Advanced Lane Finding Project

Here is the link to my project on Github:

https://github.com/victor3105/udacity_advanced_lane_finding

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.

1 Rubric Points

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

1.1 Writeup / README

Provide a Writeup / README that includes all the rubric points and how you addressed each one.

This document is the writeup :)

1.2 Camera Calibration

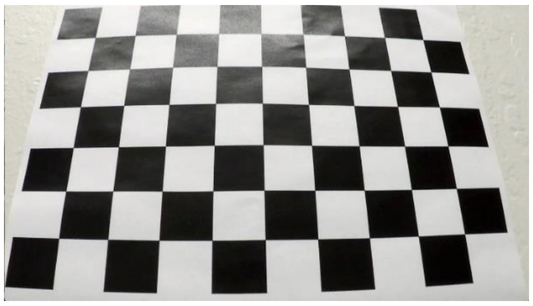
Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The camera calibration code is in lines 8-38 of *main.py* file.

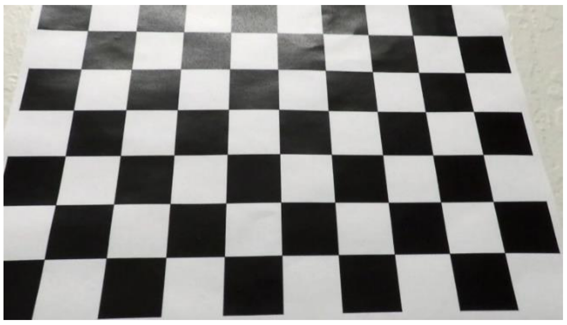
First, we obtain list of calibration images. After that we prepare object points, which are real-world *x, y, z* coordinates of the chessboard corners. Here we assume that the chessboard is flat and $z = 0$.

For each calibration image we convert it to grayscale, after that we find chessboard corners using the OpenCV function. Next, we append pixel corner coordinates in pixels to “imgpoints” list and real-world coordinates of corners to “objpoints”. After that we can use these lists to obtain the camera matrix “mtx”, distortion coefficients “dist”, rotation and translation vectors using the OpenCV function “*calibrateCamera*”.

We can use “mtx” and “dist” to undistort images taken by this camera. Figure 1 contains result of undistortion.



a)



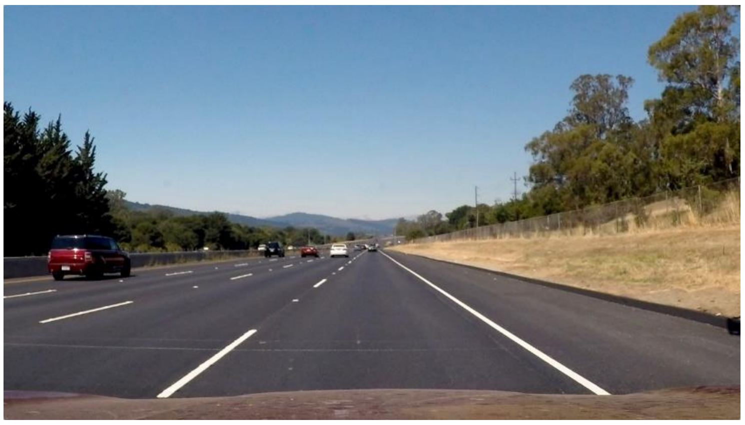
b)

Figure 1 – Original (a) and undistorted (b) images

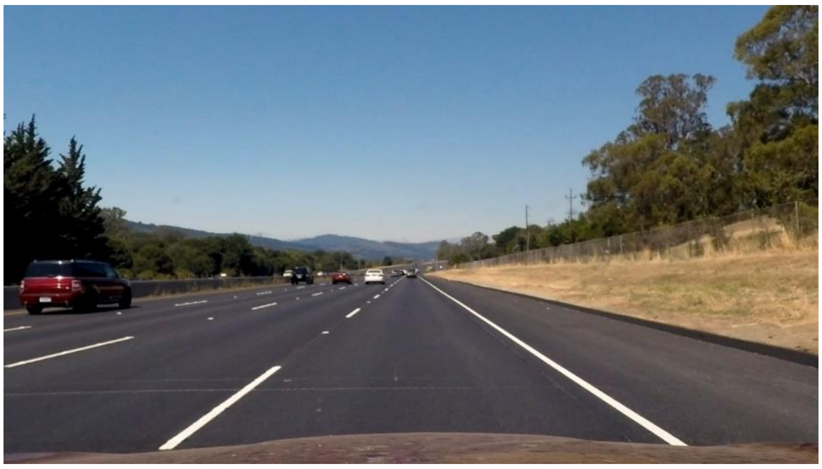
1.3 Pipeline (test images)

