Advanced Lane Finding

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.

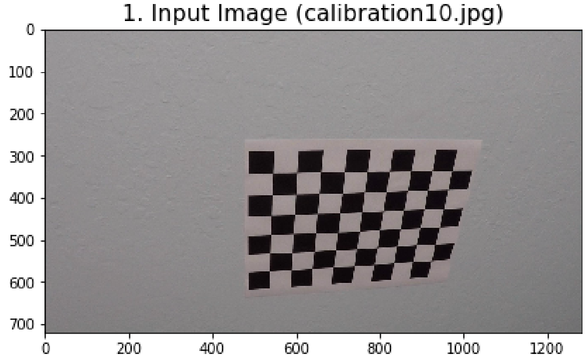
Camera Calibration

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

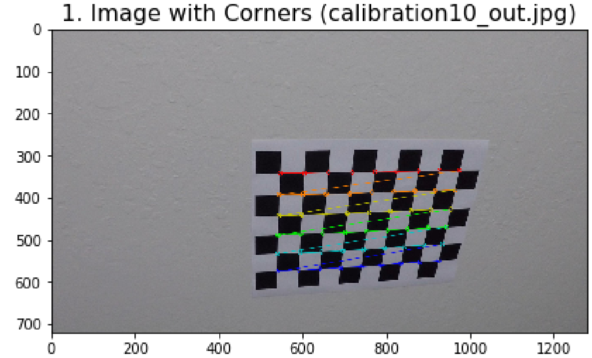
I used the OpenCV functions findChessboardCorners and drawChessboardCorners to identify the locations of corners on a chessboard photos in camera_cal folder taken from different angles.

I then used the output points_3d and points_2d to compute the camera calibration and distortion coefficients using the cv2.calibrateCamera() function. I applied this distortion correction to the test image using the cv2.undistort() function and obtained this result:

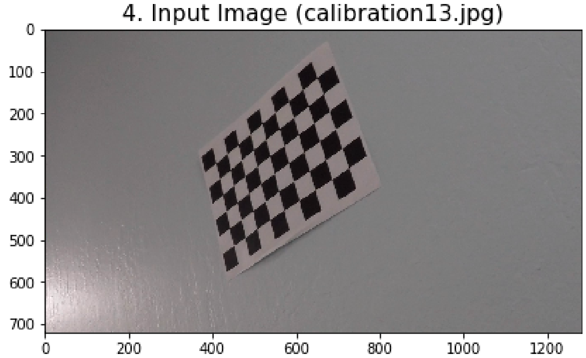
1. Input Image (calibration10.jpg)



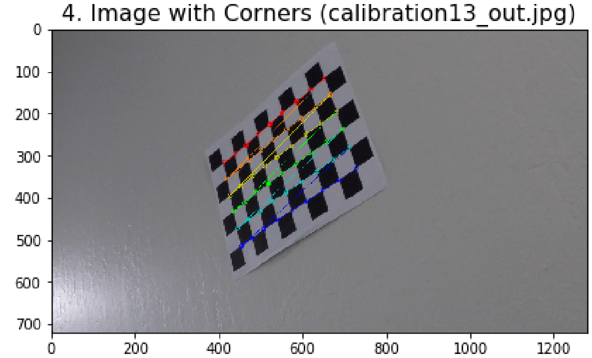
1. Image with Corners (calibration10_out.jpg)



4. Input Image (calibration13.jpg)



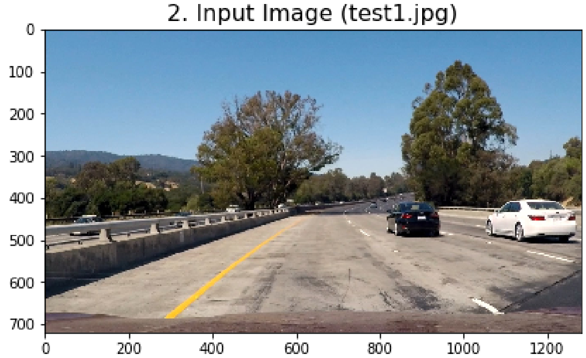
4. Image with Corners (calibration13_out.jpg)



