Advanced Lane Finding Writeup

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!

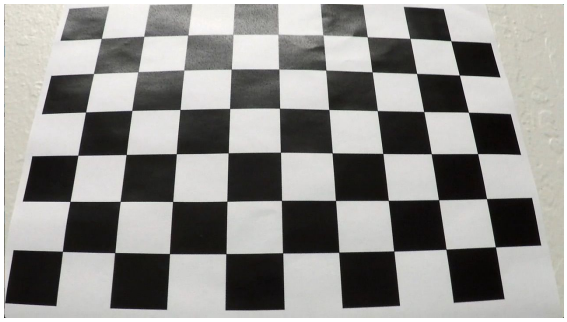
Camera Calibration

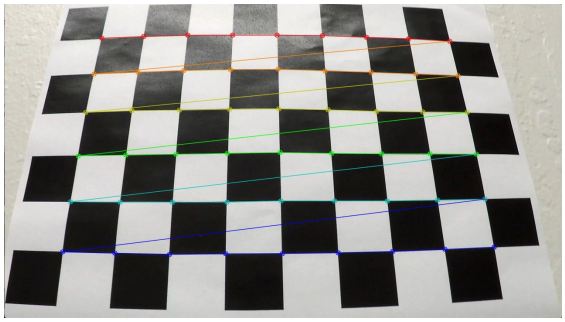
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

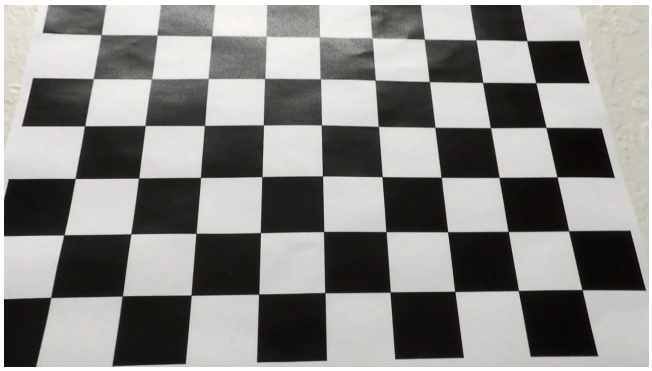
Since the camera introduces distortion, first we need to calibrate it to find out distortion coefficients and use them to undistort images. In my code, in lane_finding.py, I do this on lines 285-302. I read in images of chessboards taken from different angles and distorted, and I find corners on the chessboard using the findChessboardCorners() function.

I create an array of object points, which are (x, y, z) coordinates of chessboard corners in 3D space and an array of image points, which are coordinates of the corners found in the chessboard images. As I read in images in a loop and find corners, I append the result of the function call to my image points array. Building up these arrays eventually gives me enough information to determine values needed to calibrate the camera and undistort images.

The images below show how chessboard corners are detected on an image of a chessboard. The last image is the result or running the undistort function on the first image.







Pipeline (single images)

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

Distortion correction makes up for the fact that the camera lens ends up distorting the image when we take a picture. Distortion can make object appear different in size and shape than

they really are. In my code in lane_finding.py, I undistort on line 332. Here is an example of a distorted and undistorted image:

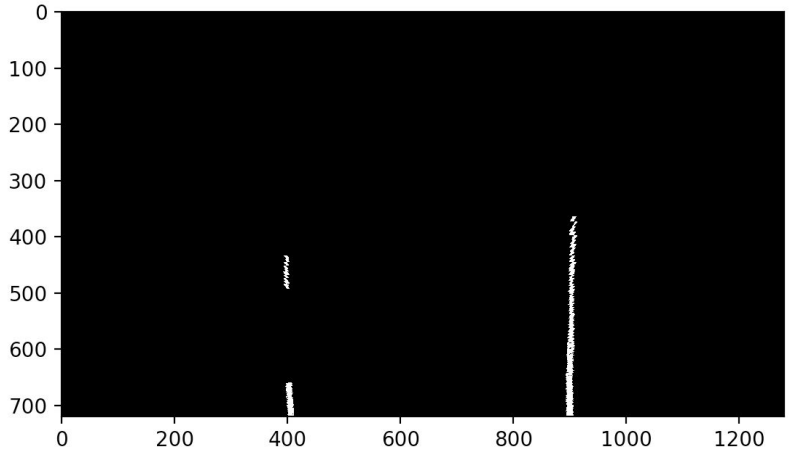
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I experimented with different color spaces and different techniques by looking for features that each of them picked out well. I applied one method at a time to all images to see if they provided something useful. For example, some features were excellent at detecting lanes even on a lighter section of the road, but lost the lanes further in the distance. Others captured the curve in distance well, but introduced some noise. I carefully thought through the logic I would need to combine those features (whether I would use a union or an intersection).

