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

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

Writeup / README

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

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 code cell of the IPython notebook located in "./image-proc.ipynb" with the title **Compute the camera calibration using chessboard images**.

I use a set of known images of a chessboard for which the cv2 library provides a useful API to calibrate images. 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 applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

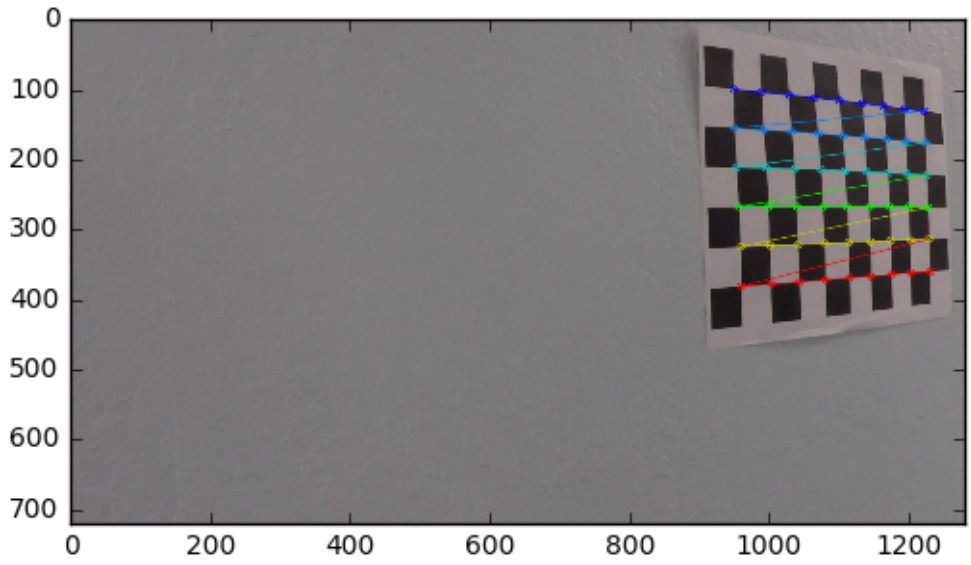

Figure 1: Undistorted

Pipeline (single images)

Note 1: Points 2 and 3 are flipped because that is the order in the image pipeline I have implemented.

Note 2: Many more images can be found in the `output_images` *directory included in the zip file.*

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:

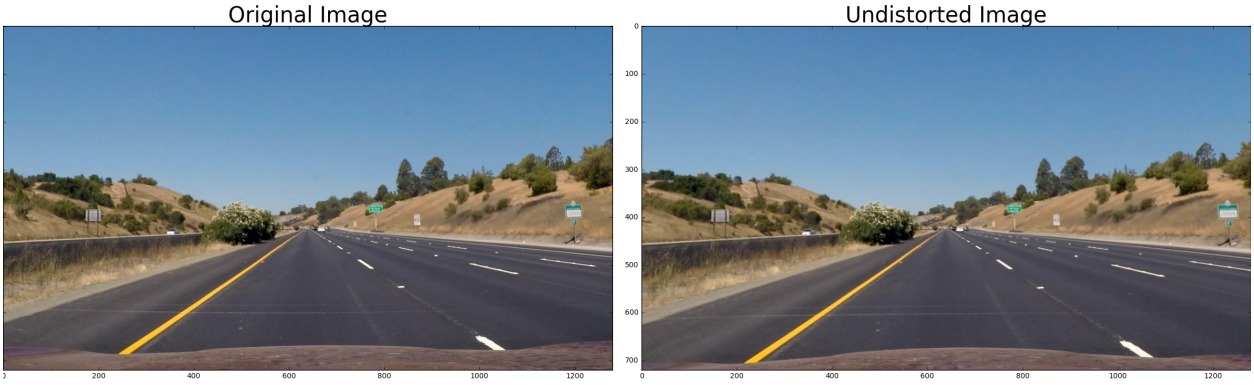

Figure 2: Original and Undistorted

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 is in the function `transform_perspective()`, which appears in the cell below the title **Transform Perspective**. It takes as inputs an image (`img`), as well as source (`srcpoints`) and destination (`dstpoints`) points. I chose to hard-code the source and destination points in the following manner:

```
srcpoints = np.float32([[270,670],[592,450],[691,450],[1041,670]])
dstpoints = np.float32([[270,670],[270,100],[1041,100],[1041,670]])
```

This resulted in the following source and destination points:

Source	Destination
270, 670	270, 670
592, 450	270, 100
691, 450	1041, 100
1041, 670	1041, 670

I verified that my perspective transform was working as expected by processing an image of validated straight lane image.

