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.

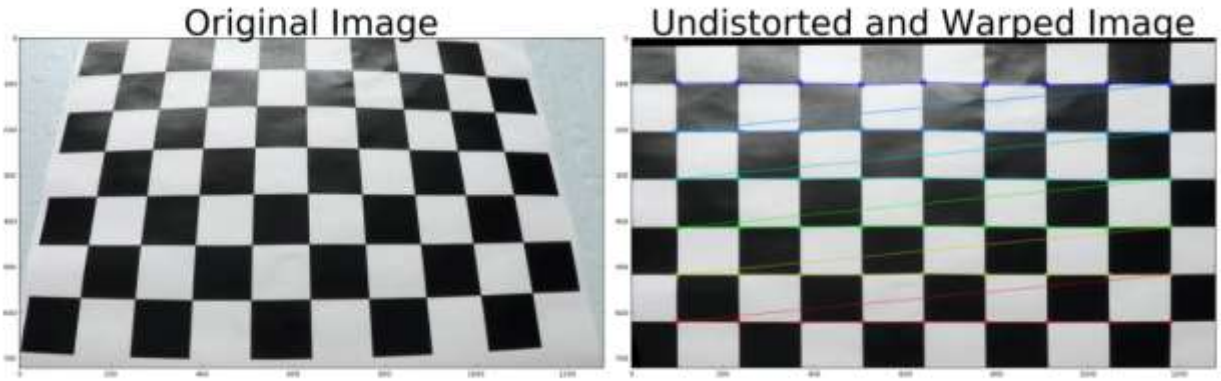
Writeup / README

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 located in get_coefficients.ipynb. 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. The `objpoints` is appended with an array 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:



A perspective transform has also been applied to the chessboard image. This will be discussed later on in the project.

Pipeline (single images)

All images are passed through the function named *pipeline*. Each step in this pipeline is discussed below:

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:

The distortion coefficients from the chessboard images are loaded into a pickled dataset and do not need to be calculated again. The remaining code for the project may be found in **project_submission.ipynb in cell block 4.**

The function *undist* is used to undistort all further images. It simply reads in the calibration matrix and the distortion coefficients calculated from the chessboard images and returns an undistorted 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 (in the function *color_shift*). Here's an example of my output for this step. The function first converts the images to HLS space, then applies a gradient and grayscale thresholding operation. The H and S channels are also thresholded and combined with the gradient thresholded binary images.

The result is shown below:

There are certain regions that commonly present problems. I masked these images to remove unwanted regions.

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 `warper()`, which appears in lines 93 through 99 in the 4th cell block. The `warper()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. The src points can be identified in a line. I used an image reader on my desktop to find these points. The destination points were calculated by hand using slope formulas and accounting for perspective shift.

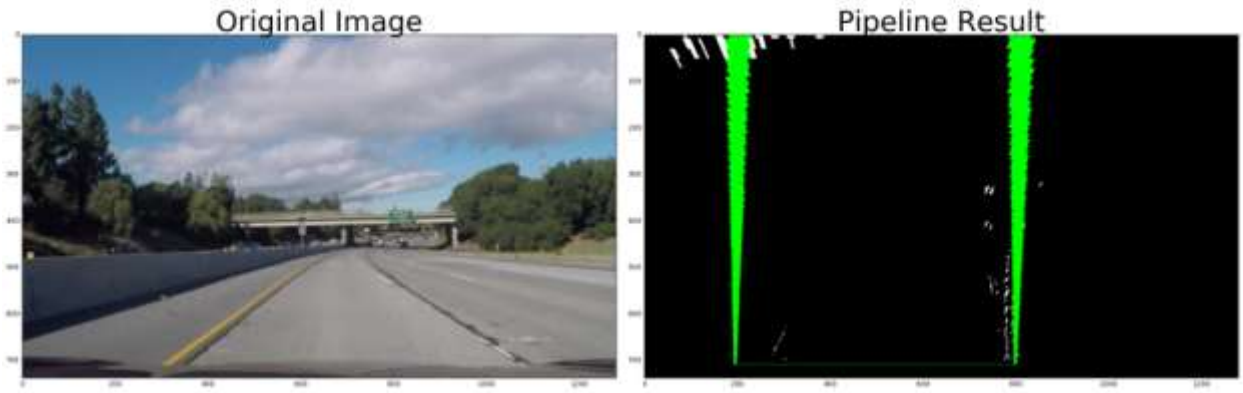
src = np.float32([[570,460],[713,460], [155,705],[1200,705]])
dst = np.float32([[155,100],[870,100], [155,705],[870,705]])

This resulted in the following source and destination points:

Source	Destination
470,460	155,100
713,460	870, 100
155,705	155, 705
1200,705	870, 705

Notice that the destination points form a rectangle. The Y values need to be close to the top of the image.

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.

