Self Driving Car Engineer Nanodegree
Project 4: Advanced Lane Finding
Mano Paul

The goals / steps of this project were the following:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply a distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image ("birds-eye view").
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Map the lane by drawing the driving area.

Once the above pipeline (steps 1 through 7) were successfully complete, the pipeline was applied to the project video and the lanes were mapped real-time to the video.

1. Camera Calibration

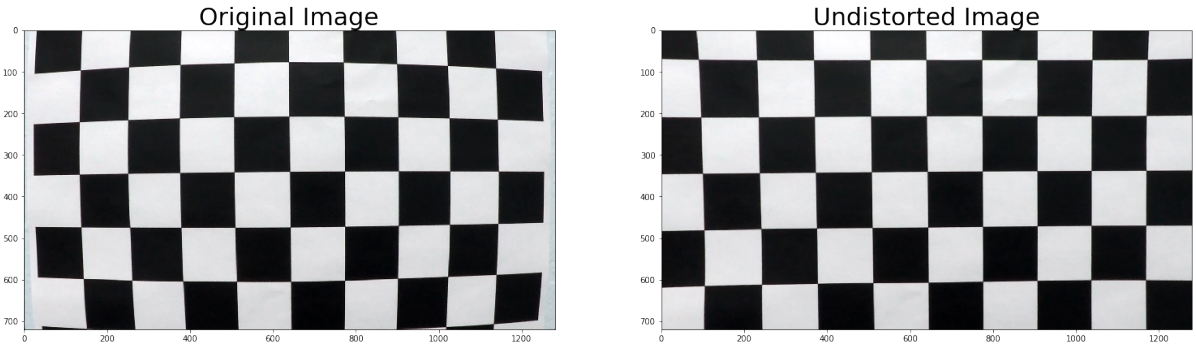
The code to compute the camera matrix (mtx) and distortion coefficients can be found in cell 3 of the IPython notebook `Advanced_Lane_Finding.ipynb`

I start by identifying that the number of inside x corners is **9** and the number of inside y corners is **6**. The I started 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, `objpoints` 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. `objpoints` will hold the 3d points in real world space. Then I defined an `imgpoints` array to hold the 2d points in image place. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

For each test image in the `camera_cal` folder, the `chessboardcorners` were tried to be found and the ones for which it was not possible to detect them, that files was identified and printed out to the screen as shown below.

```
findChessboardCorners failed: camera_cal/calibration1.jpg
findChessboardCorners failed: camera_cal/calibration4.jpg
findChessboardCorners failed: camera_cal/calibration5.jpg
```

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:



Pipeline (single images)

2. Correct distortion in Images

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

3. Binary (thresholded) image using color & gradient transforms

I used a combination of color and gradient thresholds (abs_sobel_thresh, magnitude_thresh, direction_thresh) with a kernel size of 17 to generate a combined binary image (thresholding steps at cells 1 through 10 in `Thresholds.ipynb`). Here's an example of my output for this step.

4. Transform Perspective to get a birds-eye-view

The code for my perspective can be found in cells 15 through 18 in the file `Advanced_Lane_Finding-Test_Harness.ipynb`. (Note: this is a Test Harness File and not the same as the Project ipynb notebook).

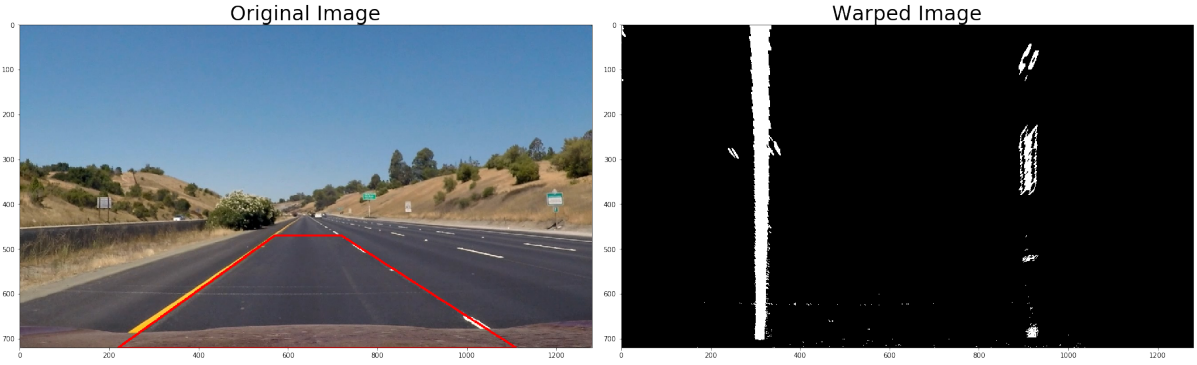
I chose the hardcode the source and destination points in the following manner:

```
In [15]: src_bottom_left = [220,720]
         src_bottom_right = [1110, 720]
         src_top_left = [570, 470]
         src_top_right = [720, 470]

         # Destination points are chosen such that straight lanes
         # appear more or less parallel in the transformed image.
         dest_bottom_left = [320,720]
         dest_bottom_right = [920, 720]
         dest_top_left = [320, 1]
         dest_top_right = [920, 1]

         '''
```

