Advanced Lane Finding Project

by Sergiy Fefilatyev

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.

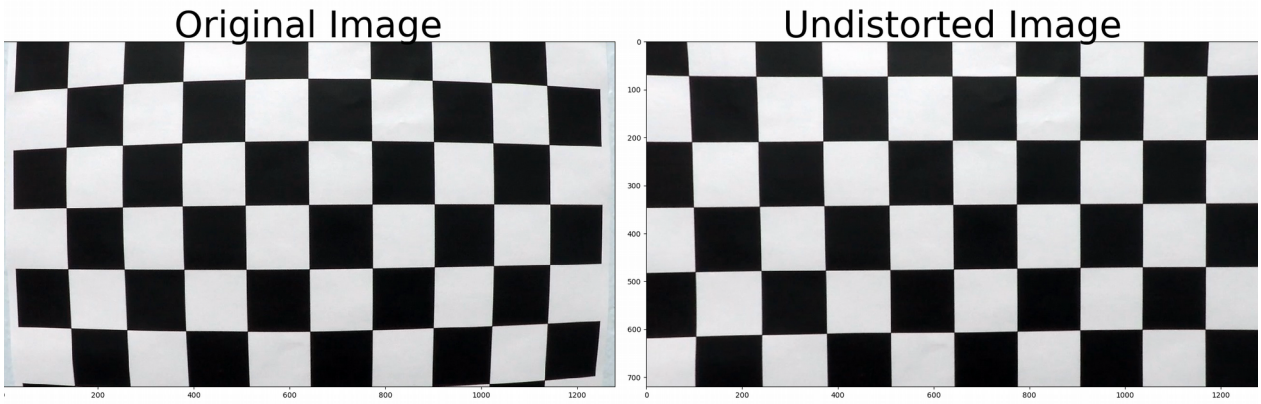
Rubric Points

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're reading it!

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

I used the provided images of chessboard pattern taken from different viewpoints for camera calibration. The chessboard images are used for the ease of finding correspondences between ideal undistorted orthogonally viewed chessboard and its images. When such correspondences are found it is possible to use known relationship between the distorted or projected image vs undistorted in order to find values of parameters. I used OpenCV's methods `cv2.findChessboardCorners,` `cv2.calibrateCamera,` to find correspondences, calibrate the camera, find distortion coefficients and methods `cv2.drawChessboardCorners,` `cv2.undistort` for display correspondences and obtain undistorted (rectified image). The image below shows an original chessboard image and its undistorted equivalent obtained with the described procedure.



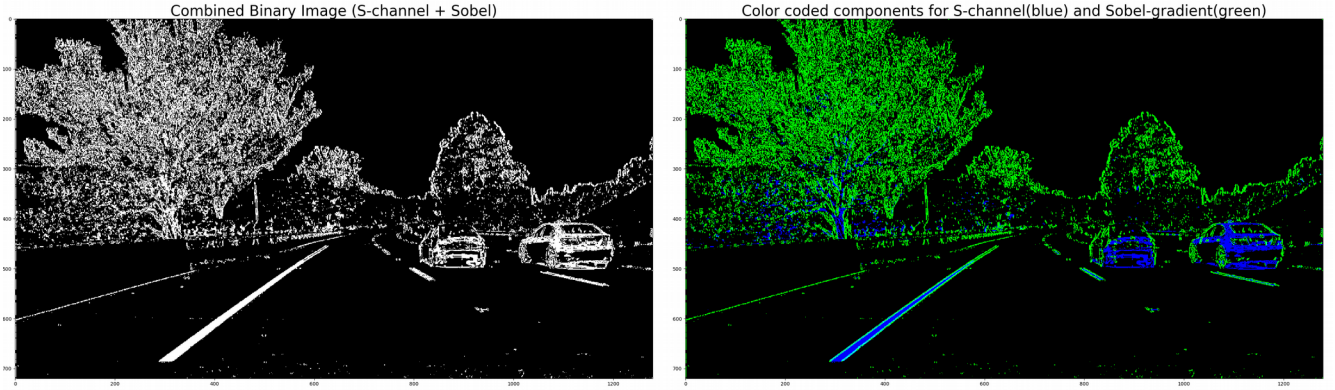
3. Provide an example of a distortion-corrected image.

