Aidos Utegulov

Project 4 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.

Camera Calibration

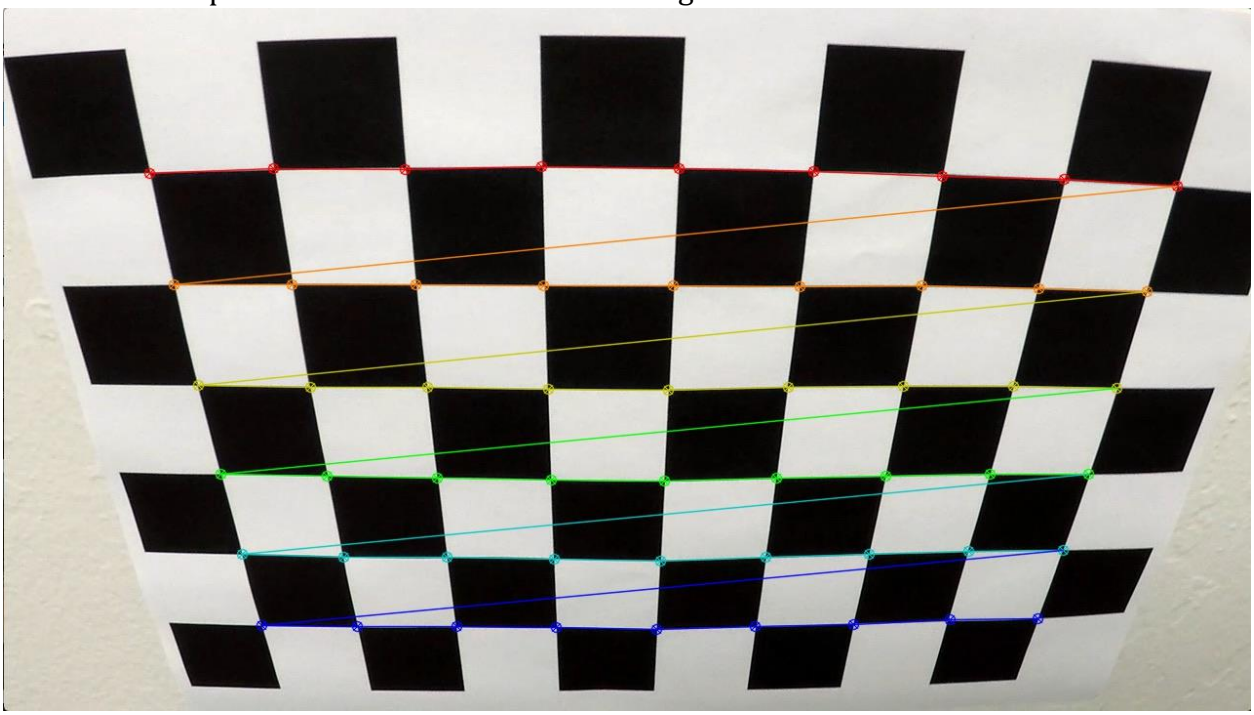
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the first code cell of the IPython notebook named P4-Advanced-Lane-Lines (simply notebook hereafter).

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the 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.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I then pickled them in a file called "calibration_pickle.p" (this is all done in the first code line of the notebook). Here

