# Advanced Lane Finding Project - Project Report

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.

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.

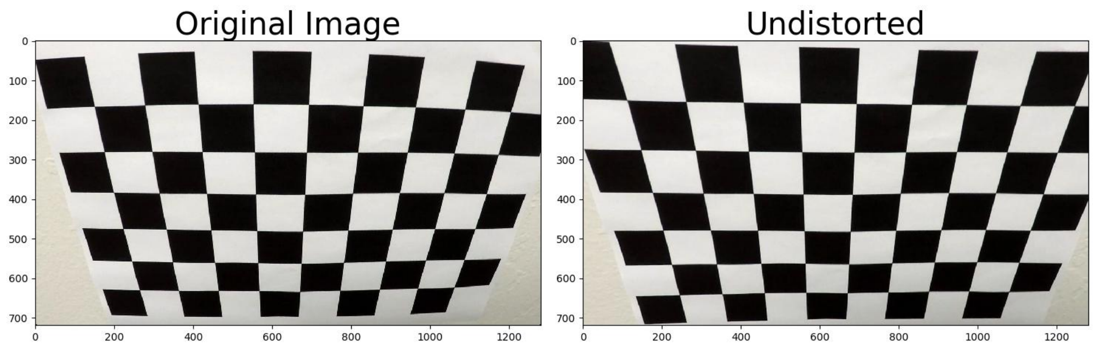
   You're reading it!

## Camera Calibration

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

   First, I prepared the image points and object points in the function readImagePoints (process.py lines 30~59). Preparation of the object points is done by making a grid of the size 9x6 whose content is the (x, y, z) coordinates of the chessboard corners in the world. Assuming that the chessboard is fixed on the z=0 plane, the z coordinates are all set to zero. And the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image.
   Preparation of the image points is done by reading in chessboard images and using the OpenCV function findChessboardCorners to find the image coordinates of the chessboard corners. `imgpoints` will be appended with the found (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output of the above function `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. (process.py line 301) I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

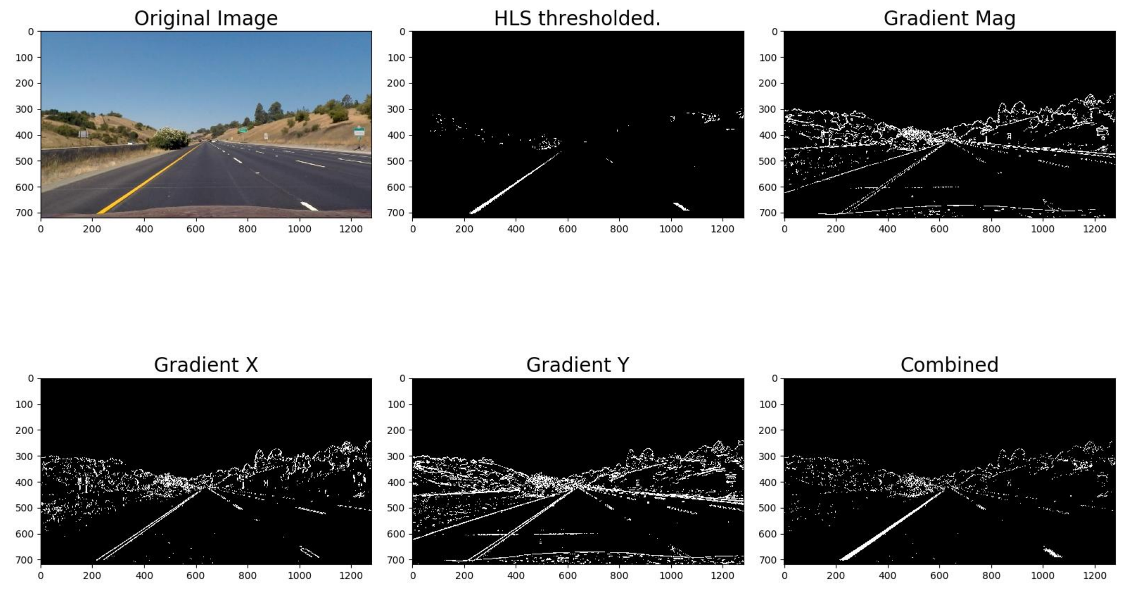


**Pipeline (single images)**

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

4. 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 used a combination of color transform and gradient thresholds to generate a binary image in the function detect_lane_binary (process.py line 160~193). Here's an example of my output for this step.