Similarly, once the distortion parameters of the lens are found every image in the video sequence is rectified to remove radial distortion. The image above shows example of rectification applied to "test6.jpg" from the set of examples for this project.

Some obvious changes include the fact that the visible area has been shrank – very visible in the left right corner. The straight lines look more straight as well.

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.

In my method, **`threshold_image() [84-133]`** I used a single color channel "S" from HLS color space and a magnitude of gradient in horizontal direction (i.e. x-axis). For conversion of original RGB space into HLS I used an opencv's method `cv2.cvtColor`. To obtain a gradient I convert the RGB image first into a grayscale with the help of `cv2.cvtColor`, and then obtain a gradient with the help of `cv2.Sobel`. The image below shows an example of a binary thresholded image for the original "test6.jpg".



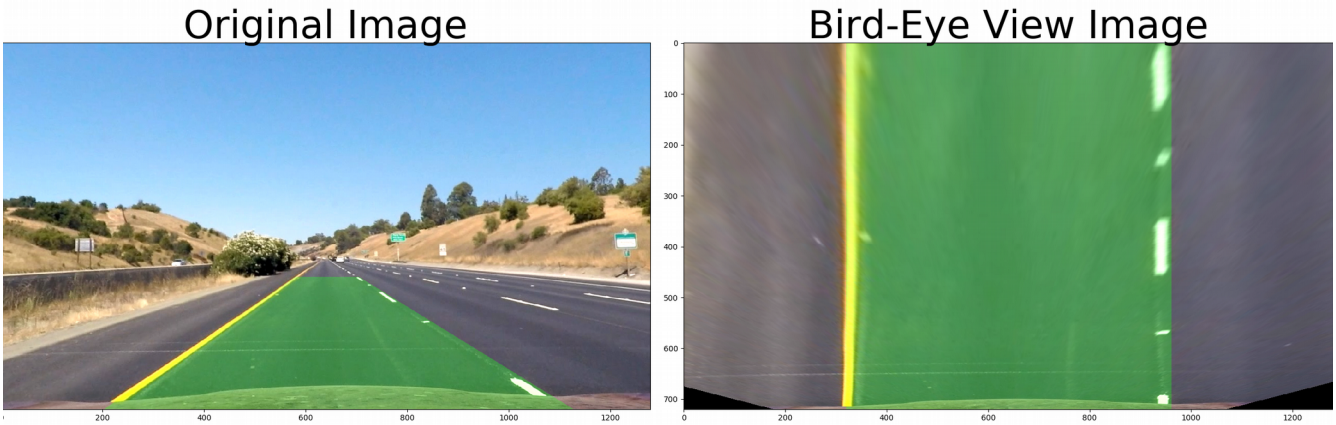
5. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I used a methods **`get_transform_params`**`() [136-145]`, **`make_perspective_transform`**`()[147-170]`, to perform a perspective transform. The former ones calculates the parameters needed the for correct perspective transformation. The later actually transforms the input image. In order to get transformation parameters I used a four hard-coded points in the image from "straight_lines1.jpg". I mapped those four points into their ideal position in a bird-eye image and calculate direct and inverse

perspective transforms with the help of OpenCV's method `cv2.getPerspectiveTransform`. To make a perspective transformation, I used OpenCV's method `cv2.warpPerspective` that makes use of the direct-transform parameters. The following four hard-coded points in the original input image and four points in the output image define the perspective transform:

	Input Image Coordinates	Bird-Eye Image Coordinates
Point 1	`580, 460`	`320, 0`
Point 2	`203, 720`	`320, 720`
Point 3	`1127, 720`	`960, 720`
Point 4	`705, 460`	`960, 0`