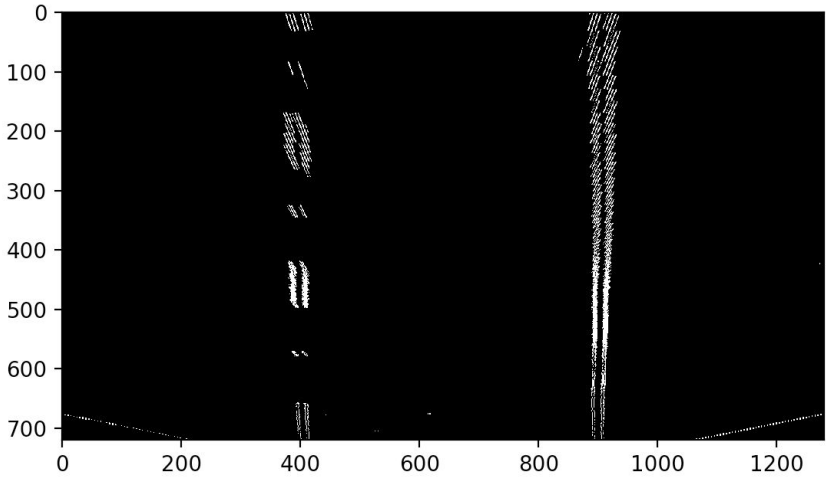
At some point I experimented with hue, because if tuned correctly, it did a good job picking out the solid yellow line in almost all light conditions. However, in the end, I decided to go against it, since I was getting sufficiently good results with other methods, and swapping the order of thresholding and transform broke my hue tuning.

I ended up going with a saturation image and an x sobel on a grayscale image, both converted to binary. I combined them with and “or” operation since they both picked up on features I needed, and doing the transform first helped me avoid a lot of extra noise.

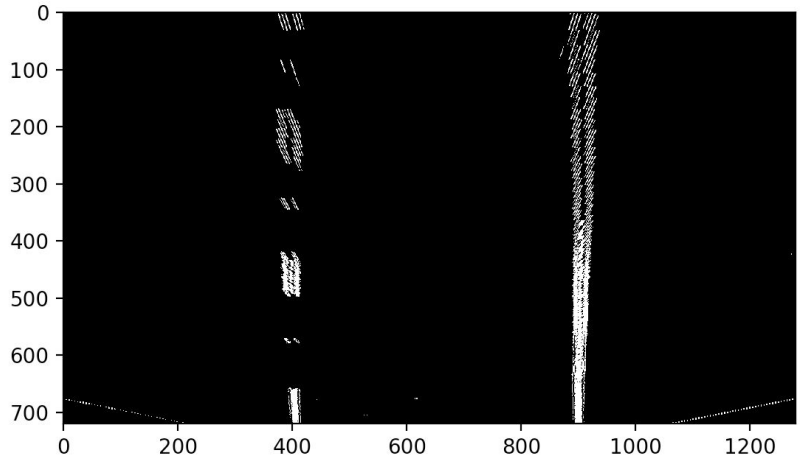
Saturation binary image:



Sobel x gradient binary image:



Combination of saturation and Sobel x gradient:



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

I did a perspective transform to obtain a “top down” view of the lane lines. I performed the perspective transform in lane_finding.py file approximately on lines 328-347 on the original images, before doing any thresholding, which actually proved to be important.

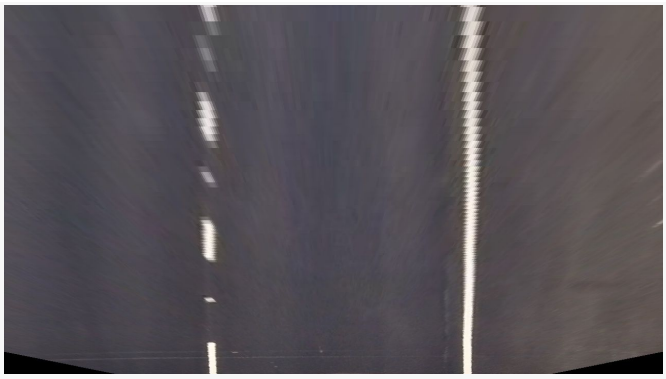
I defined a trapezoid in the original image that encompassed the lane lines with the following values (with some help from the Udacity livestream for this project):

```
bot_width_pct = .76
mid_width_pct = .08
height_pct = .62
bottom_trim_pct = .935
```

I obtained perspective transform matrices through the getPerspectiveTransform() function and used them to project the trapezoid onto a rectangular shape in a new image. For the rectangular destination region, I used offsets to center it in the new image. It went all the way from top to bottom on the y axis and took up 50% in the center of the image on the x axis.