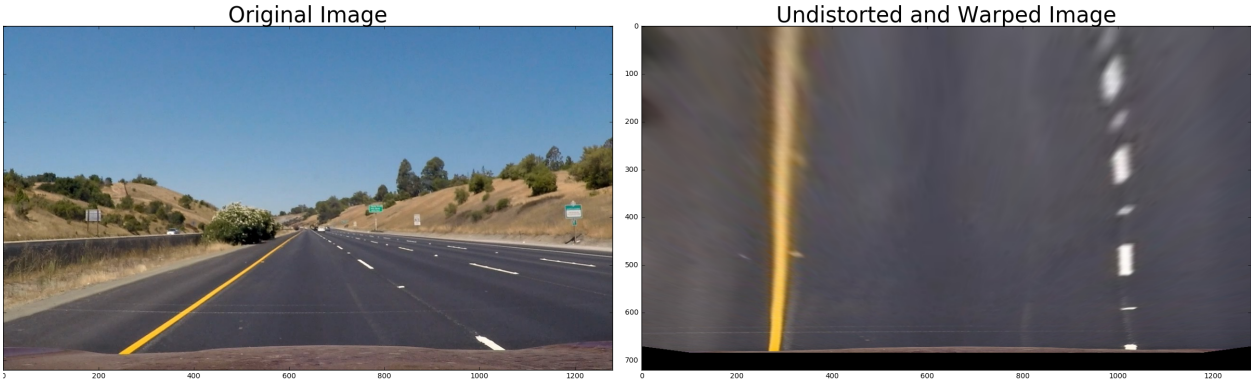

Figure 3: Transform Perspective

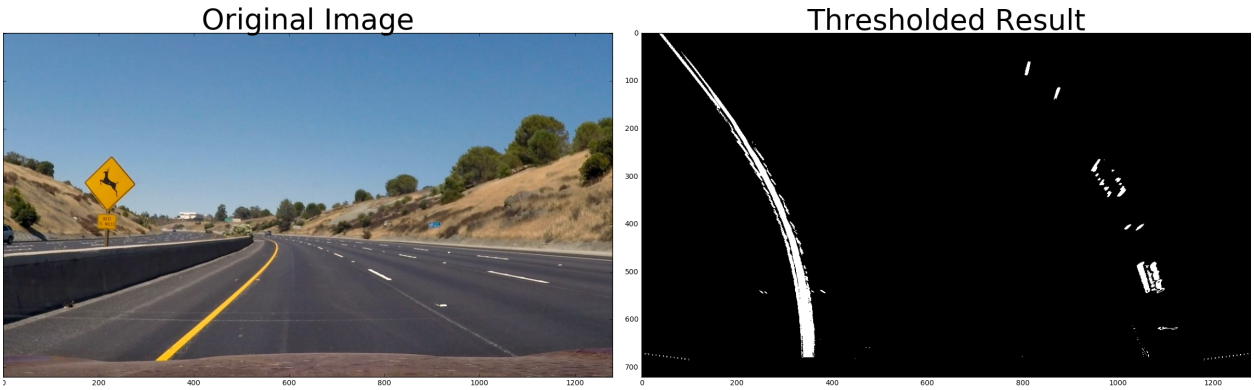

Figure 4: Color Space

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.

After changing of perspective I applied color transform. First I convert the RGB image into HLS color space. Then I applied the Sobel operator (see https://en.wikipedia.org/wiki/Sobel_operator) over the X derivative which highlights vertical lines while dimming horizontal ones (which supposedly would help to detect line lanes). Finally I filtered the S channel which would take care of picking white and yellow lanes. The code is under the cell with the title **Thresholded image**. I have tried several value combinations and finally I settled with a threshold of *(170-255)* for the S channel and a *(50-150)* for Sobel operator X derivative. Similar values were used by other collegues like Paul Heraty The result is shown in **Figure 4**.

Figure 5: Centroids by Sliding Window

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

Then I implemented a sliding windows algorithm to identify the centroids of windows (of size `window_width=50` by `window_height=80`) where the left and the right lane lines have the biggest probability to be in. The code is under the cell with the title **Centroids using Sliding Window**.