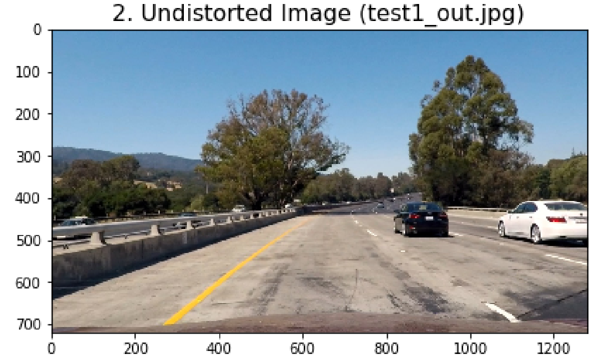
Pipeline

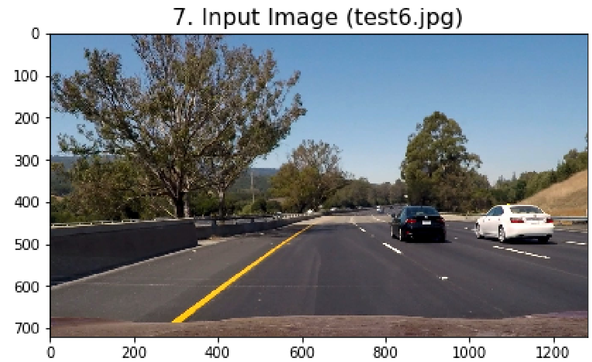
1. Applied distortion correction on the images provided using calculated camera calibration matrix and distortion coefficients. Code provided in cell [4].

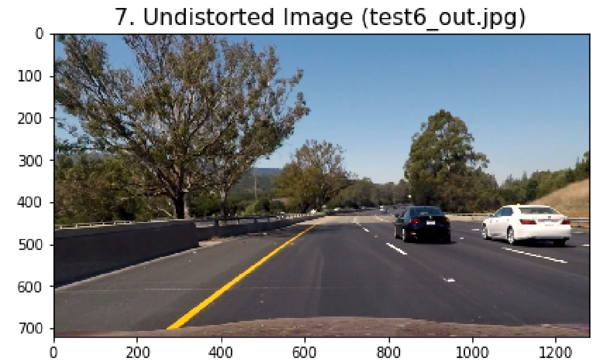
2. Input Image (test1.jpg)



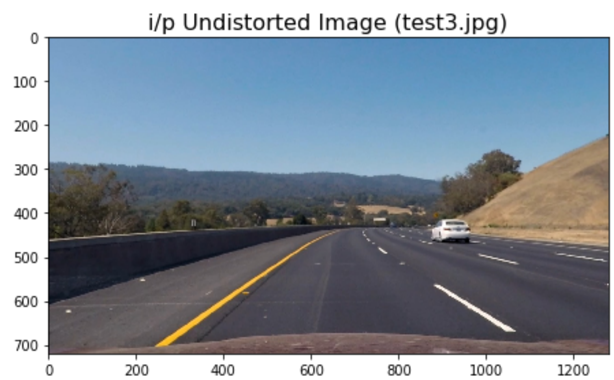
2. Undistorted Image (test1_out.jpg)