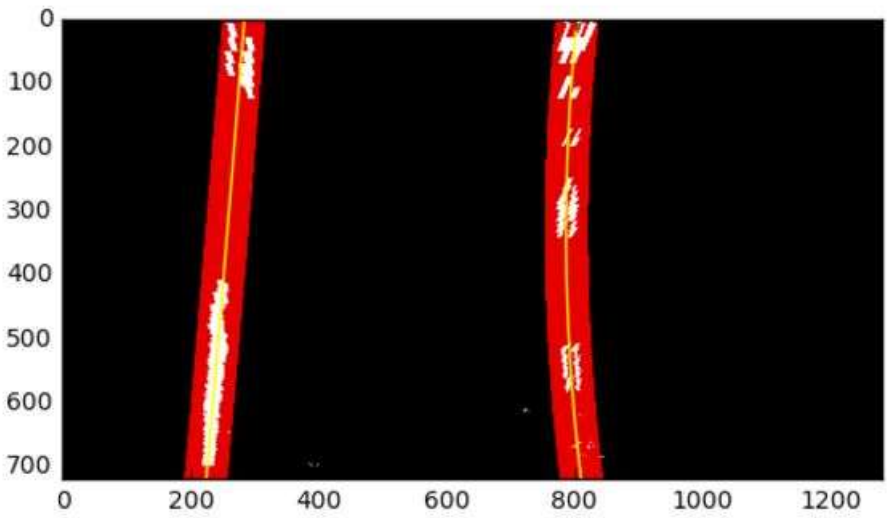


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The largest function in the code is to identify lane line pixels. This can be found in the find_lines function in lines 130 to 292. The function first stacks the line pixels into a 3 channel image. If this is the first image, a histogram is used to identify the lane line base position. From then, 9 sliding windows are used to identify the rest of the lane line pixels, By identifying all nonzero pixels. If a pixel group is below a certain size, it is discarded.

The left and right pixel positions are stored in the variables X_left and X_right. If these arrays are empty, the line is discarded, otherwise the line is fitted to a 2^{nd} order polynomial using *np.polyfit*. These coefficients are then fitted to the y points in the image and stored in a Line object.

For all subsequent frames, The sliding window does not need to be used. The function can search around the previous line for new line pixels. For error checking, lines with coefficients that are substantially different than the previous coefficients are discarded. The lane lines are then drawn onto a blank image as shown below.



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.

In line 292 – 337 in my code, I calculated the radius of curvature of the lines. First, the polynomial coefficients and lane lines are loaded into the function. The radius of

curvature is first calculated in pixel space using the formula:

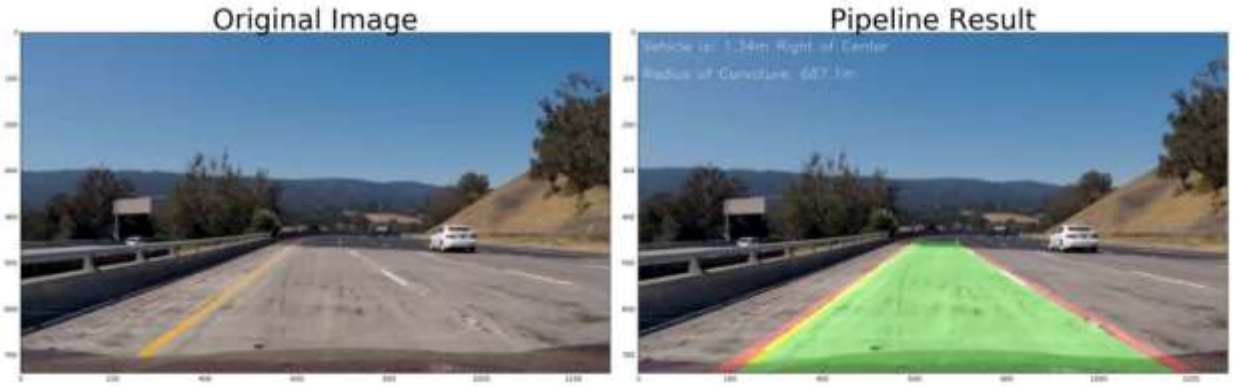
$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

Where Y = Ax^2 + Bx + C represents a lane line. The lane lines are also calculated in real world meters by multiplying the y values by 30/720 and the x values by 3.7/700 (formula given in class. The radius of curvature needs to then be recalculated using the above formula with a newly fitted line.

The base position is calculated in lines 365 through 376. I measured from the center of the image to the line base position for both the left and right lines, then subtracted the result to get the amount that the vehicle was off center.

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

Lines 358 through the end of the cell block is used to plot the result. The left and right points are first transposed into a usable format, then stacked into a single array. *CV2.FillPoly* is then used to fill the area between the lane lines. The result is then plotted over the original, undistorted image. The radius of curvature and the amount that the vehicle is off center is then displayed over the image. The result is shown below:



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 final 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?

The largest difficulty that I ran into in the project was the perspective transform. Although the results visually made sense, The lane lines were not parallel. I had to read several discussions and apply trial and error to find the source and destination points. I also discovered that I my threshholds were too stringent in the recoloring phase. This resulted in several lane lines that could not be detected.

Although this project works for this particular video, alternate lighting conditions (such as glare) or false lane lines, such as with road construction could prove detrimental to this project. I am toying with additional filtering options in order to pass the further challenge videos.