Advanced Lane Finding Project

The goals / steps of this project are the following:

Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.

Apply a distortion correction to raw images.

Use color transforms, gradients, etc., to create a thresholded binary image.

Apply a perspective transform to rectify binary image (“birds-eye view”).

Detect lane pixels and fit to find the lane boundary.

Determine the curvature of the lane and vehicle position with respect to center.

Warp the detected lane boundaries back onto the original image.

Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You’re reading it!

Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

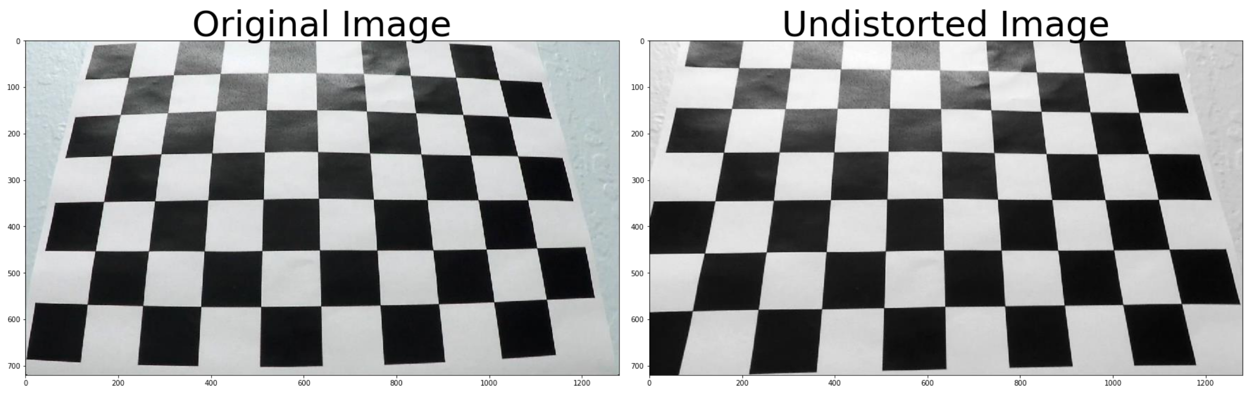
The code for this step is contained in the first code cell of the IPython notebook located in “AdvancedLaneFinding.ipynb”.

The camera calibration requires object and image points taken at different angles and distances. Using the 20 images in the “camera_cal” directory, I was able to use the cv2.findChessboardCorners function to find the corners on the chessboard images. These corners can be fed into the cv2.calibrateCamera function to compare against the actual corners and come up with a calibration matrix.

Pipeline (single images)

1. Provide an example of a distortion-corrected image.

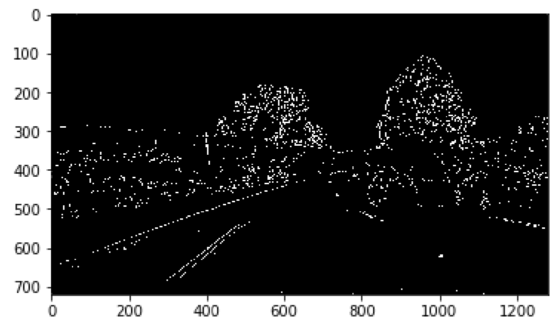
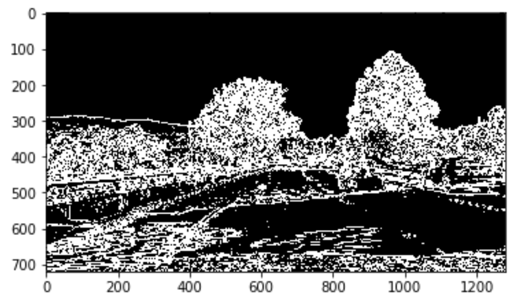
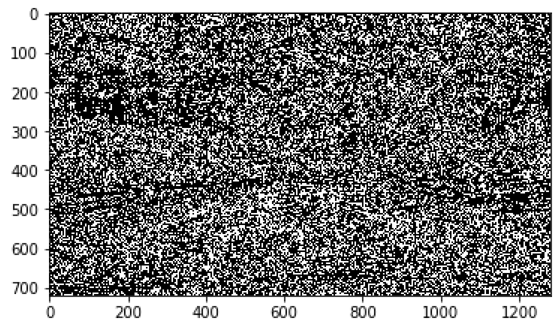
After coming up with the camera calibration matrix, I used the cv2.undistort function to get the proper distortion-free image. This pulled in the camera calibration data to undistort the image and validate that the model is working properly for later operations.