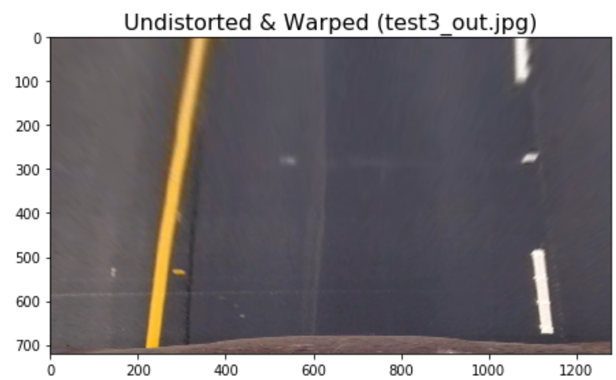


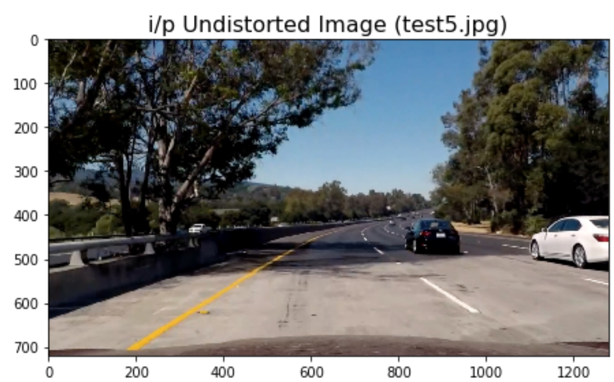


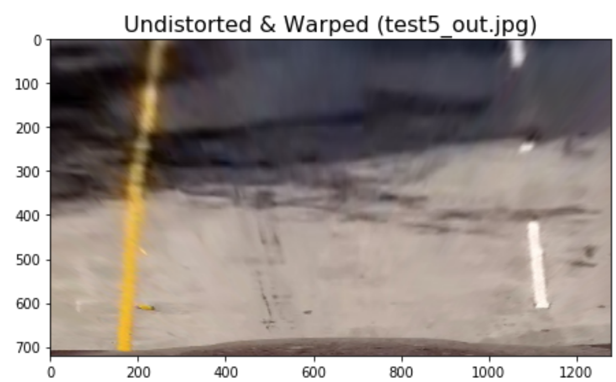


2. Apply a perspective transform to the image ("birds-eye view").
   Code is provided on cell [5] on function called prespective_transform.
   It uses the CV2's **getPerspective**Transform and **warpPerspective** fns and **remove_distortion** written as discussed.





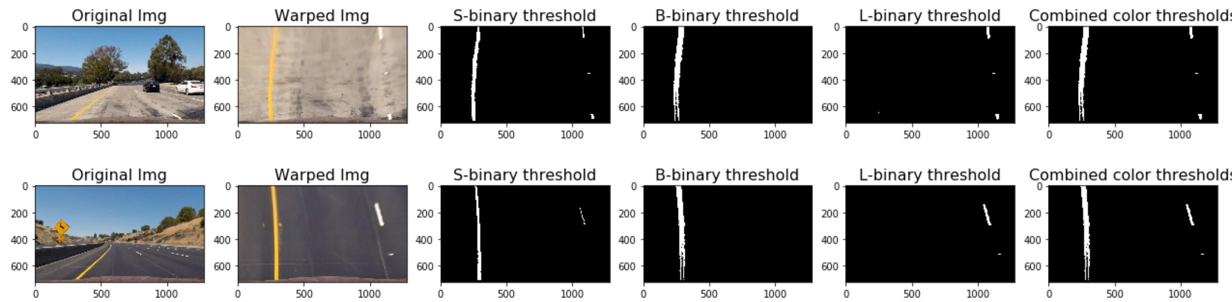




3. Use color transforms, gradients, etc., to create a thresholded binary image. To find the following channels and return them and combined image:
   a. S-Channel
   b. B-Channel
   c. L-Channel

   Code is provided on cell [6]. I used the color combination and gradient to get the binary image.

   Wrapped image is converted to another color space and generated the binary image to be able to highlight the lane lines only and ignore others