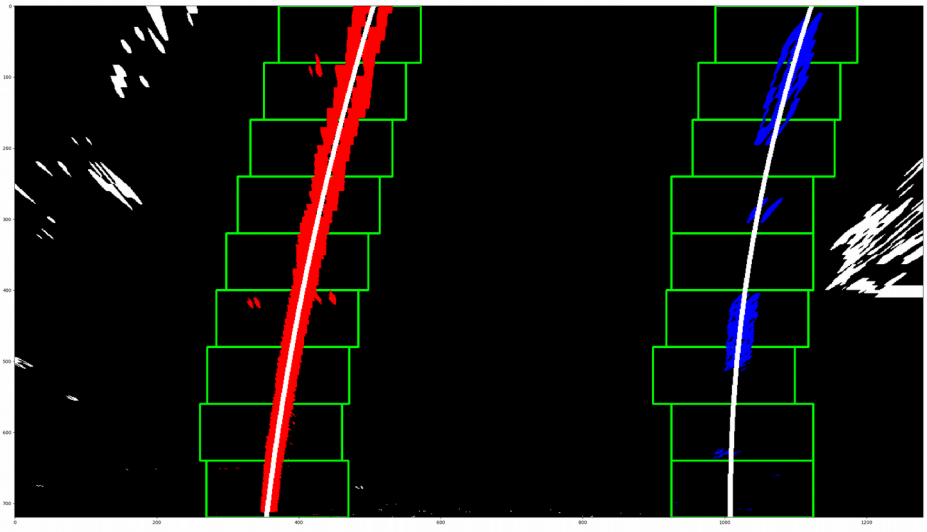
The image below shows an example of transforming input image 'test6.jpg' in its bird-eye equivalent.



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

The pixels for lane lines are initially found by using a sliding window approach in **`find_lanes_from_scratch()[303-390]`**. The follow-up frames use previous averaged position as a search space for looking for new lines. These are found by method **`find_lanes_continue()[393-432].`**

Both, methods **`find_lanes_from_scratch()`** and **`find_lanes_continue()`** use method **`fit_lanes`**`()[225-268]` for fitting the lane pixel position with a polynomial. I used `np.polyfit` for actual fitting of the data and getting polynomial parameters for second order polynomial. The following image shows the windows (green rectangles), detected line pixels (red for left lane, blue for right lane), and fitted polylines (white lines) as example of processing "test6.jpg".



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.

The code for identifying the radius of a curvature is located in method `calculate_curvature()[202-222]`. The curvature is calculated separately for left and right lanes and then averaged. I noticed that often there is a pretty large difference in curvature between two lanes and averaging is really needed. For conversion into metric units I used one constant 30/720 that defines a relationship between screen pixels and metric lane length = 30 meters for 720 vertical pixels. For horizontal pixel conversion I used an average distance between the lanes calculated in each frame and made equal to legally bound 3.7 meters.
To calculate the position of the vehicle with the respect to center I found a mid-point between the found lanes and compared it to the horizontal center of the frame. This provides a pixel difference, not a metric difference. In order to obtain a metric position with the respect to the center of the lane I divided the found pixel difference with the average length of the lane in pixels and multiplied by 3.7 meters – the assumed width of a lane. The code for this calculation is located in method `find_lanes_from_scratch()` and `find_lanes_continue()`. Important – left/right side of the center is noted by the sign of the difference. Negative means left to the center and positive means right off the center.

8. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.
The following image shows an example from input image “test6.jpg” with lanes clearly identified and warped onto the original image. The curvature value and distance from the center of the line is outputted as well.

Discussion:

I implemented the minimum requirements for the project. I did not introduce smoothing of curvature by averaging between the frames, but the original video is segmented well, without major mistakes, just a little wobling. However, the quick check on the other two videos show that the pipeline often fails on shadows and change in the pavement of the road – just as I was warned :) It does not seem to be difficult to manually tweak thresholding pipeline and introduce smoothing but in my opinion, that will only make it work on those two videos, without generalizing to other situations. For more robust lane detection the method should learn the parameters instead of hard-coding. Ideally, this project should rely on deep learning techniques which require would require a lot of training data – but I guess it is outside of the scope for this project.