Advanced Lane Finding Project

The goals / steps of this project are the following:

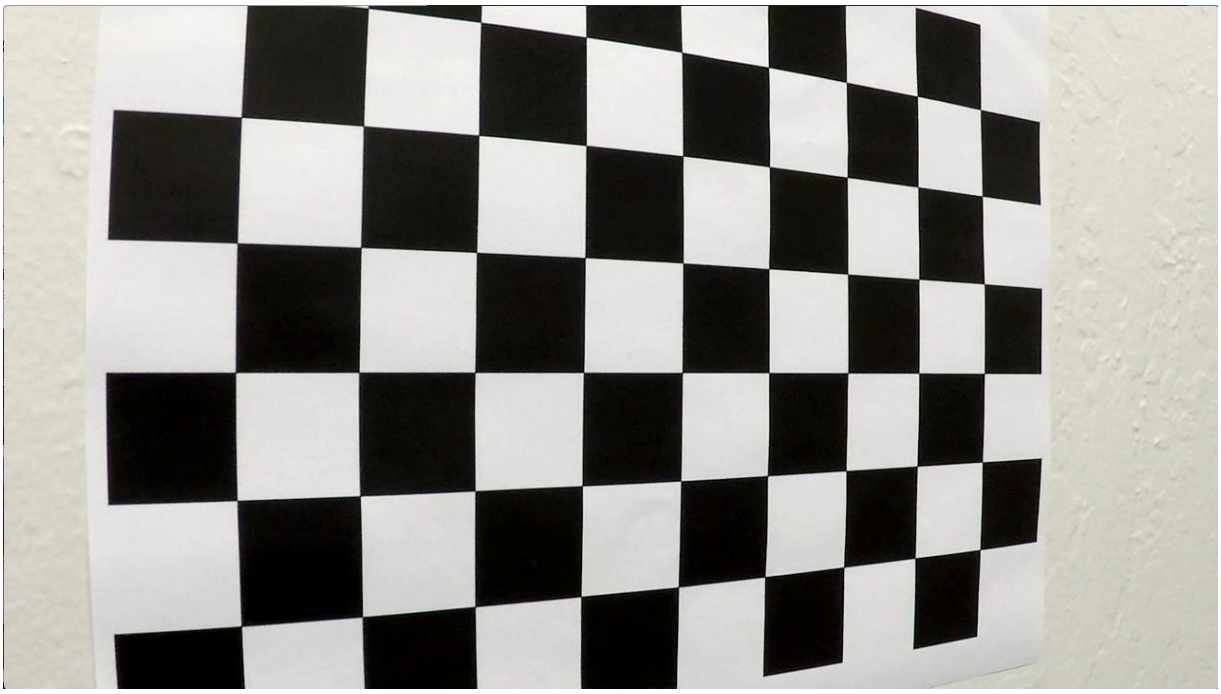
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.

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

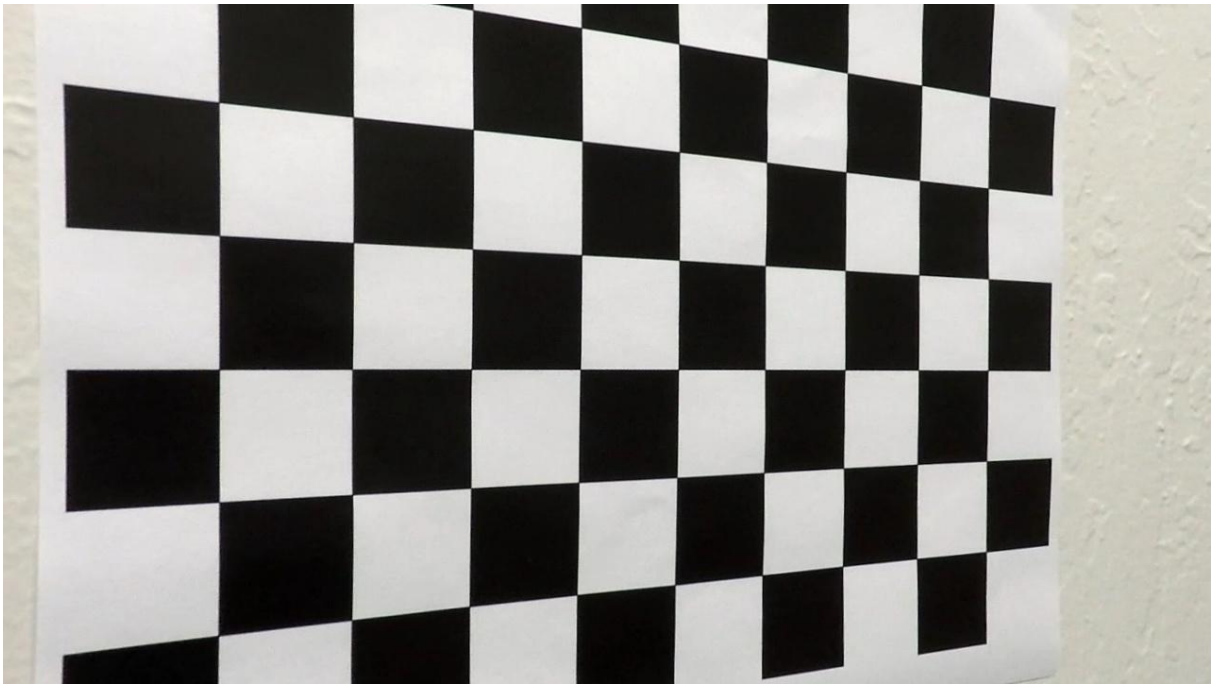
####1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in file “Project 4 - Advanced Line Detection_Create Video.py” from line number 23 to 78.
I started by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the real world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that 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. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. These are 2d points in image plane.
I read all the test images using glob. I then used the output `objpoints` and `imgpoints` 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 the below result. I saved the result as “wide_dist_pickle.p” pickle file to be reused for lane detection images.