I also applied the undistort feature to a road image to evaluate if there were any visible differences, but they were somewhat minor and hard to tell with the naked eye.

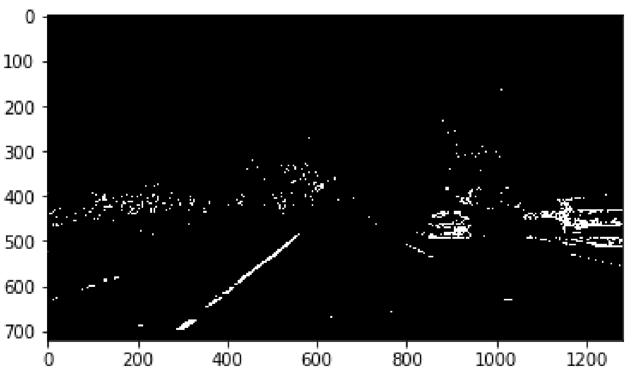
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The Sobel operator was covered extensively in the coursework. I used the X-direction, Y-direction, magnitude, and direction Sobel operators to try to come up with a binary thresholded image for later processing. The following examples are operators performed on test4.jpg:

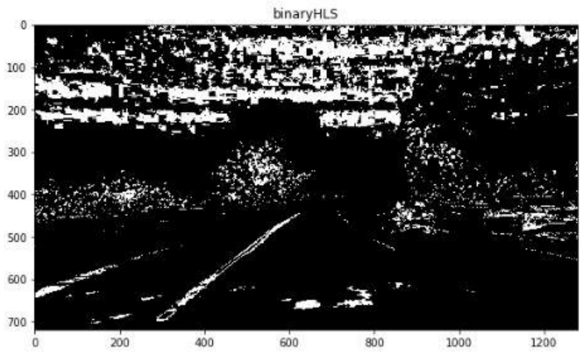
	
Sobel X	Sobel Magnitude
	Blank
Sobel Direction	

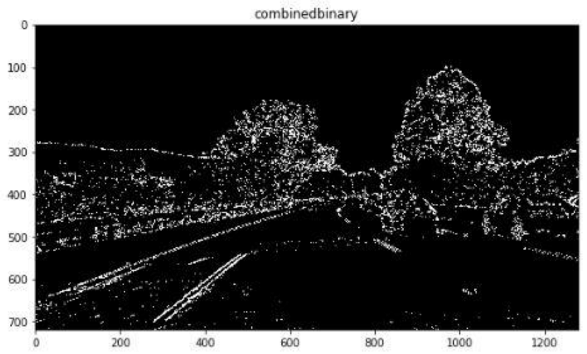
I had to extensively tweak the lower and upper thresholds so that the road was properly sectioned off. I did a little bit of tweaking of kernel sizes but it did not seem to affect results as much as thresholding did.

I used the HLS transform as specified in the coursework to generate a binary image that was sensitive to the yellow color of the left lane line. This was critical because the video in the project has a section where the pavement goes from dark to light, washing out the left yellow line. This dark to light transition causes the Sobel filters to not work as well. After adjusting thresholds, I got great results with HLS with tight thresholds as seen below.