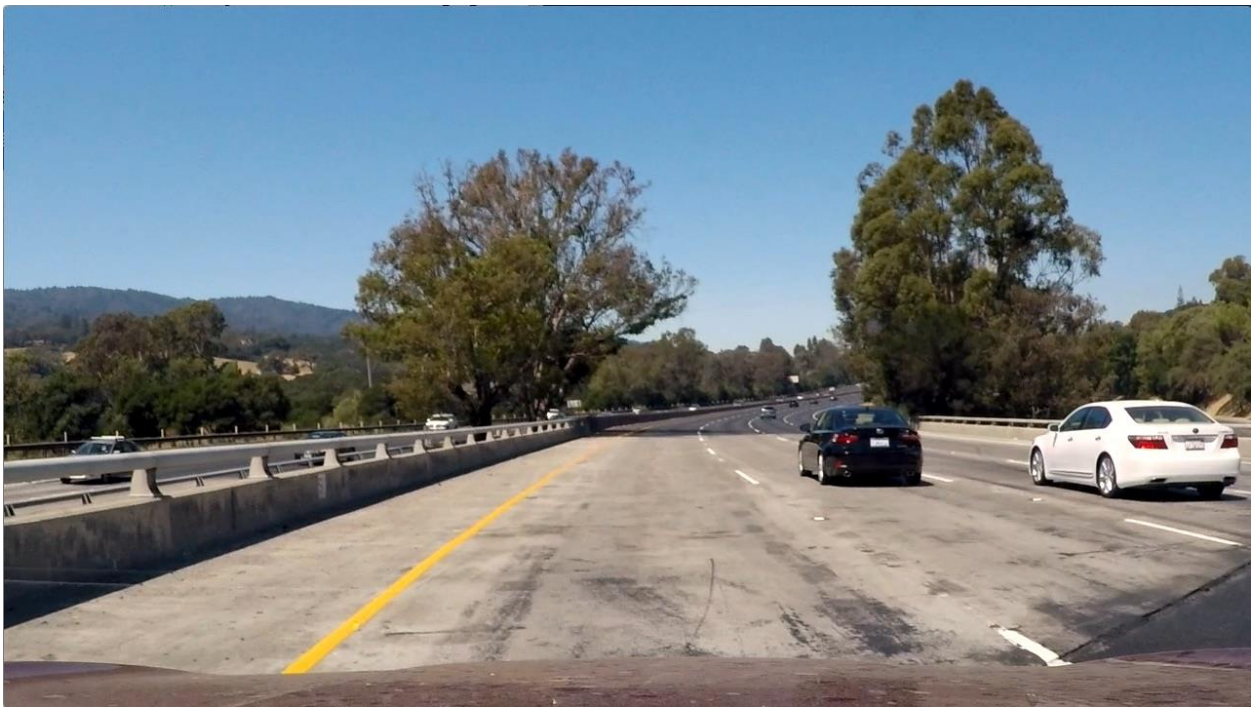
are some examples of undistorted calibration images:



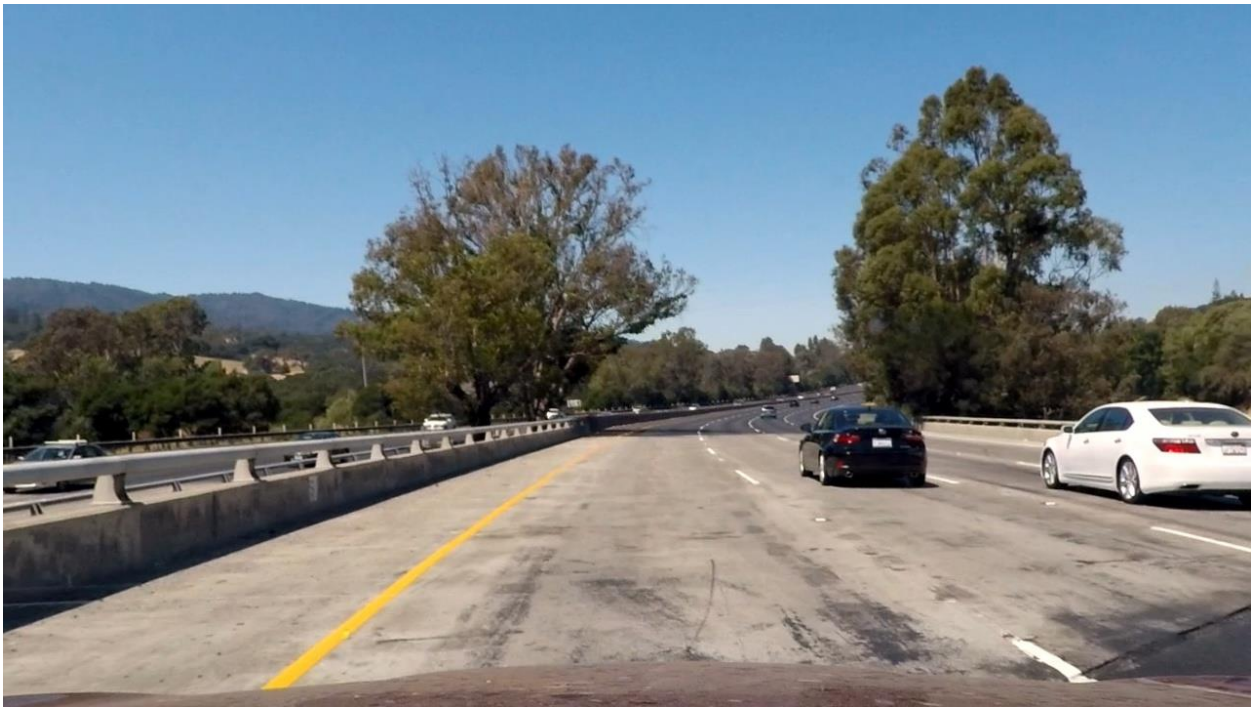
Pipeline (single images)