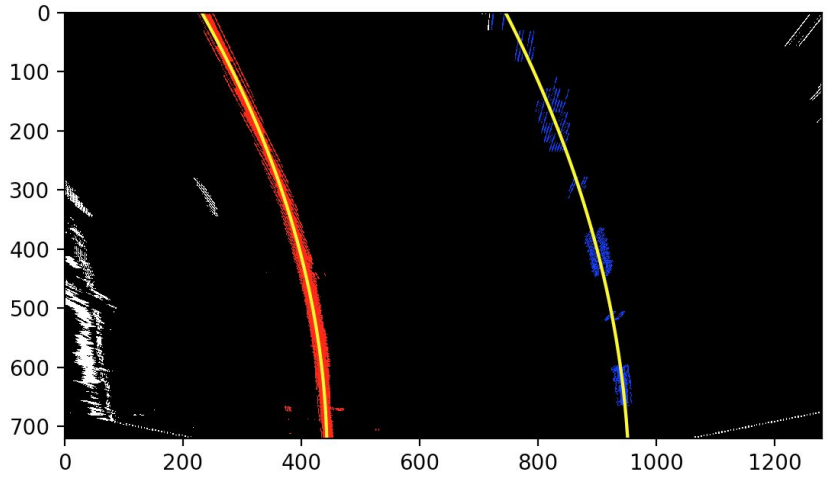
I made all of the transform values proportional to the size of the image so this code could be applied on other image sizes too. I verified visually that the resulting lines after the transform looked parallel.

Here is an example of an image of straight lines and a resulting image after a perspective transform:



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

I used a histogram of pixels to find areas with the most pixels, going by histogram peaks. I used a sliding window search where I searched within 9 windows stacked vertically from top to bottom to find the histogram peak within each window. If there were enough pixels in the window above, I recentered the window which let the search find lines that curved slowly from the bottom of the image to the top. Once I found a cluster of pixels that was likely to represent a lane line, I called a polyfit function to fit a second degree polynomial curve to the centers of all the windows. I then checked if the curves looked parallel to each other and generally reasonable. Here is an example image:



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.

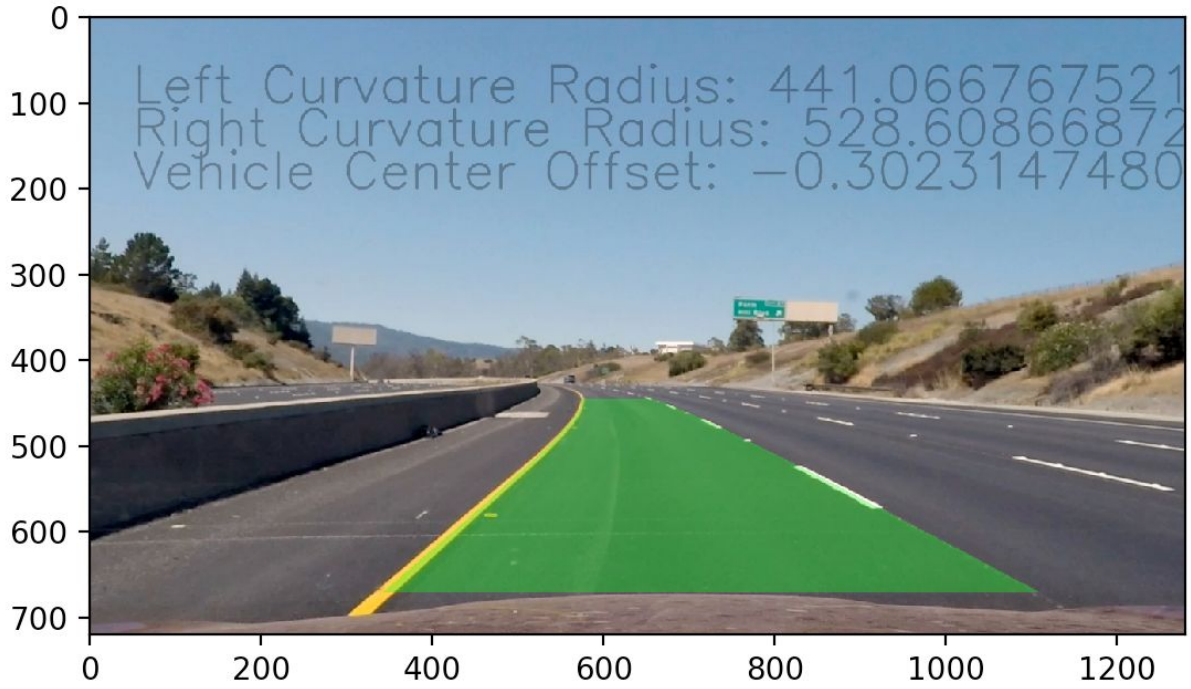
I calculated the radius of the curvature inside my sliding_window() function since I had all the data I needed there, like my polyfit lines. I used the following formula, which was provided in the lecture:

$$R_{curve} = (1 + (2Ay + B)^2)^{3/2} / |2A|$$

I converted the curvature from pixels to meters using the fact that the lane is supposed to be about 30 meters long and 3.7 meters wide.

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

The final image is generated in the draw_road function on lines 245-261. It receives the left and right lane (which are returned by sliding_window function) and warps them back onto the original image, shading the space between them. It also receives curvature and offset and prints them on the image. Both the curvature radius and the offset are displayed in meters.



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!).

Here's a link to my video result

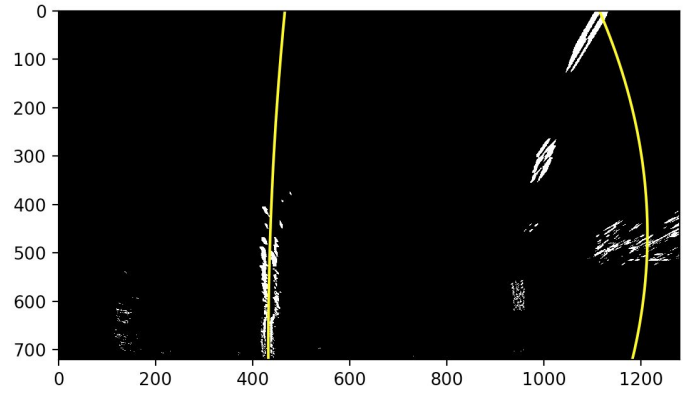
(Since this document will be in PDF, here is the link address:

https://github.com/rizzles38/CarND-Advanced-Lane-Lines/blob/master/project_video.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?

During the implementation of this project, I faced a number of issues. The first issue I ran into was when I was drawing polyfit lines through my transposed binary images. For some reason the binary images ended up with a lot of garbage and the poly lines were all over the place. Here is an example:



I played around with different sobel thresholds and color spaces but in the end, I still either didn't have enough points to detect the curve or had too much garbage on the sides that the histogram would pick up on. Then one of my classmates tipped me off to switching the order of the transpose and the color and gradient thresholding. So, I changed my code to do the transform first. This actually made a pretty big difference and most of the frames had very clearly defined curves. I think the reason was that after the transpose, the farther part of the lane had more pixels and they were not as easily lost during the thresholding process.

One thing that made a pretty big difference is changing the flags argument in the warpPerspective function. Instead of INTER_LINEAR, I used INTER_NEAREST, which is a nearest neighbor filtering. It gave the gradients stronger edges because it was able to represent changes between pixels better.

The next problem I ran into was when I was putting the video through the pipeline. While it mostly looked ok, around the shadows and bright sun spots, the lines started looking really crazy. At some points left and right poly lines overlapped and completely went off the screen. I combated that problem with restricting my area of search. I followed that with removing outliers I went through many frames manually and looked at their polyfit coefficients to get an idea of what an "outlier" looks like, since I didn't want to accidentally filter our reasonable curves. I settled on defining an outlier to be anything where the first coefficient is either more than twice and big or less than half as small as the previous frame. Now, removing outliers completely and replacing them with a mean or median seemed like a bad idea, because it the curve of the road changed enough that my polyfit line just barely qualified as an outlier, I would still want to store that change in my buffer. Otherwise, I would just fill the buffer with average values and not pick up on a change (which the real values could keep changing). So, in my code, I ended up

drawing an “average” value but adding the outlier to the buffer anyway, in order to account for the change in case it was important.

After these steps, I still had some jitters and weird curves occasionally so I decided to fix those with smoothing over a number of frames. My final result looks pretty smooth, but there are a couple of spots near the shadows where it’s wobbly. I believe what happens there is there are a few outliers in a row, and since I’m still adding outliers to my buffer, the buffer gets filled with outlier and shifts the average a little. I couldn’t find a good way to smooth that part out but I have a few ideas if I had more time to work on the project. One would be to not add all outliers to the buffer, but only a limited number of them. A different idea is to experiment more with thresholds and color spaces and specifically focus on the frames with trees and shadows. I felt like going that route could be risky since I turned my thresholds to a point where most of the frames look great, but I would try it if I were to revisit the project.