1.3.1 Provide an example of a distortion-corrected image

Figure 2 demonstrates original and undistorted test images.



a)



b)

Figure 2 – The original (a) and undistorted (b) images

1.3.2 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

Binarization is done by the function which is in lines 55-84 of *main.py*.

First, we convert image from BGR to HLS. HLS system makes color thresholding much easier. I used saturation channel in my function. Also, I use horizontal Sobel edge detection to binarize image using gradient threshold. Threshold values were picked up experimentally.

After that the results of both methods were combined using logical disjunction. Figure 3 illustrates the result.

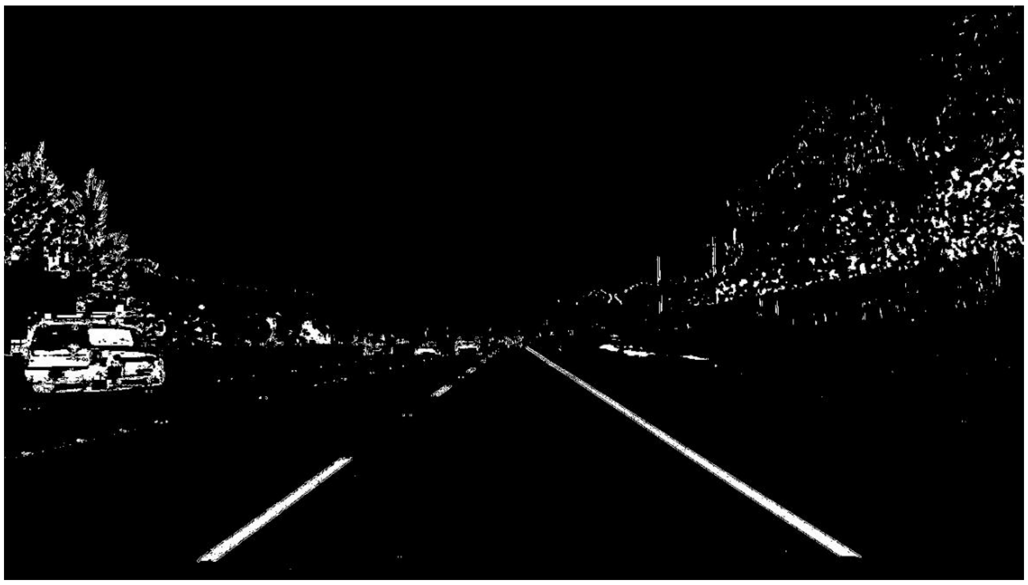


Figure 3 – Thresholded binary image

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

Perspective transform is defined in lines 87-112 in *main.py*.

Source and destination points were defined as

```
src = np.float32([[490, 482], [810, 482],
                  [1250, 720], [40, 720]])
dst = np.float32([[0, 0], [1280, 0],
                  [1250, 720], [40, 720]])
```

Standard OpenCV functions were used. The result is shown in figure 4.