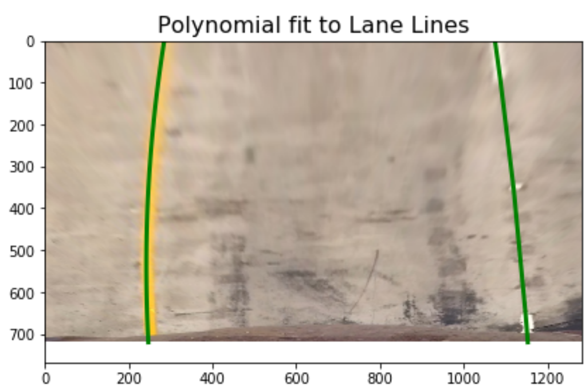


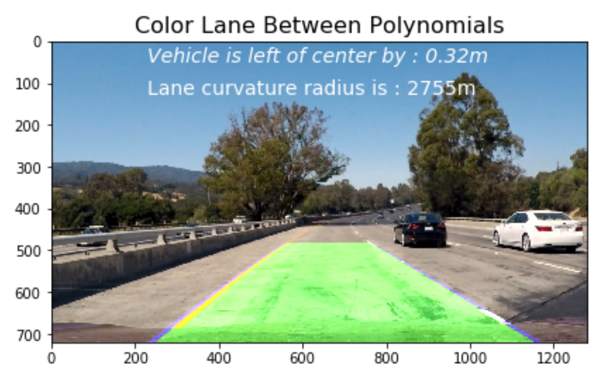
4. Fitting a polynomial to the lane lines and fill the space between them
   Code is provided in cell [17].
   I have used the combined binary image to fit the polynomial to each lane line, as follows:

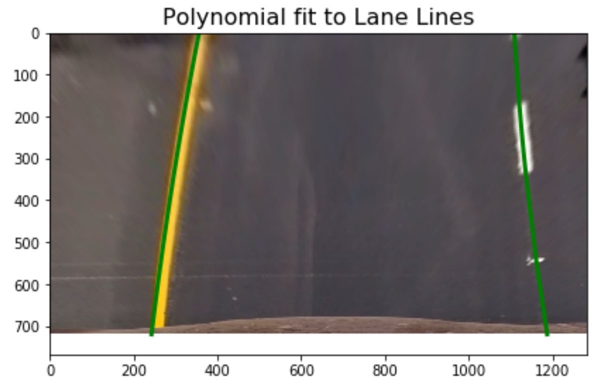
- Determine the location of lane lines using the peaks of image histogram.
- Identified all non-zero pixels around histogram peaks using numpy.nonzero().
- Fitted polynomial to each lane using numpy.polyfit().

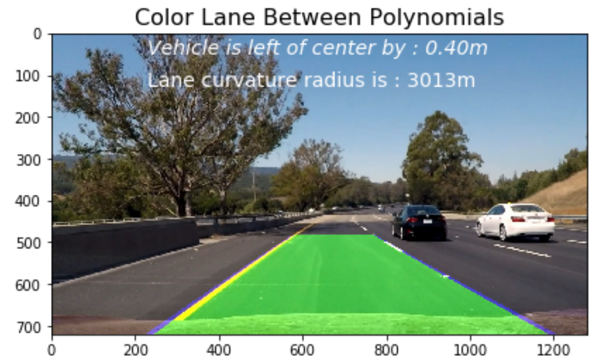
With this, I was able to calculate the position of the vehicle w.r.t center with the following calculations:

- Calculated x intercepts avg. from each of the two polynomials position
- Calculated distance from center by taking the abs value of the vehicle's position and subtracting the halfway point along the horizontal axis distance from center.
- If horizontal position of the car was > image_width/2, then car was considered to be on left of center, else right of center.
- Finally, the center distance was converted from pixels to meters by multiplying the number of pixels by 3.7/730.









