Advanced Lane Lines Project

Brent Wylie, May 4th, 2017

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.

In this writeup 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.

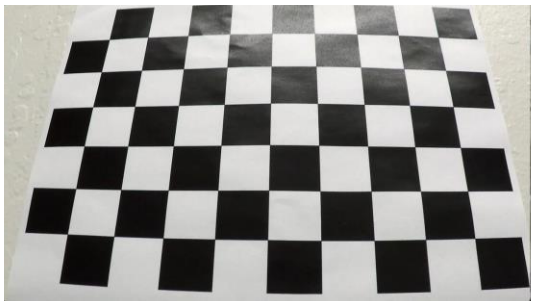
This document will consider all of the rubric points below.

Camera Calibration

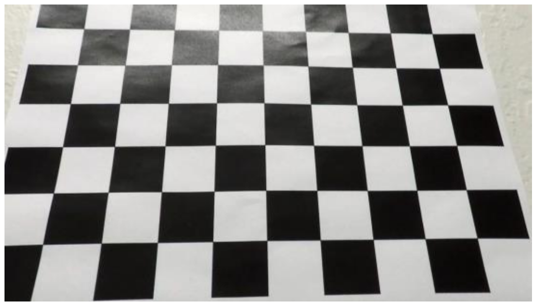
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

All cameras have some physical intrinsics that can distort pictures to some degree. In order to counteract this, I used a set of calibration images consisting of a known checkerboard pattern in conjunction with the OpenCV functions “findChessboardCorners”, “drawChessboardCorners” and “calibrateCamera” to compute the camera matrix and distortion coefficients. Using these parameters along with the OpenCV function “undistort” allowed me to undistort the below camera image. This was performed in section 1.0.

Distorted Image



Corrected Image



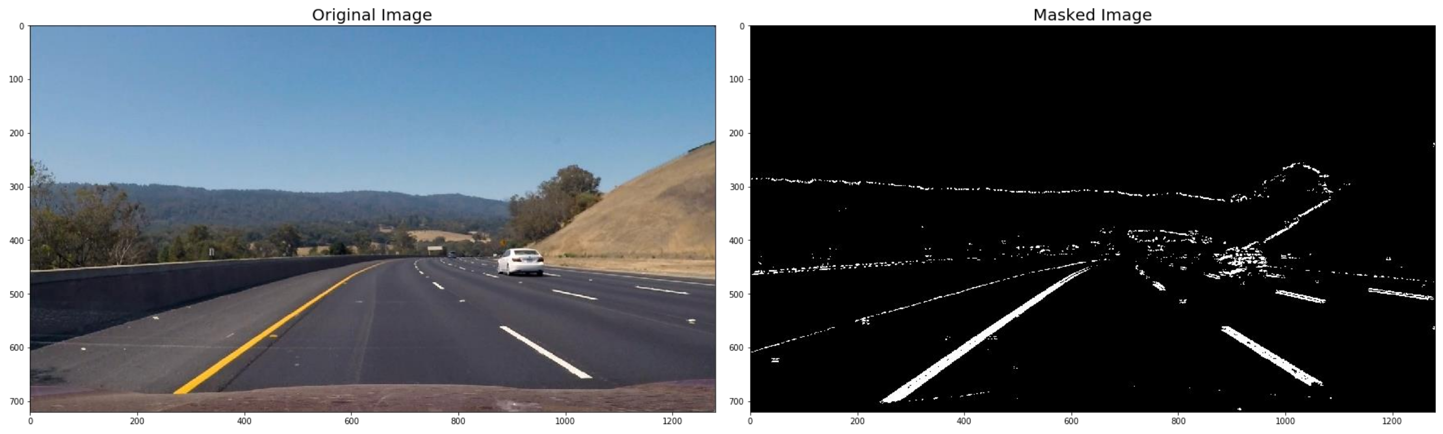
Pipeline (single images)

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

Here is an example of a corrected image.