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:

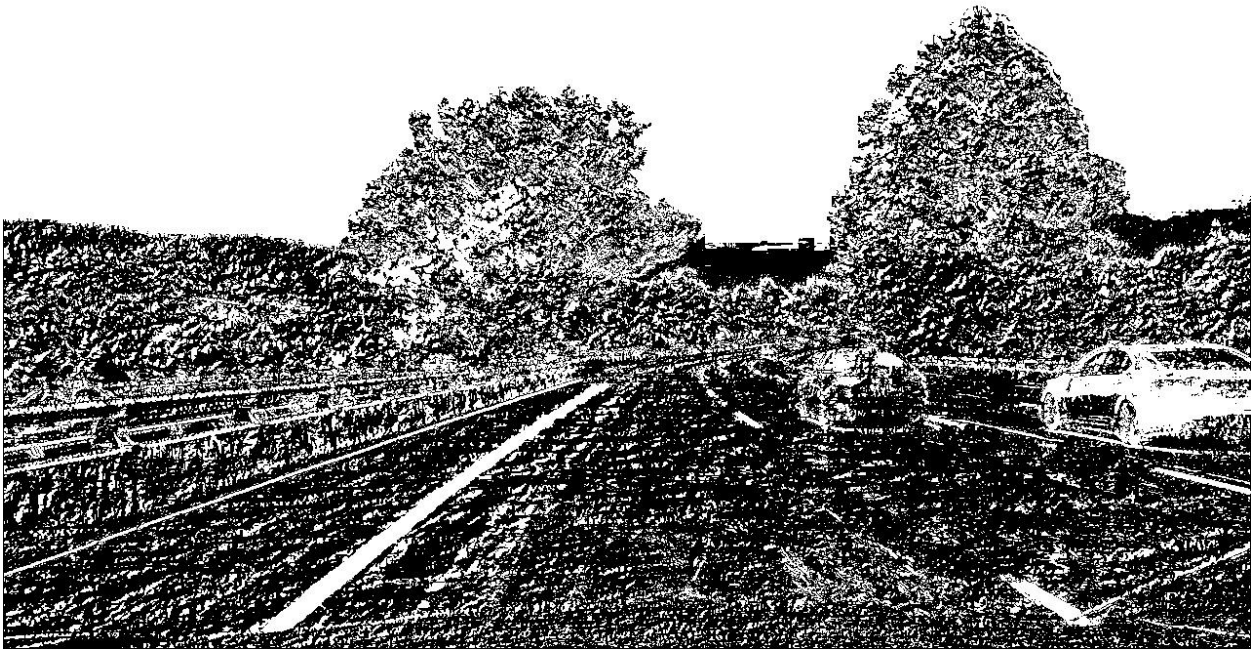


After distortion correction the image looks like this (Although this is a barely noticeable distortion correction, you still can observe that the white car is now much closer to the right end of the image):



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 second and third code cells of the notebook, functions of the pipeline: abs_sobel_threshold() with orient = 'x', and 'y', and color_threshold). For color thresholding I converted to HLS and HSV color spaces and masked the s and v channels in the resulting image. I also added the yellow white, white_2 and white_3 channels for hsv and hls images. Gradient thresholding is done with the help of cv2.Sobel() function. The combination of gradient and color thresholding is performed in the 'threshold_img()' function in the third code cell of the notebook ('In [12]') I found that x and y gradients in combination with the color thresholds work best. Here's an example of my output for this step.