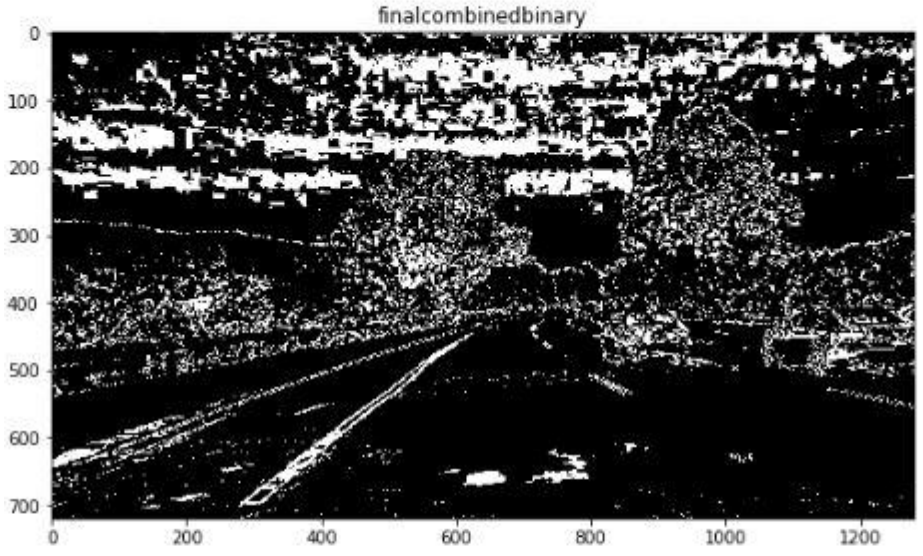


However, the yellow line was not being seen far enough ahead so I had to open the threshold a bit wider. Finally, I combined the HLS and the Sobel binary threshold images as seen below.



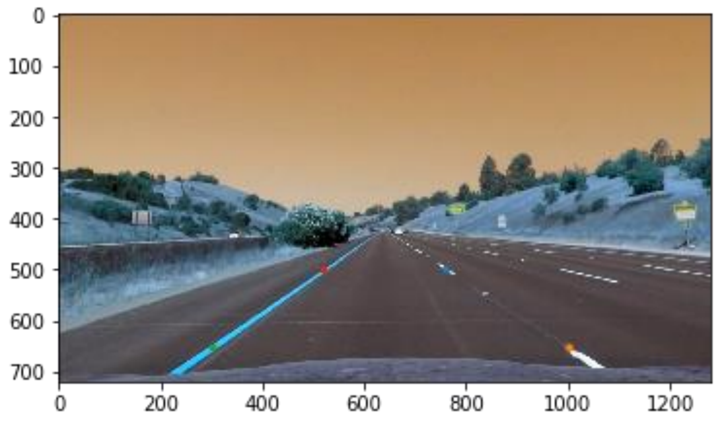


This resulted in a beautiful combined image where the lines could finally be observed clearly and within the light-colored pavement.