5. Calculating raduis of curvature
   Code is provided in cell [19] funtion calculate_radius_of_curvature.

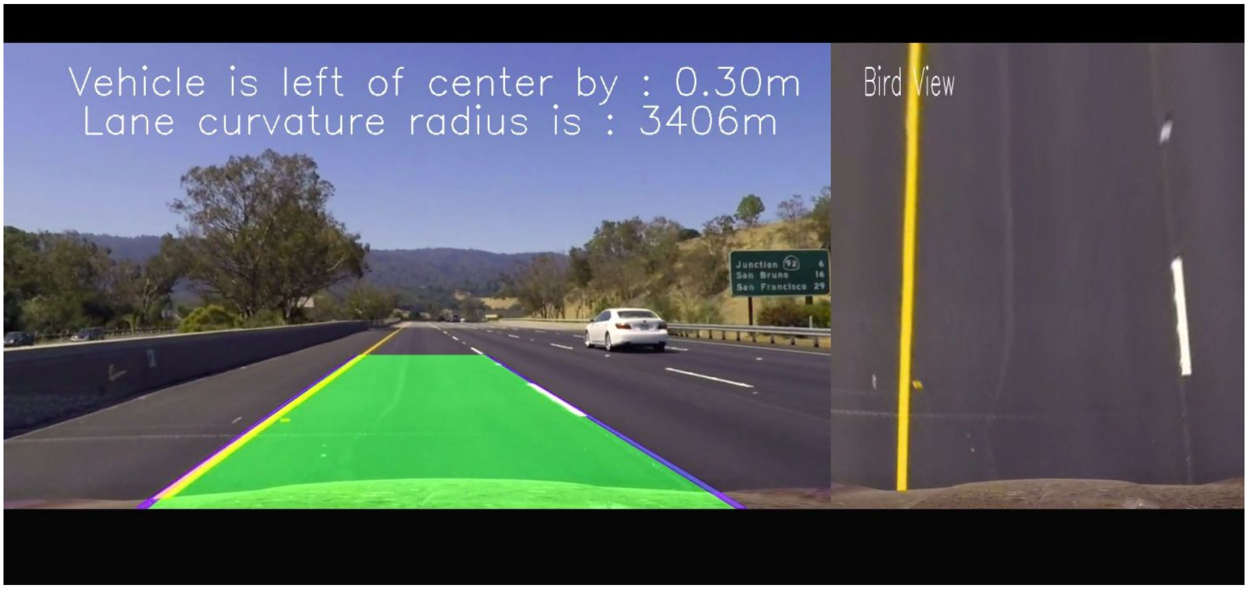
The curvature is calculated using the following code:

```
# Find radius of curvature for both lane line
    xm_per_pix = 3.7/730 # meteres/pixel in x dimension
    ym_per_pix = 23.0/720 # meters/pixel in y dimension

    left_lane_fit_curvature = np.polyfit(left_y*ym_per_pix, left_x*xm_per_pix, 2)
    right_lane_fit_curvature = np.polyfit(right_y*ym_per_pix, right_x*xm_per_pix, 2)
    radius_left_curve = ((1 + (2*left_lane_fit_curvature[0]*np.max(left_y)*ym_per_pix +
left_lane_fit_curvature[1])**2)**1.5)/np.absolute(2*left_lane_fit_curvature[0])
    radius_right_curve = ((1 + (2*right_lane_fit_curvature[0]*np.max(left_y)*ym_per_pix +
right_lane_fit_curvature[1])**2)**1.5)/np.absolute(2*right_lane_fit_curvature[0])
```

Final output

Final output videos are in the output_video folder, and below sample of the output video.



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?**

   The problems I encountered were almost exclusively due to lighting conditions, shadows, discoloration, etc. It wasn't difficult to dial in threshold parameters to get the pipeline to perform well on the original project video (particularly after discovering the B channel of the LAB colorspace, which isolates the yellow lines very well), even on the lighter-gray bridge sections that comprised the most difficult sections of the video. It was trying to extend the same pipeline to the challenge video that presented the greatest (ahem) challenge. The lane lines don't necessarily occupy the same pixel value (speaking of the L channel of the HLS color space) range on this video that they occupy on the first video, so the normalization/scaling technique helped here quite a bit, although it also tended to create problems (large noisy areas activated in the binary image) when the white lines didn't contrast with the rest of the image enough. This would definitely be an issue in snow or in a situation where, for example, a bright white car were driving among dull white lane lines. Producing a pipeline from which lane lines can reliably be identified was of utmost importance (garbage in, garbage out - as they say), but smoothing the video output by averaging the last n found good fits also helped. My approach also invalidates fits if the left and right base points aren't a certain distance apart (within some tolerance) under the assumption that the lane width will remain relatively constant.
   I've considered a few possible approaches for making my algorithm more robust. These include more dynamic thresholding (perhaps considering separate threshold parameters for

different horizontal slices of the image, or dynamically selecting threshold parameters based on the resulting number of activated pixels), designating a confidence level for fits and rejecting new fits that deviate beyond a certain amount (this is already implemented in a relatively unsophisticated way) or rejecting the right fit (for example) if the confidence in the left fit is high and right fit deviates too much (enforcing roughly parallel fits). I hope to revisit some of these strategies in the future.