the left or right based on where the previous pixels were found. I added another failsafe here to keep the window from moving too far if a stray shadow or other distraction momentarily fouls the process.

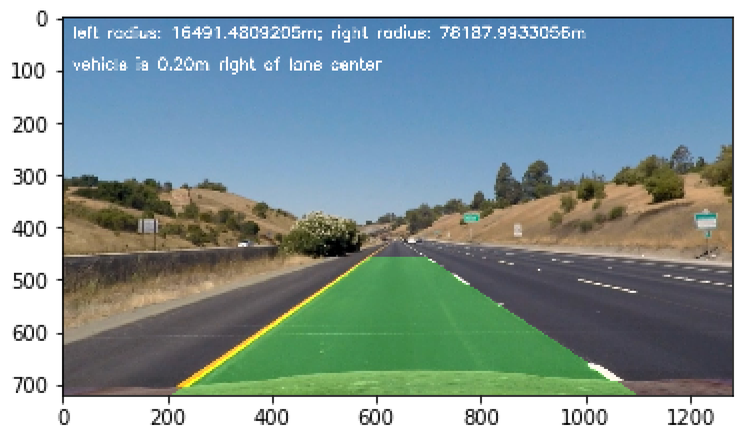
   Once the pixels have been found, numpy polyfit is used to fit a second order polynomial for the left and right lane lines. From there, x and y coordinates are generated to plot on the image.

   At this point, the program calculates the curve radii in pixels and then convert into meters.

   Then, it plots the lines on the warped image and de-warps it back to its original perspective. I then add some text giving the curve radii and the vehicle’s estimated position within the lane.

5. Curve Radius and Vehicle Position
   I covered this in the previous section, but both of these are handled in the “major process” function in my pipeline.

6. Example of Finished Product


left radius: 16491.4809205m; right radius: 78187.9933056m
vehicle is 0.20m right of lane center
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Pipeline (video)

This is included in my GitHub submission.

Discussion

I found this project to be an enjoyable (occasionally frustrating) challenge. Overall, I’m pretty pleased I was able to get as far as I did, especially integrating the video controls from Project One into this project.

The left lane line is pretty well handled by this iteration of my project, but I’ll admit the right side is still a little temperamental. A more robust and subtle approach to mapping the lines would help here - my “if” statements are a bit blunt with how they handle possibly errant line determinations. I could speed it up a bit, I think, using the method described in the lesson to start from a known line segment rather than trying to find each one anew.