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

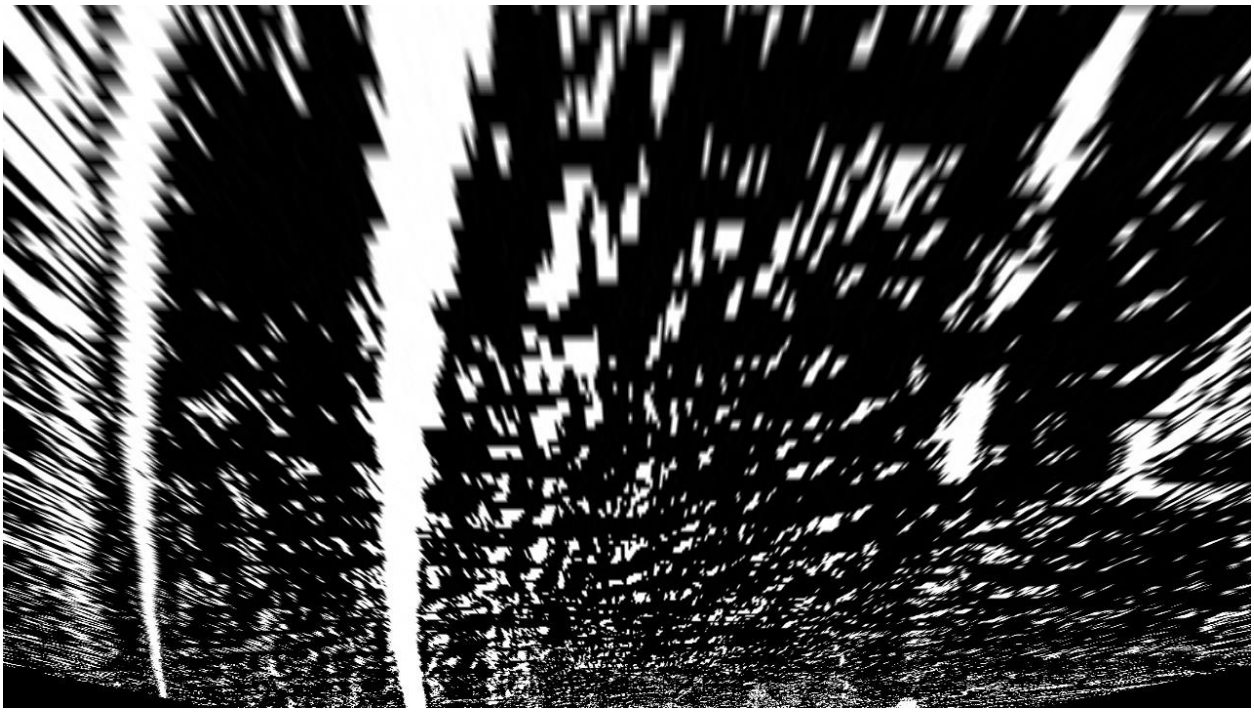
The code for my perspective transform includes a function called `persp_transform()`, which appears in the third code cell (Labeled 'In[12]')in the notebook. The `persp_transform()` function takes as inputs an image (`img`), and computes the resulting source and destination points and returns a transformed image. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32([[img.shape[1] * (.5-mid_width/2),
img.shape[0]*height_pct],[img.shape[1] * (.5+mid_width/2),
img.shape[0] * height_pct], [img.shape[1] * (.5 + bot_width /
2), img.shape[0] * bottom_trim], [img.shape[1] * (0.5 -
```