Original Image:

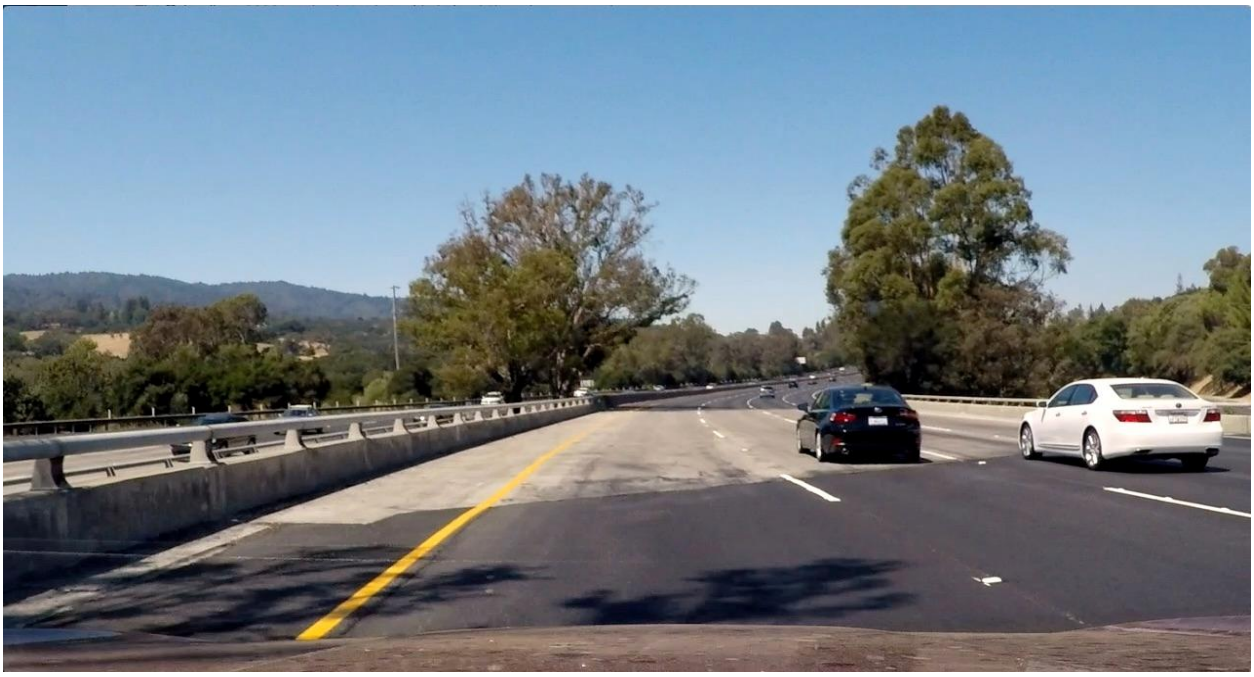


Corrected Image:

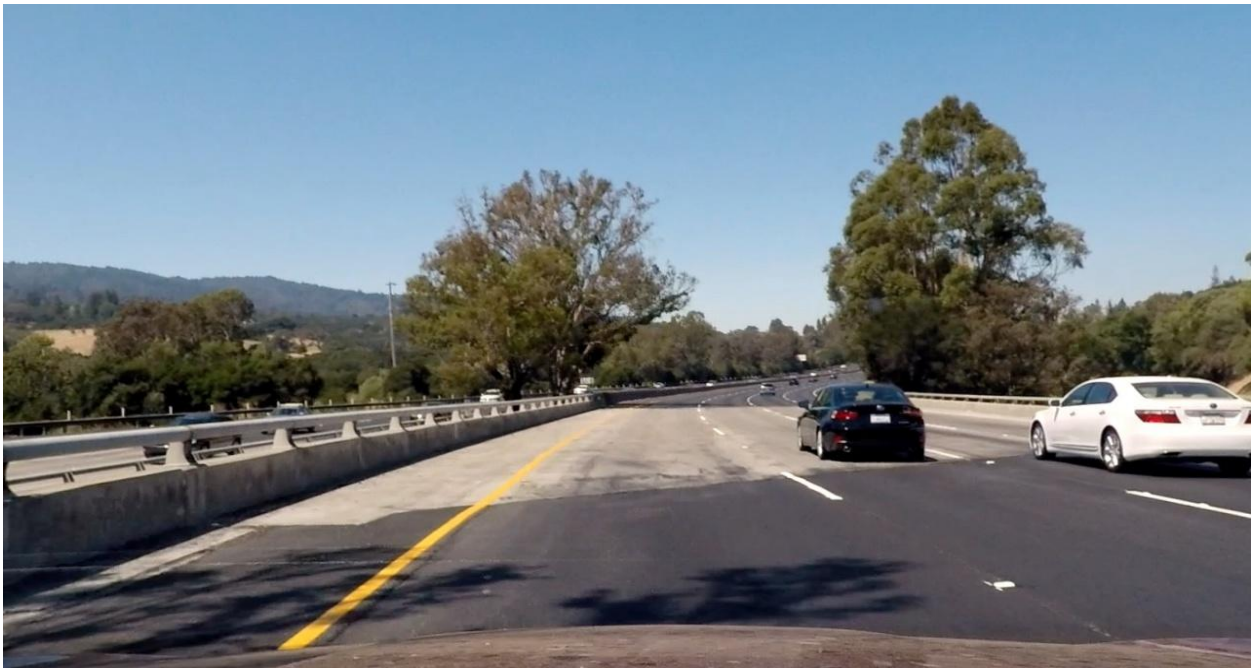


####1. Provide an example of a distortion-corrected image.
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

Original Image:



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 used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines #455 through #465 in attached code file, Also functions are listed from line #164 to #200). I used combination as below:

combined[((mag_binary == 1) & (dir_binary == 1)) | (hls_binary == 1)] = 1

Here's an example of my output for this step for the same original image above.

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

The code for my perspective transform includes a function called `corners_unwarp()`, which appears in lines 89 through 120 in the attached code file.
The `corners_unwarp()` function takes as input an image (`img`). I chose to hardcode the source and destination points in the following manner:

Source	Destination
[550, 480]	[330, 350]
[220, 700]	[330, 700]
[1080, 700]	[970, 700]
[800, 500]	[1090, 350]