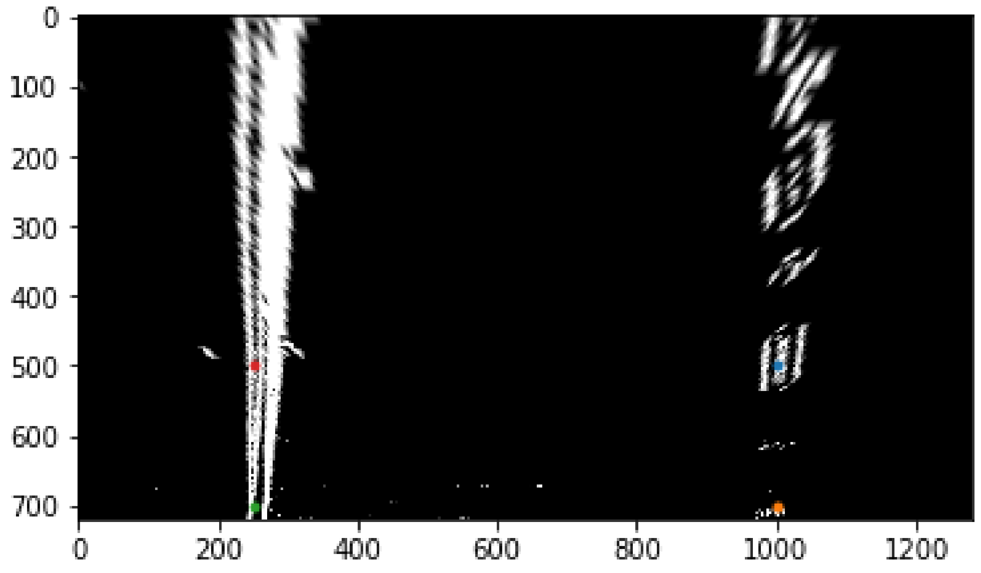


3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

First, I used the plotted image with coordinates shown to get an approximate idea of where the transform coordinates should be.   These four points are shown against the straight_lines1.jpg image below.

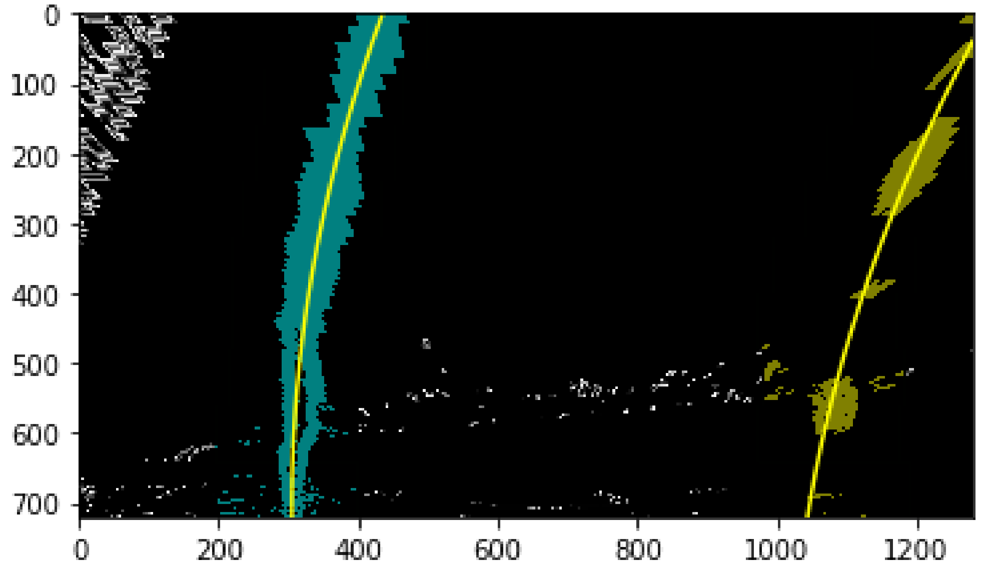


I then squared off the image by making the upper two points match the width of the lower two points, which should create the birds-eye view. I used the cv2.getPerspectiveTransform to achieve this, and the results were promising on the binary image. I expected them to be straight parallel lines given that the input was of straight_lines1.jpg:



The next objective was to determine the curvature of the road. To do this, I needed to use the windowed histogram search function try find histogram peaks. The coursework covered the method of starting with the histogram of the bottom half of the image, then working your way upwards on the image, adjusting centers to adapt to the curvature of the road. It used the peak of the histogram in each particular region to ensure that lane lines were relatively close to where they were in the previous window, so that noise would be less likely to affect the data.

I largely used the code given as part of the coursework, and noted the way that the code used sliding windows 100px wide with a 50px deviation as a way to find the relative position of each set of lane lines. The adjustment in each window that moved to the next region of lane lines was particularly innovative.



