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

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. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

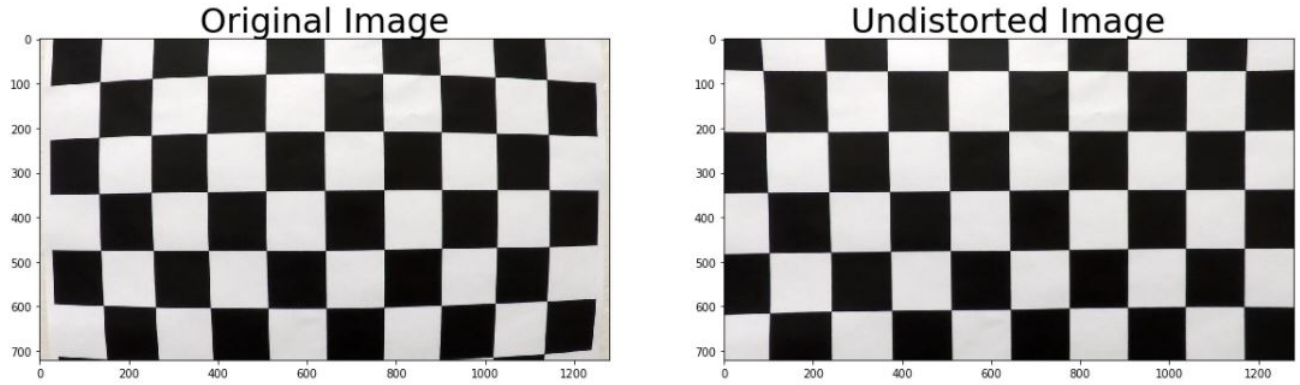
Camera Calibration

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 first section the IPython notebook located in `Workbook.ipynb` , labelled under **Camera Calibration**.

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:



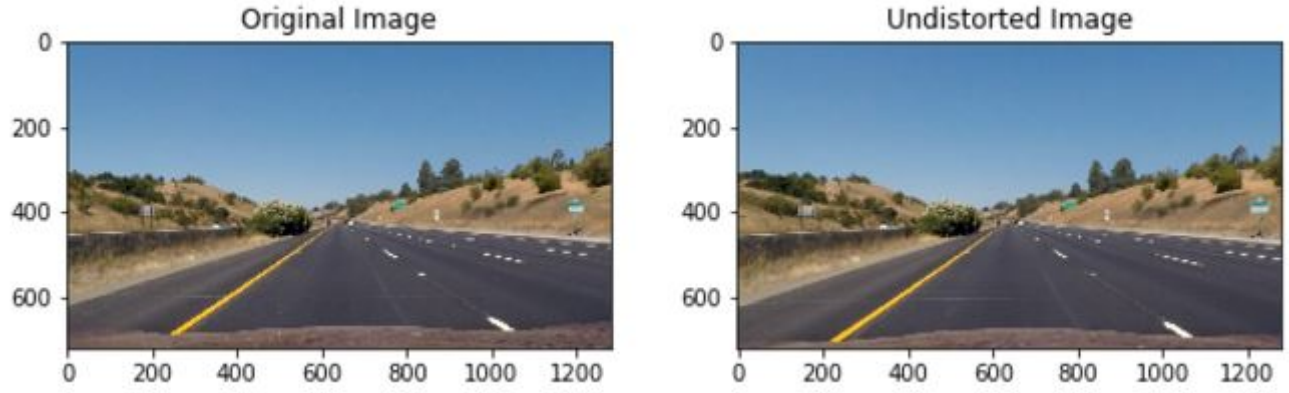
Calibration data is then saved to "./camera_cal/dist_pickle.p" to be used in later part of the notebook.