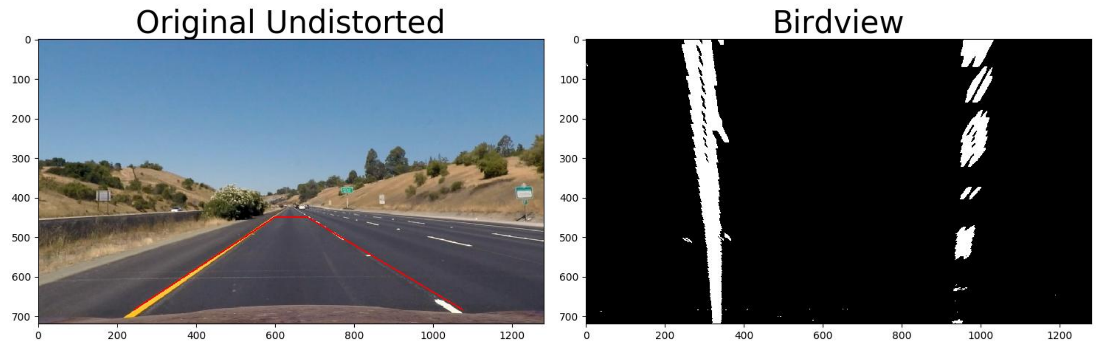
First I calculated the gradient of the image in the x direction, and generated a binary image by thresholding gradient, the result is the Gradient X image above. Then I converted the input image to HLS colorspace and did a threshold on the S channel, the result is the HLS thresholded image above. Finally I combined the above 2 binary image with an OR operation. The result is the Combined image above.

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

First the matrix for the perspective transform is calculated (process.py line 323 ~ 329). I chose to hardcode the source and destination points. The source points were handpicked from the 2 straight line test images. The destination points were set so that the boundary of the source points will appear at approximately 1/4 and 3/4 width of the resulting image.

```
src = np.float32([[242, 685], [599, 451], [681, 451], [1075, 685]])
dst = np.float32([[[(imgShape[0] / 4), imgShape[1]],
            [(imgShape[0] / 4), 0],
            [(imgShape[0] * 3 / 4), 0],
            [(imgShape[0] * 3 / 4), imgShape[1]]]])
```

Then I transform the thresholded binary image using the resulting matrix from previous step in the function DoPerspectiveTransform (process.py line 197~199).



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

I identified the lane-line pixels in process.py lines 341 and fit them to a polynomial in process.py lines 342~349.

First I tried to find the center for the left and right line of the lane in function find_window_centroids. I used the convolution method introduced in class. I found the starting point for my left and right line by summing the bottom 1/4 of the image and finding the maximum position.

Then I worked my way up to the top of the image by processing horizontal slices. In each iteration:

1. I sum up a horizontal slice of the image, whose height corresponds to the window height.
2. I convolve the summing result with my defined kernel, which emphasizes the vertical center of the window.
3. Then I identify the x coordinate of the maximum result on the left and right part of the image individually. This gives the new of the left and right line for this iteration.
4. I did a sanity check on the new center: If the difference between the new center and the center obtained from the last iteration is too large, I discard the new center and used the center from the last iteration instead. This way I can prevent the line from breaking.
5. The resulting left and right centers are appended to the list that will be returned when the function terminates.

To fit the left lane to a polynomial, I retrieved the left centers, which are pixel coordinates, and translated them into units of meters. Then I used the function np.polyfit, along with generated y coordinates to obtain the fitted polynomial coefficients. The same is also done for the right lane.

## 7. 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 curvature of the lane in the function calculateCurvature (process.py lines 258~261) and calculated the position of the vehicle in process.py lines 434~437.

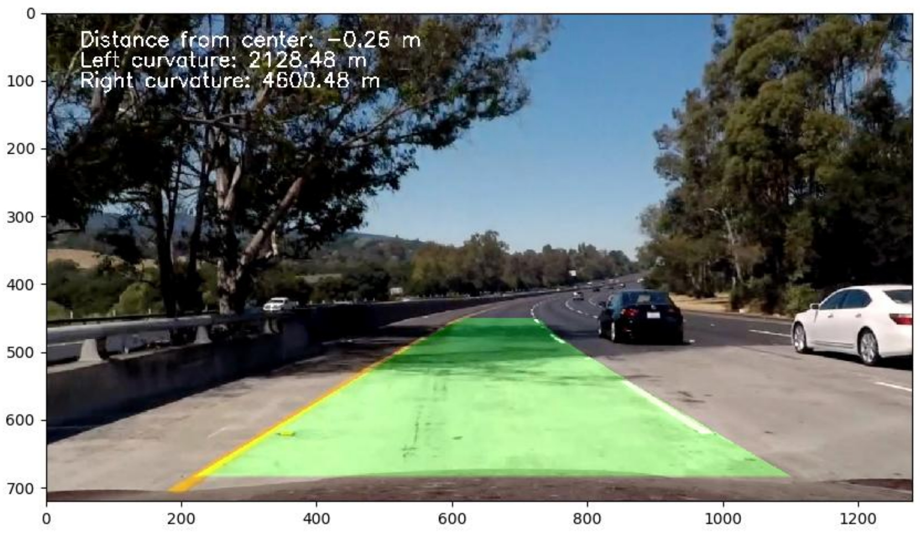
The curvatures of the left and and right lane are calculated using the below formula.