```
bot_width / 2), img.shape[0] * bottom_trim]]) offset =
img_size[0] * .25 dst = np.float32([[offset, 0], [img_size[0] -
offset, 0],[img_size[0] - offset, img_size[1]], [offset,
img_size[1]]])
```

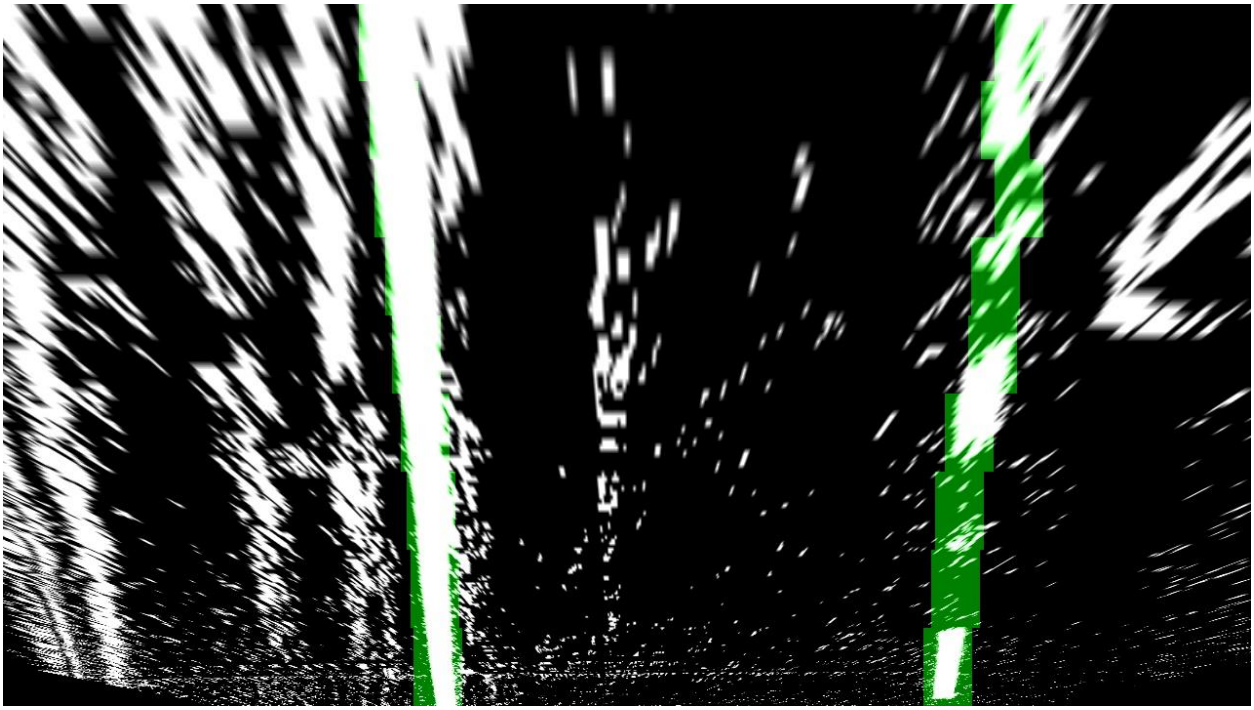
This resulted in the following source and destination points:

Source	Destination
588, 446.4	180, 0
691.2, 446.4	540, 0
1126.4, 637.2	540, 1280
153.6, 637.2	180, 1280



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

I used the sliding window technique discussed in project lessons and the file named 'tracker.py' which was provided to us during the recording of the Project 4 Q&A session on youtube to identify lane-line pixels. The 'tracker.py' class takes care of finding the pixels using cv2.convolve() function, which applies convolutions to maximize the number of "not" pixels in each window. Then it stores the peaks of the convolved signals and averages them to smooth out lane line identification. (file 'tracker.py' code lines #26 through 68 - function 'find_centroids()'). The lane positions are stored in leftx and rightx arrays. I then fit them with polynomials whose coefficients are computed and stored in left_fit and right_fit arrays. After that I substitute those coefficients into the polynomial 'Ay^2 + By + C' (code cell 3 in the notebook ('In[12]') function 'find_lanes'):



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.

The raidus of curvature is calculated using the formula described [here] (http://www.intmath.com/applications-differentiation/8-radius-curvature.php). The code for calculating the the radius of curvature of the lane is contained in the function find_lanes() in the third code cell of the notebook. I first take the meters per pixel in x and y directions and substitute those values in the formula as follows (Turns out radius of curvature was not computed correctly in previous submission, setting ym_per_pix to 30/720 and xm_per_pix to 3.7/700 does the job):

```
ym_per_pix = curve_centers.ym_per_pix # meters per pixel in y
dimension xm_per_pix = curve_centers.xm_per_pix # meters per
pixel in x dimension
```

```
curve_fit_cr = np.polyfit(np.array(res_yvals, np.float32) *
ym_per_pix, np.array(leftx, np.float32) * xm_per_pix, 2)
curverad = ((1 + (2 * curve_fit_cr[0] * yvals[-1] * ym_per_pix +
curve_fit_cr[1])2)1.5)/np.absolute(2 * curve_fit_cr[0])

camera_center = (left_fitx[-1] + right_fitx[-1]) / 2 center_diff
= (camera_center - warped.shape[1] / 2) * xm_per_pix side_pos =
'left' if center_diff <= 0: side_pos = 'right'
```

variable curverad stores the curvature radius

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 the third code cell of the notebook in the function `find_lanes()`. 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!).

Here's a link to my video
https://drive.google.com/open?id=0B3wm7W2ZfOg2MFB4SHQ0NEMxOGs

---

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?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further. My pipeline will probably fail in a series of sharp turns. I tried using different combinations of gradient and color threshholding, namely, tresholding the gradient along x and thresholding the magnitude of the gradient with color thresholds unmodified, and obtained extremely bad results. The lanes kept jumping to the left and narrowing towards the middle of the image. I believe using the direction of the gradient in the combination could somewhat improve those results