A `undistort()` function was also created to simplify calling of the `cv2.undistort()`

Pipeline (single images)

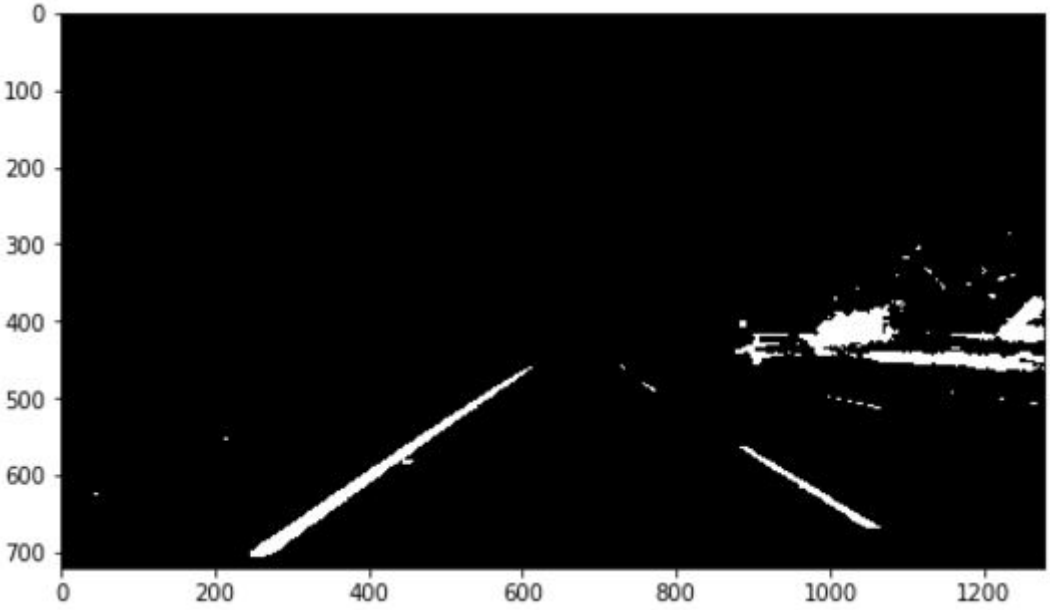
1. Provide an example of a distortion-corrected image.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



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 Red channel from RGB and B channel from LAB to generate a binary image (thresholding steps under section **Create Binary Thresholded Image** in `Workbook.ipynb`). 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 `perspective_transform()`, which is defined in section **Apply a perspective transform to rectify binary image** under `Workbook.ipynb`. The `perspective_transform()` function takes as inputs an image (`image`),

as well as a boolean variable `topdown` to calculate either `M` or `Minv`. I chose the hardcode the source and destination points in the following manner:

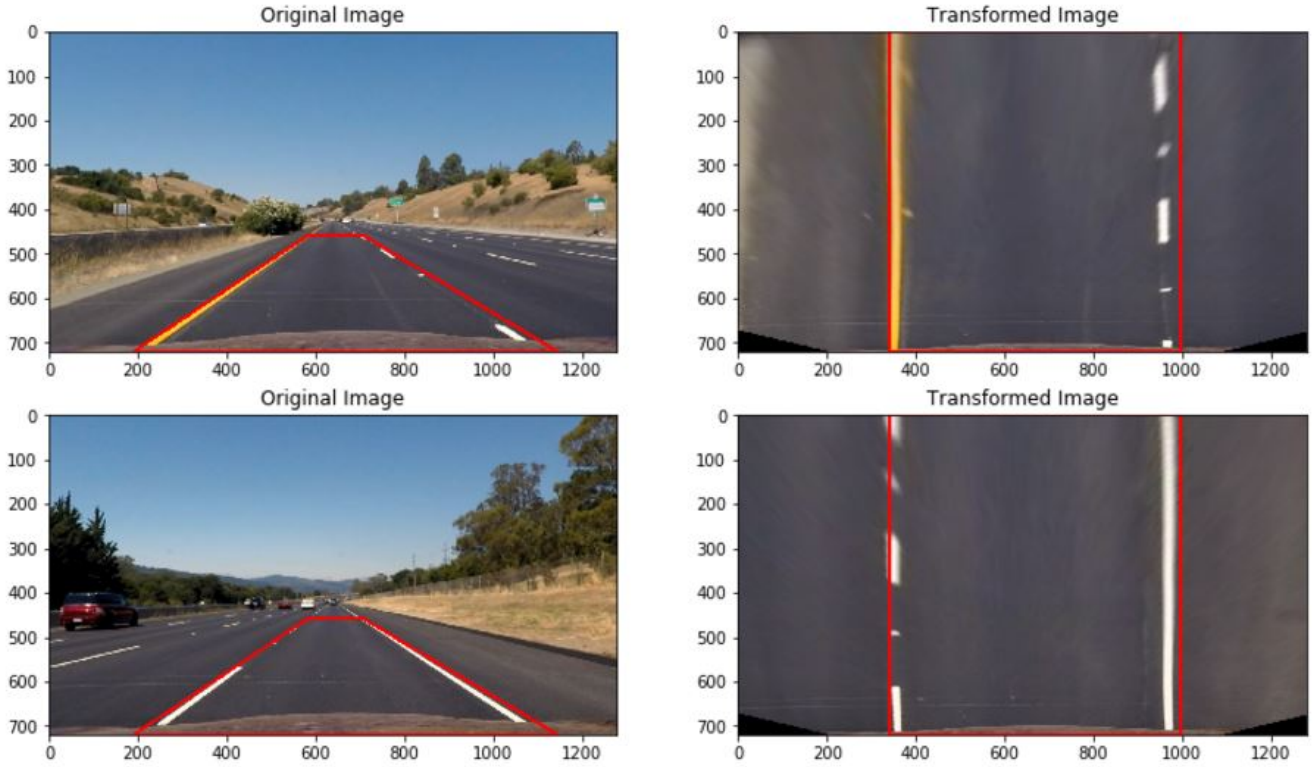
```
src = np.float32([[190,720], [584,457], [705,457], [1145,720]])
    new_top_left = np.array([src[0,0], 0])
    new_top_right = np.array([src[3,0], 0])
    offset = [150,0]

    dst = np.float32([src[0]+offset,
                      new_top_left+offset,
                      new_top_right-offset,
                      src[3]-offset])
```

This resulted in the following source and destination points:

Source	Destination
190, 720	340, 720
584, 457	340, 0
705, 457	995, 0
1145, 720	995, 720

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.



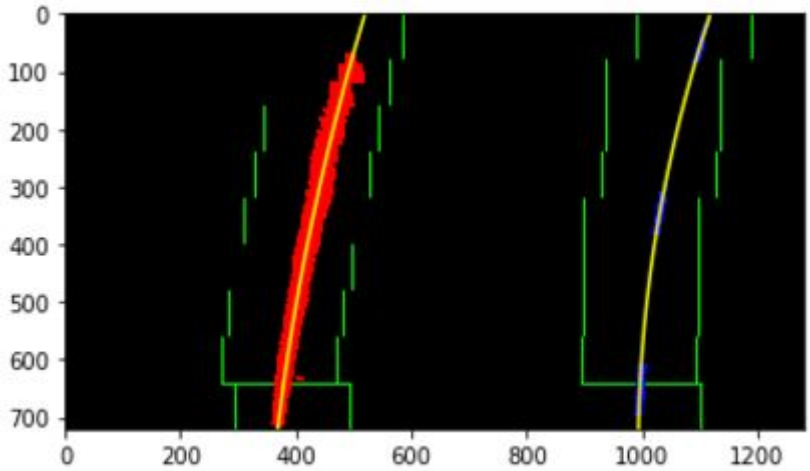
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The work to indentify lane-line pixels and fit their positions are in section **Locate the Lane Lines and Fit a Polynomial** in `Workbook.ipynb` . This is done using the **Peaks in a Histogram** method as taught in class.

The function first takes a histogram of second half of the image. Then it finds peaks of the two halfs of the histogram. These two are the starting point of the left and right lanes.

The function then divides the image in y-axis into 9 to create sliding windows. Starting from the bottom, it searchs upwards from the two starting point. Allowing to be recentered within +/- 100 pixels if minimum of 50 pixels are detected from the histogram.