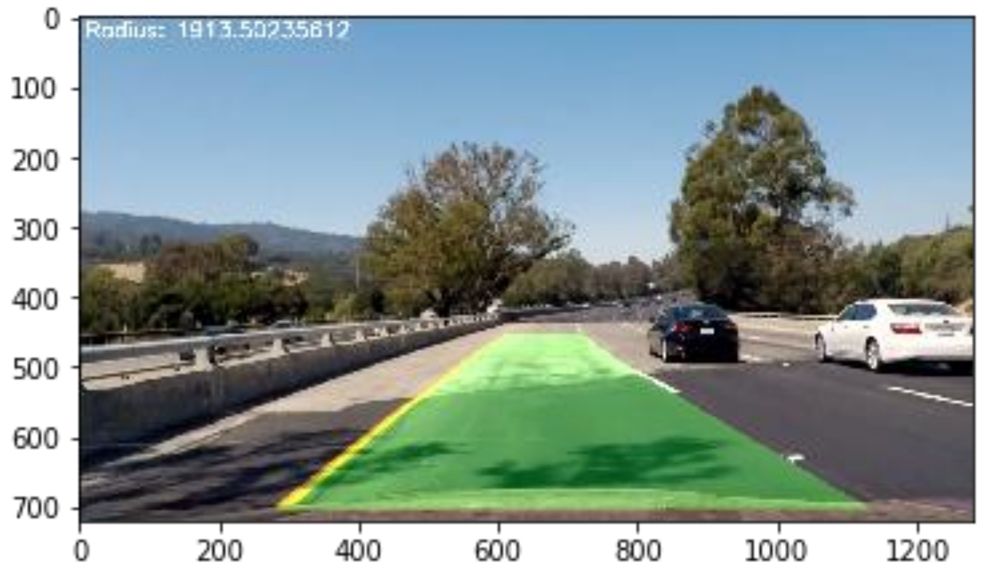
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

After each region of lane lines was detected, a polynomial fit was performed on the warped image, which now give us the data to apply a smooth over the image, and more importantly allows us to determine what the curvature of the road is. This is found in Section # 4 of the Jupyter Notebook.

In Section # 5 of the Jupyter Notebook, I calculate the radius of curvature using the equations given in the coursework. This data could help the driver know if the speed is too fast for the curve of the road. I was able to calculate the curvature of the road given the polynomial fit coefficients and use that data to determine the radius of curvature. With the polyfit data, I was able to have the image generate a 2D overlay showing where the system thinks the lane is. In the warped space, this would be a line in the center between the two lane lines. I also used the x-coordinates at the bottom of the image to measure how centered the car was within the lane. I assumed the center of the camera view was also the center of the car, and measured deviation, given that the width of the lane lines is 3.7 meters.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

To be able to plot this against a non-birds-eye-view image, the image would need to be warped back to the original perspective view with vanishing points. I used the inverse of the warping function to regain the view once a green overlay had been drawn over the roadway. The result is shown below, where the green area shows the curved road ahead. The calculated radius is plotted in the upper left hand corner, corrected for the length and width of the road and the lane line design guidelines from the US highway system (30m from one line to the next, and 3.7m wide).



Success!

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

My video result is in the directory ./output_images/project_video_output.mp4

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

I thought my implementation was pretty robust since it made it through the dark/light transitions pretty effectively. However, once I ran my model through the challenge videos, I realized just how inadequate they were. I had to further refine my warped image guidelines to ensure that I was seeing road curves far enough ahead, but not too far ahead. I also had to tweak my BGR2RGB to ensure that the inputs to functions were correct, since I relied on the red channel heavily.

I noticed that my algorithm is easily fooled by fresh asphalt laid on top of light colored pavement as is seen in challenge_video.mp4, which triggers a Sobel operator activation and results in the pavement transition being treated as a lane marking. This is tough to overcome without additional processing. However it would make significant rewards and result in the ability to drive in a wider range of conditions.

In the harder_challenge_video.mp4 file, my model does poorly, and was easily fooled by the huge changes in lighting, where the sun was at a low angle and polluting the image with lens flare and reflections. With perhaps some averaging of the previous steering angle, I could perhaps beef up the search algorithm to utilize more history of the recent steering angles to predict what the next ones will be.