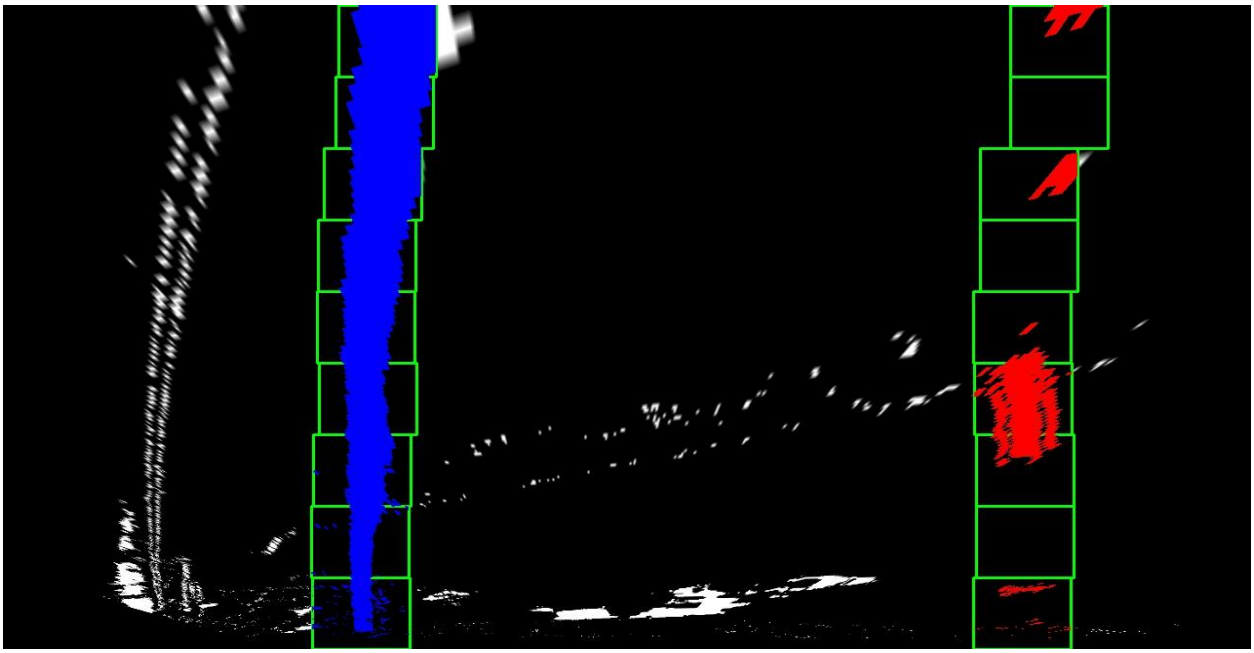
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image. Below is the warped image of the same above test example.

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

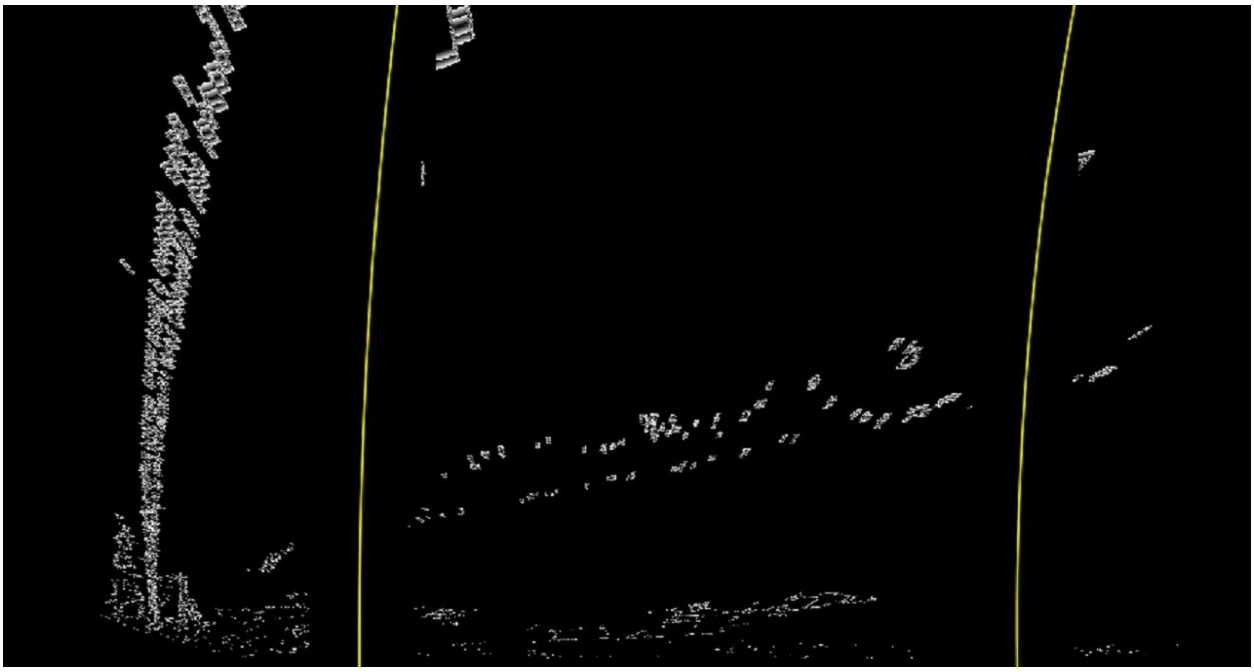
Code in the attached file from line 496 to 499 calls the functions slidinig_window() or skilp_slidinig_window() to identify lane line pixels and fit the polynomial. The functions are written from line # 202 to #361. I identified the peaks of the left and right halves in the histogram of the bottom half of the image. They served as the starting point for the

left and right lines for the first frame. Then I defined the sliding window and identified the x and y positions of all nonzero pixels in the image. I extracted left and right line pixel positions and fitted a second order polynomial to each. For subsequent frames, I used the left and right fit from previous frame as a starting point.