After some processing, two second order polynomial is fitted over the left and right lane data. An example of the fitted data can be visualized below.

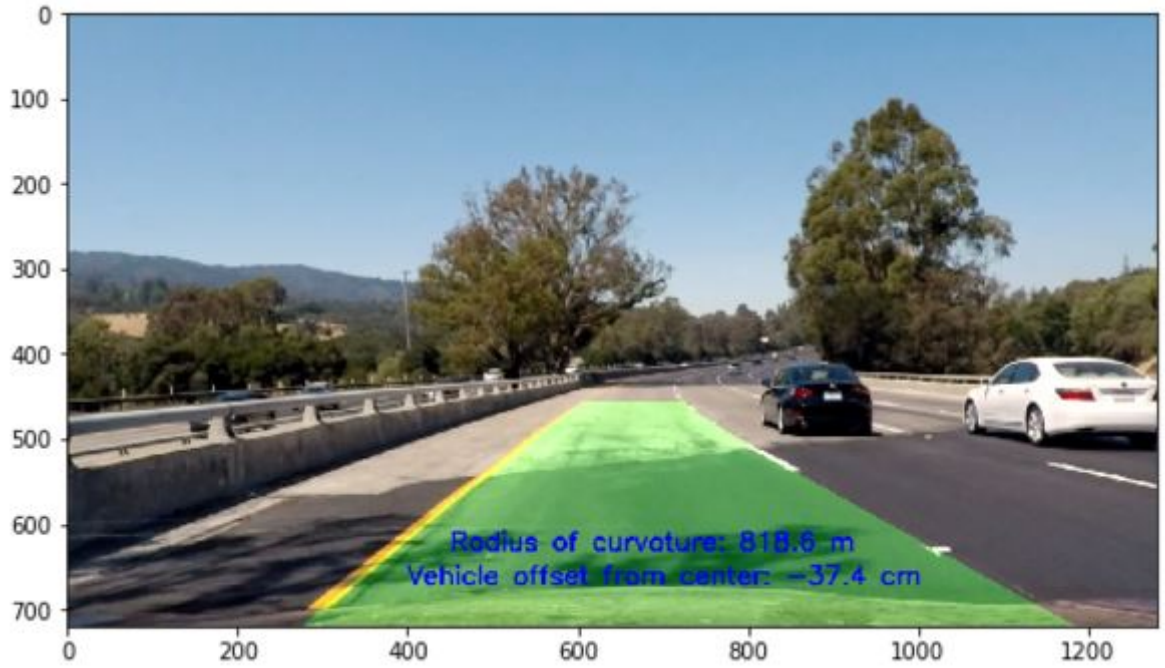


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Implementation is presented in section **Measuring Curvature** within `Workbook.ipynb`. Again, this is done using the method presented in class. There are two functions defined in the section: `calc_curve()` and `calc_vehicle_offset()`. `calc_curve()` calculates the curvature using the method demonstrated here. `calc_vehicle_offset()` calculates averages of the left and right lanes relative to center of the image. Midpoint of the image is assumed to be the center position of the car. The averages of the two lanes are calculated using the two bottom points of the fitted lanes.

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 section **Build the processing pipeline** in `Workbook.ipynb`. 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 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?

To create the binary image for detection of the lane lines, I did not use any gradient thresholding. Using only two color channels I was able to get good results in the test images and project video. However, it fails in the more challenging videos. To improve it in the future, I would extract images from these challenging videos and evaluation the function for generating binary image.

Lines detected from the previous frame should be used to help detect the next frame. Previous line would just be shifted down (as a function of the speed). This would also allow the use of filtering to reduce noise on the image.

There are often times when lane markings couldn't be accuractly detected from the image. I was finding it hard to visually distinguish the lines in the more challenging videos. By building in a reject mechanism when there aren't enough data to detect lane lines would improve accuracy.

Instead of using least square fitting, robust regression could be used to reject unwanted noise.