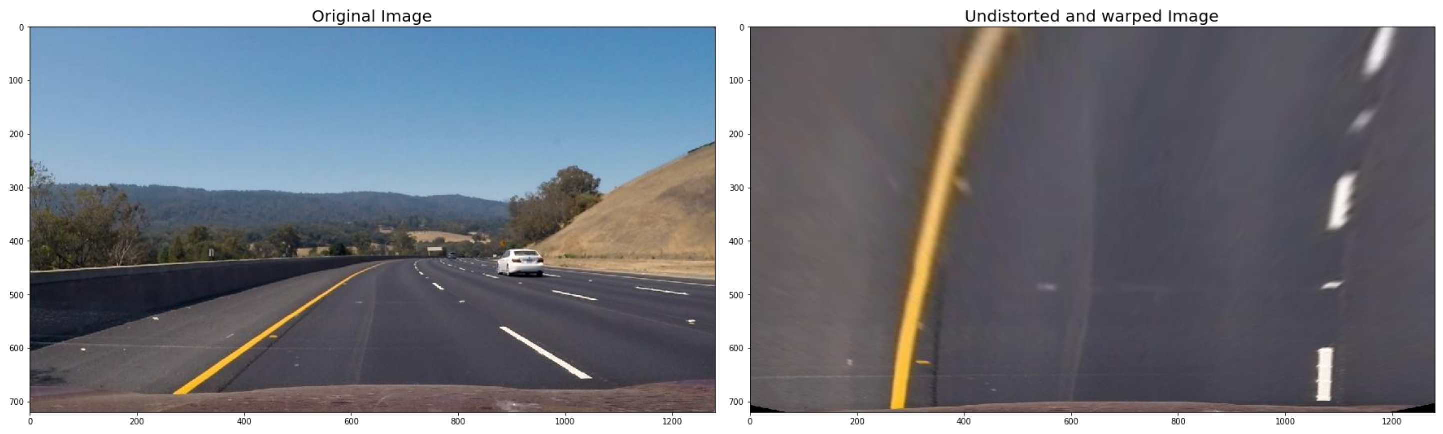
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I created numerous helper functions for thresholding the image data in various ways. I then combined various techniques in a thresholding function to single out lane lines. The helper functions are defined in section 2.0 and the combination of techniques (and subsequent testing of techniques) in section 3.2. I ended up using a light Gaussian blur (kernel size of 3) to prevent noise from skewing results, followed by S-channel thresholding combined with sobel, gradient magnitude and directional thresholding. Below is an example of the thresholding result.



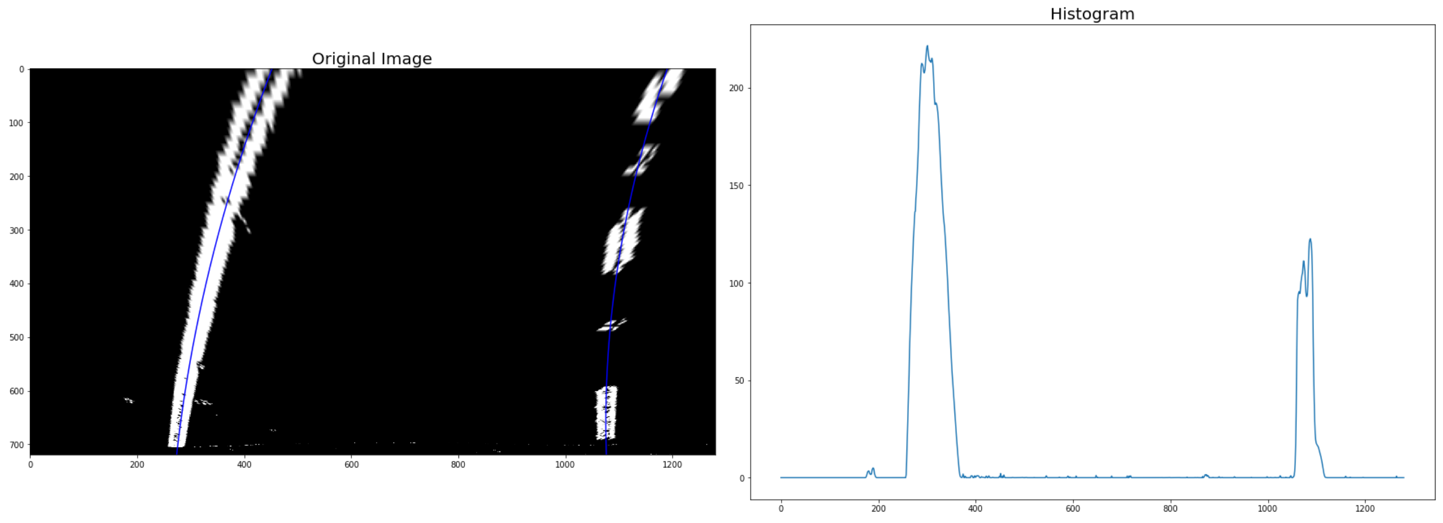
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I defined a helper function ‘transform’ in section 2.0 to perform the perspective transform. I chose to hardcode the source and destination points within this function. The function is tested along with the undistort function in section 3.0, then combined after the thresholding operation in section 3.2. Below is an example of the perspective transform being applied to a full-color image.