$$R = \frac{(1+(2*\text{Coeff }[0]*y+\text{Coeff }[1])^2)^{3/2}}{|2A|}$$

The distance to center is calculated by getting the difference between the x coordinate of the middle of the image and the x coordinate of the middle of the two lane lines.

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

I implemented this step in process.py line 413~443 in my code in file process.py. Here is an example of my result on a test image:


Distance from center: -0.26 m
Left curvature: 2128.48 m
Right curvature: 4600.48 m
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

**Pipeline (video)**

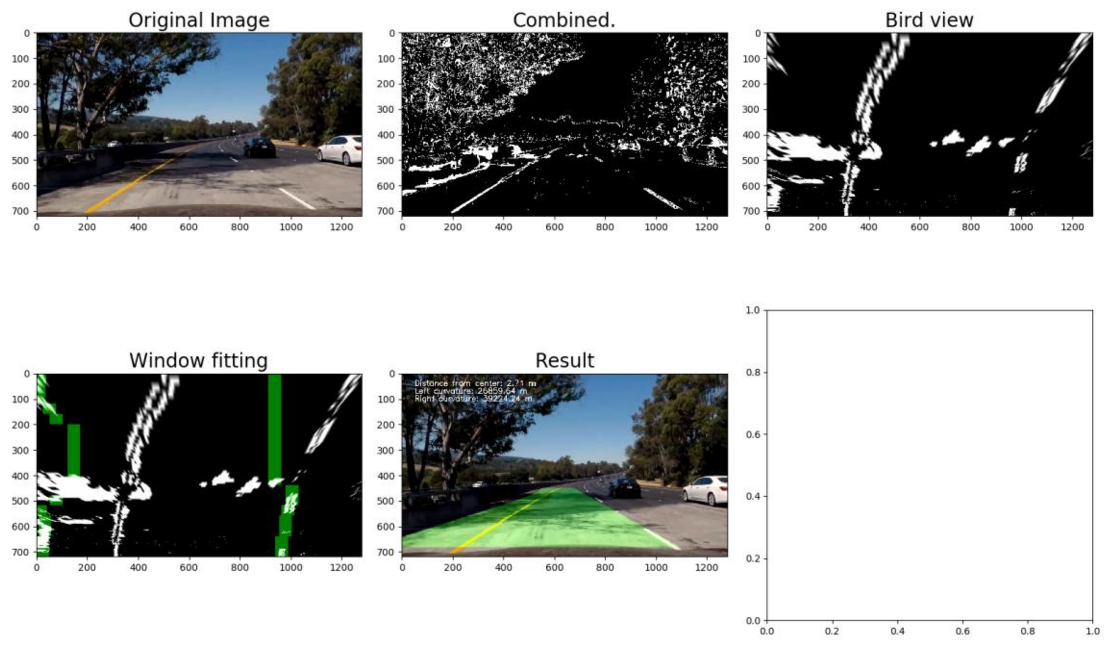
9. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's the link to the resulting video:
https://github.com/EllinaLu/CarND_AdvancedLaneLines/blob/master/project_result.mp4

**Discussion**

10. 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?

Originally, my pipeline failed at frame 1037 using the project_video.mp4. The left line of the lane was not detected, due to more positive results in the binary thresholded image. However, I haven't found a way to resolve this issue from the start where the detection took place. But the detection was correct in frames before 1037. So I added a history component and filtered the starting position of the line if it differs significantly from the starting position of the previous frame.



I also filtered the finishing position of the line in bird's view if it differs significantly from the finishing position of the previous frame.

However, there are two places in the video where there weren't enough pixels at the base of the right line to properly detect it. This resulted in a small curvature been calculated and the overlayed polygon incorrect. So I averaged the polynomial coefficients over previous 5 frames to get a good enough estimate of the coefficients for the next frame. Then I checked the curvature calculated, if the curvature is too small to be valid, I skip the coefficients calculated for this frame, and used the averaged estimate from the previous frames instead.