It results in two sets of nine (720/80) points each (one for the left lane and one for the right lane.) After that I filtered centroids which are more than 50 pixels away from the previous y coordinate centroid.

Each set has enough points to let numpy fit a second order polynomial using the function `np.polyfit`. The boxes representing each centroid can be seen in **Figure 5** above (less than nine because some of them were filtered.)

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 radius of curvature at any point x of the function x=f(y) is given as follows:

$$\mathrm{R_{curv}} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = 2Ay + B$$
$$f''(y) = 2A$$

So the equation for radius of curvature becomes:

$$\mathrm{R_{curv}} = \frac{[1 + (2Ay + B)^2]^{3/2}}{|2A|}$$

And its coded in the first lines of the function `curvature()` in the cell under the title **Curvature**.

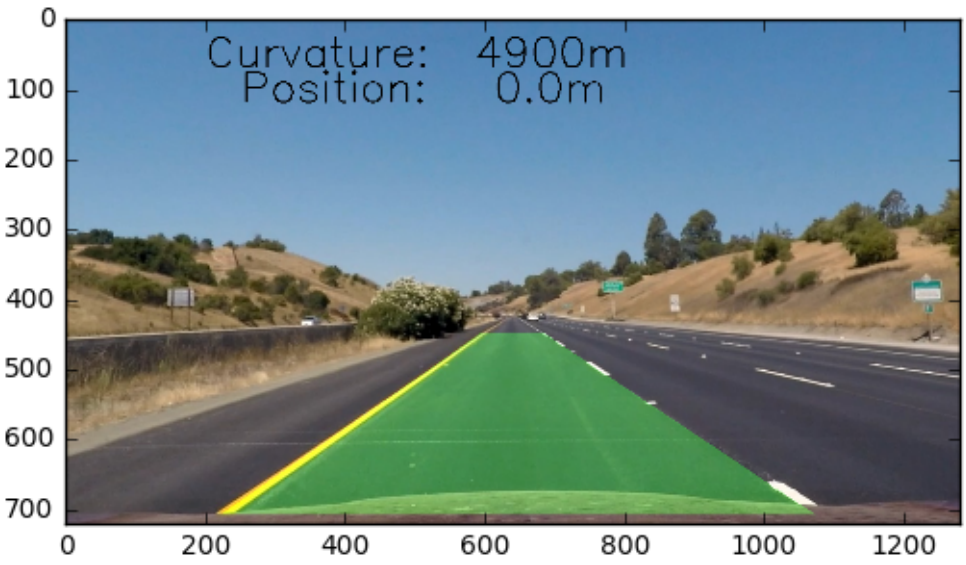

Figure 6: Drawing the lane area

The results were converted from pixels to meters and rounded to the nearest 50m for values less than 1,000 and to 100m for values over it.

Given the `x` values for the bottom of the image for the left and right polynomial and ssuming that the distance between each line is 3.7m and that the camera is in the middle of the car, its position is obtained by a simple cross-product.

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

Figure 6 shows the final output of a test image which would be like a video frame. The function `draw()` in the cell under the title **Drawing** displays over the road the area that was identified as lane.

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!).

The first version of the video resulted in too much wobbling. It specially suffered under shadow conditions. However after I applied some smoothing the quality improved a lot. The smoothing and frame validation I applied are:

1. Skip outliers in centroids.
2. Reject frames were the lane width differs more than 1.0m of the American standard 3.7m
3. Reject frames were the left and right curvature radius differs more than three times.
4. Reject frames were the located lane jumps more than 0.8m with respect to the previous ones.
5. Finally the lane to be shown is represented by a set of coefficients each of them is the result of the mean of the previous 10. I implemented it with a ring buffer (`collections.deque`)

Here's a link to my video result

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?

Although (or because) the pipeline is pretty straightforward this implementation has several issues:

1. There are still a lot of parameters to tune, like sliding window size, color space thresholds, perspective transform polygon, etc.
2. It depends too much in the ability to detect *both* lines lane.
3. The smoothing step can be improved, for instance by keeping two buffers one for each line.

A better parameter tuning (maybe using some grid search technique) most probably will help dealing with different types of pavements or line painting. Also a good smoothing step would be able to cope with faint lines. However whenever a line is missing for a long section, say a hundred meters, this implementation will fail miserably.