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

With the thresholding and transformation process solidified, I implemented lane detection, curvature analysis, and defined a lane class in section 4.0. The findlanes() function in section 4.0 was the starting point, where I performed a sliding window search based on the histogram of the warped and thresholded image. This performs a search every time and does not perform any history-based smoothing. Then using np.polyfit() I was able to find an appropriate polynomial and plot it on the binary warped image, along with the histogram results.

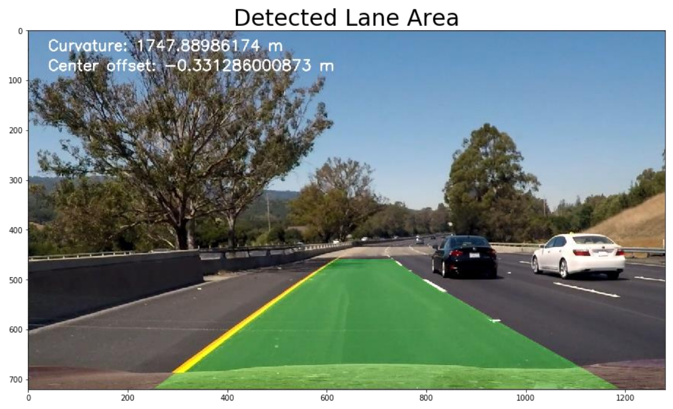


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

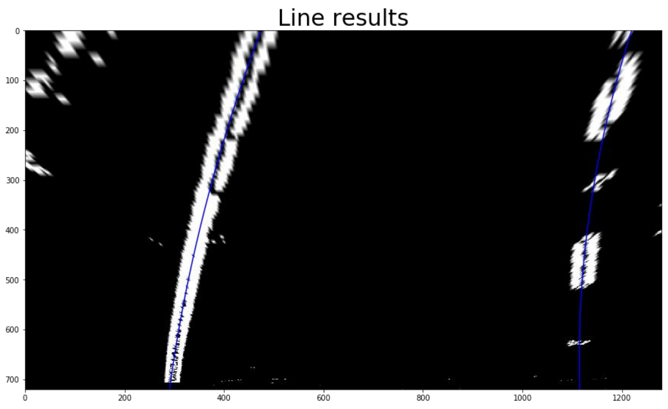
I calculate the radius of curvature and the center offset of the vehicle in section 4.0, specifically in the functions find_curvature() and find_center_offset() respectively. To determine the curvature, I calculate the curvature based on pixel values using the fitted polynomials, then translate them approximately to real coordinates as described in Lesson 35. To determine the center offset of the vehicle, I calculate the average midpoint of the lane using the fitted polynomial lines, and then assume the center of the vehicle is the center of the picture. Using both of these data points, the offset is the distance from the center of the vehicle to the center of the lane. This is also first calculated in pixel coordinates, then translated into real coordinates.

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

In section 4.1 I defined the function draw_lane_area(), which draws a shaded area onto the warped binary image, which is then unwarped and stacked on top of the original image. An example of the result is shown below.



Curvature: 1747.88986174 m
Center offset: −0.331286000873 m



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!).

The video is contained in the same zip file that this writeup was in. It is named ‘project_output_submission.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?

This pipeline has struggled the most in areas where the pavement color differs from darker colors, and where shadows dominate the road surface. I’ve introduced an averaging system to make it slightly more robust, but this is a tradeoff as it reduced the overall responsiveness of the system. I also noticed that the bumps in the road seem to throw off the detection as well—my hypothesis is that this is because the perspective transform is done statically. If some image stabilization could be done on the image, the lines would be a bit more stable. In addition, the averaging system reduced the system’s responsiveness which slightly exacerbated the system’s ability to deal with road bumps. However, this approach was still better than the alternative, as even during the road bumps the car’s projected lane area is more reliable than without the averaging system. One final, stronger way to solve the problem of finding lane lines would be to use a segmentation neural network to determine where the lines on the road were located. This solution would be far more robust at distinguishing the true lines versus things that look like lines, such as old lanes, scraped asphalt, paint spills, etc.