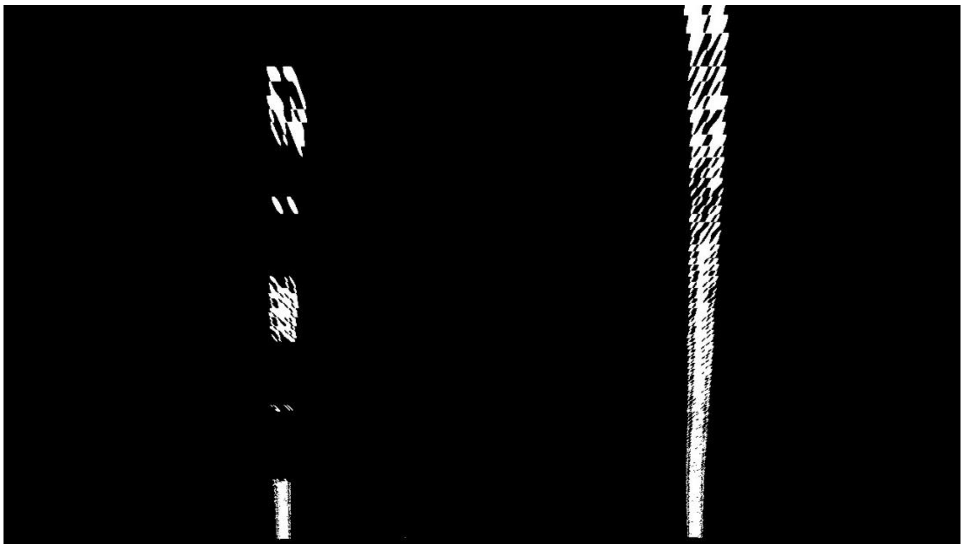


Figure 4 – Warped birds-eye view image

Parallel lines are close to parallel in the warped image.

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

Lane lines are identified by `find_lines` function. It's defined in lines 115-200. The lines search is done by sliding windows method. First, we use histogram across horizontal direction to find intensity peaks. In order to do that, we calculate sum of pixels in each column. Peak horizontal coordinates are the coordinates where we start the search. Then we go from image bottom to the top, recalculating horizontal position of each window. Here we use 9 windows for each line. The function returns *x* and *y* coordinates of left and right line pixels.

After that we fit polynomial curves to the detected lines. This is done by functions `fit_polynomial` and `search_around_poly`. The difference between these two functions is that `fit_polynomial` just uses the output left and right lines pixel positions to calculate polynomial coefficients. After that we calculate *x* position of the fitted curve for each y coordinate (which changes from 0 to the image height – 1).

`search_around_poly` function takes as input previously found polynomial coefficients and looks for line pixels around curves obtained in the previous iteration. After that new polynomials are fitted, and the average polynomial coefficients are calculated. Here we average values over 15 iterations, which makes line fitting smoother.

The results of lines pixels search and curves fitting are shown in figure 5.

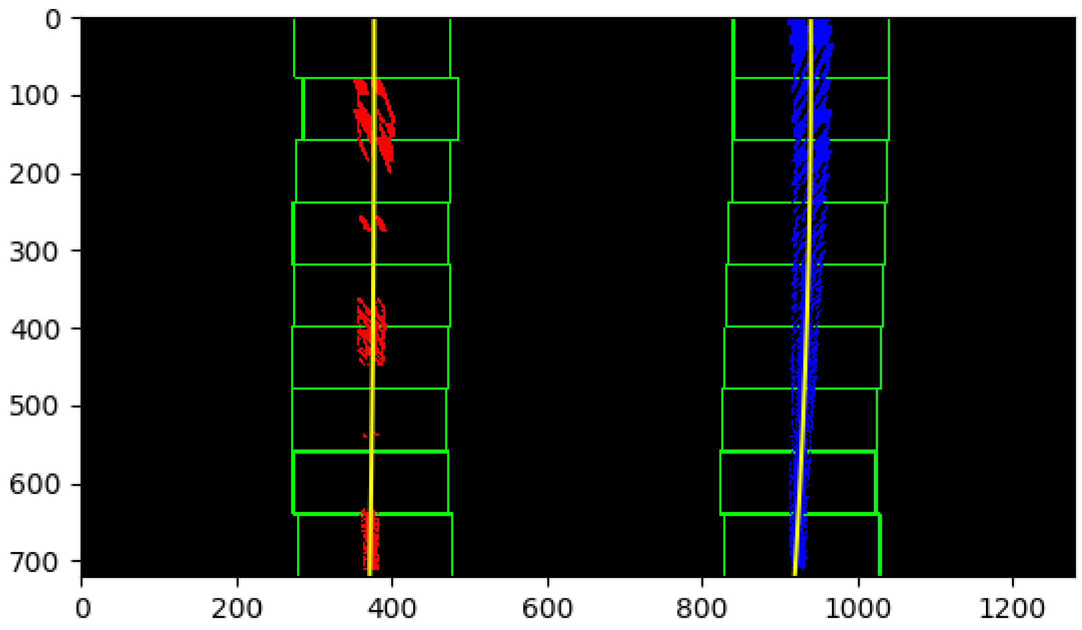


Figure 5 – Lines pixels search and polynomial fitting

1.3.6 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

The radius of curvature and the position of the vehicle with respect to center are calculated by `calc_world_parameters` function (lines 302-361). First, horizontal and vertical scales are calculated. Here we suppose that 720 pixel in vertical direction represent 30 m in real world, and 700 horizontal pixels correspond to 3.7 m.