Sliding Windows:



Fitted line:

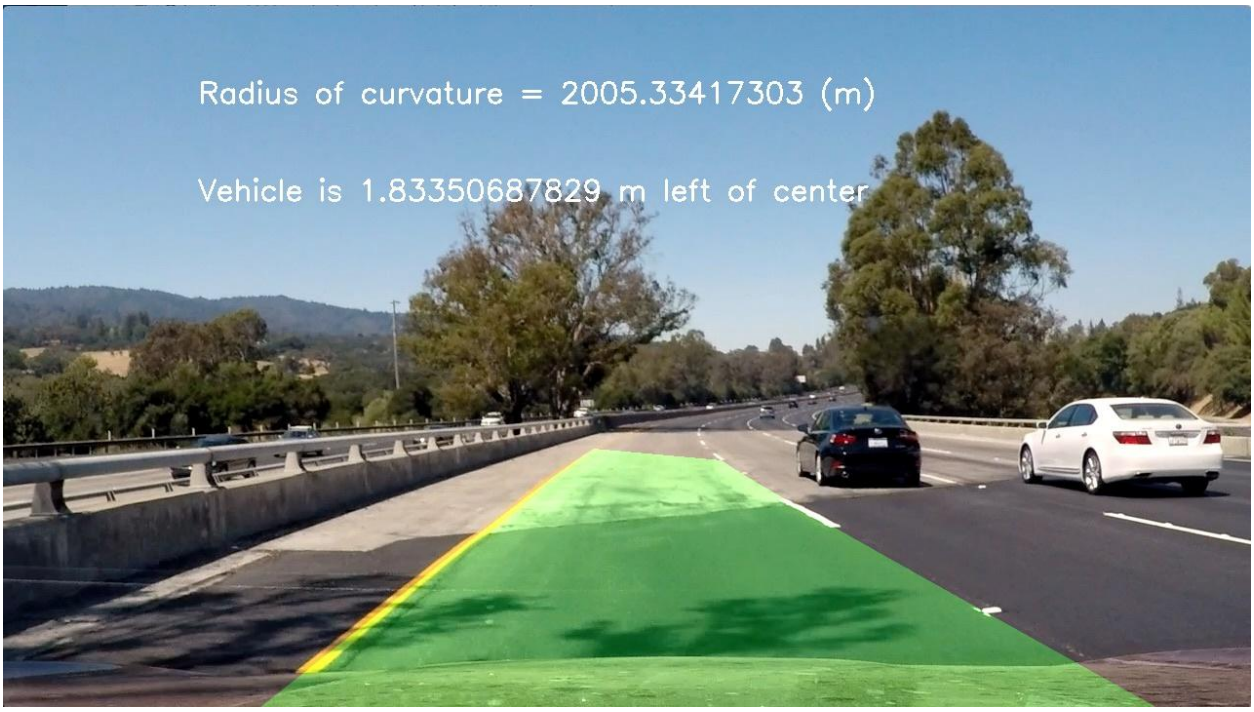


####5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in lines #364 through #389 in my code in function curvature().The results are printed on each frame in the output video.

####6. 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 lines #391 through #415 in my code in function draw_on_original(). Here is an example of my result on a test image:



###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!).

Video file is attached to the submission. Also find the video on link https://youtu.be/SwzzRVgK62k

---

###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 my implementation I faced an issue where due to sunlight, left lane detection was getting wrong. The lane was identified wider than the current lane. By adjusting the thresholding and increasing the range for color thresholding (Saturation), algorithm started detecting the lane lines even with extra light or shadow effects.

The other problem was that the lane polynomial was fluctuating for few frames giving the flash effect. This problem was resolved when I started using the fit from previous frame for next frame as starting point.

The pipeline may fail for extreme brightness situations or when curves are too sharp. I will work on the thresholding parameters and combinations to make it more robust.