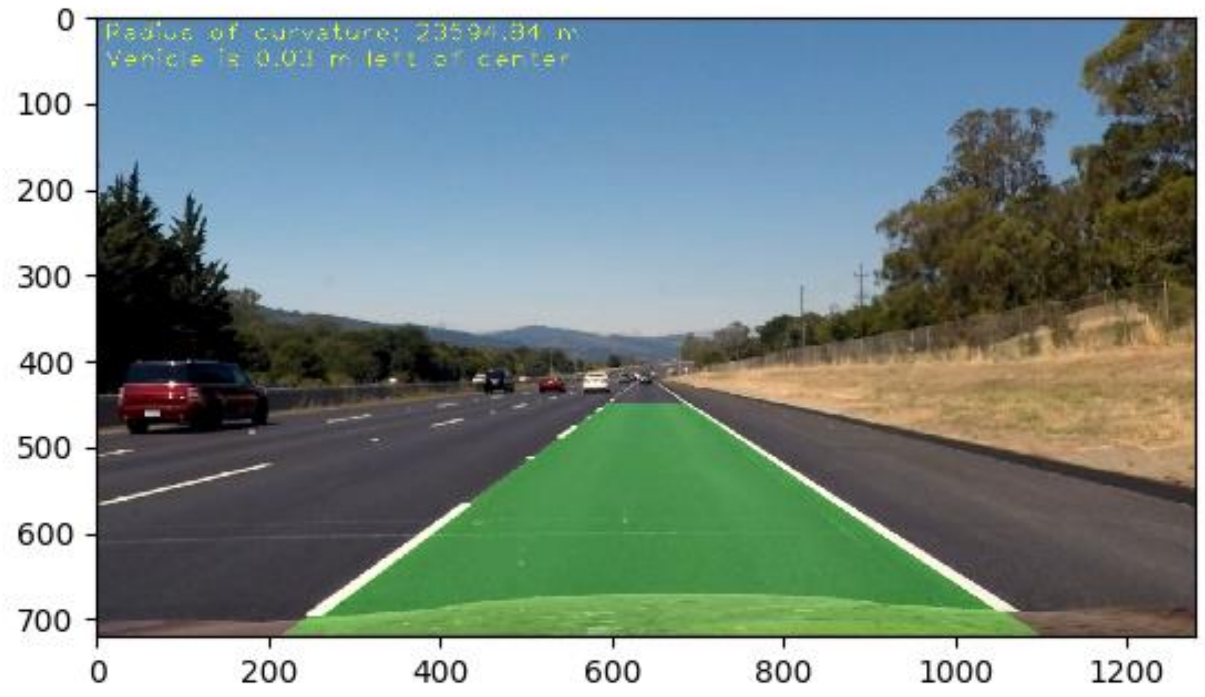
We use formula $R = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$ to calculate radius of curvature, where A and B are polynomial coefficients of the curve. We do that for both lines, after that we find the average radius for them over several iterations. Overall radius in the output image is the average radius of both left and right lines.

To calculate the car position with respect to the lane center we assume that the camera is mounted at the center of the car. Thus, we can calculate the position of the car as the deviation of the image midpoint of the lane from the center of the image.

Both calculated values are converted to meters using appropriate scale values.

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

The output image contains radius of curvature of the lane lines, vehicle position and lane plane (figure 6).


Radius of curvature: 23594.84 m
Vehicle is 0.03 m left of center
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

1.4 Pipeline (video)

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!)

https://drive.google.com/open?id=1_BVINCATf7tNyFSGDEojA0zMUAfn Z90W

2 Discussion

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?

My approach is not robust enough when a car drives through shaded areas. It could be fixed by using more sophisticated binarization method (maybe thresholding with adaptive values depending on average light intensity).

Also, my pipeline would be more robust if I checked line parallelism and monitored the horizontal distance between lines. We can check if the lines search is reliable or not and depending on it reject unreliable results.

One more way to increase accuracy and robustness is to reject outlier pixels.