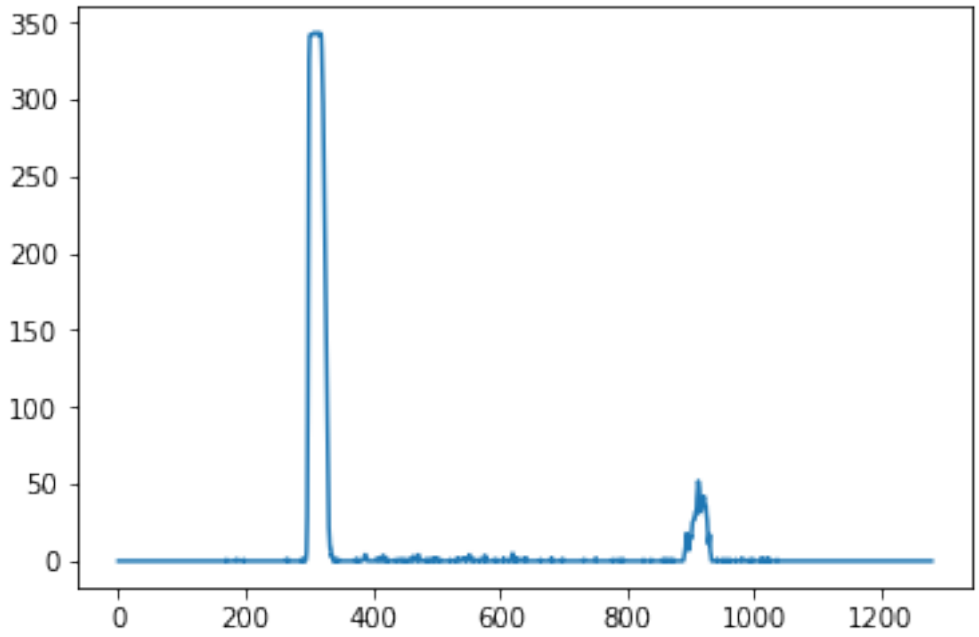
I verified that my perspective transform was working as expected by drawing the src and dst points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



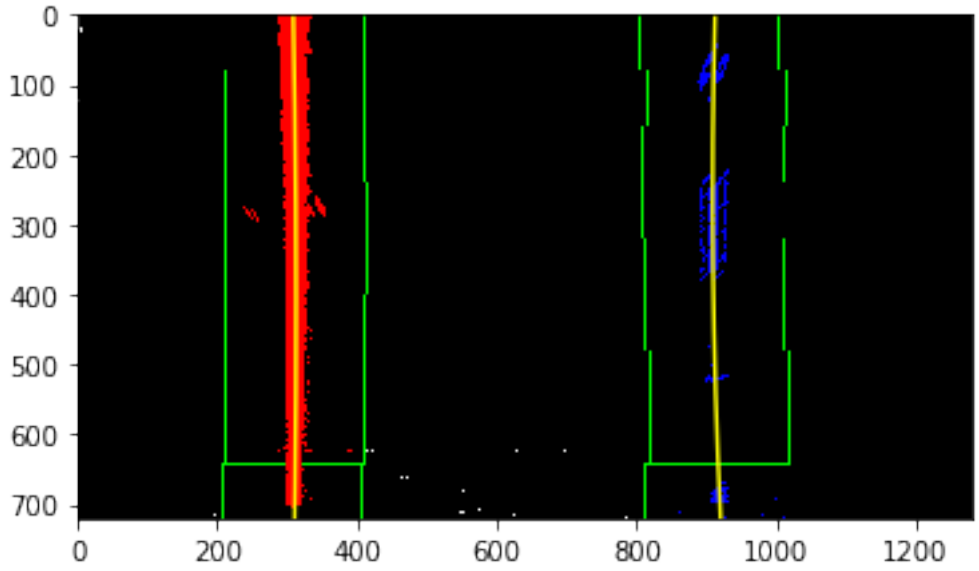
5. Detect Lane Pixels and Fit to find lane boundary

Once the perspective transform was successful on the binary (thresholded) image, I found the lane pixels by determining the peaks in the histogram of the warped binary image. This can be found in the cells 19 through 21 in the `Advanced_Lane_Finding-Test_Harness.ipynb` file.

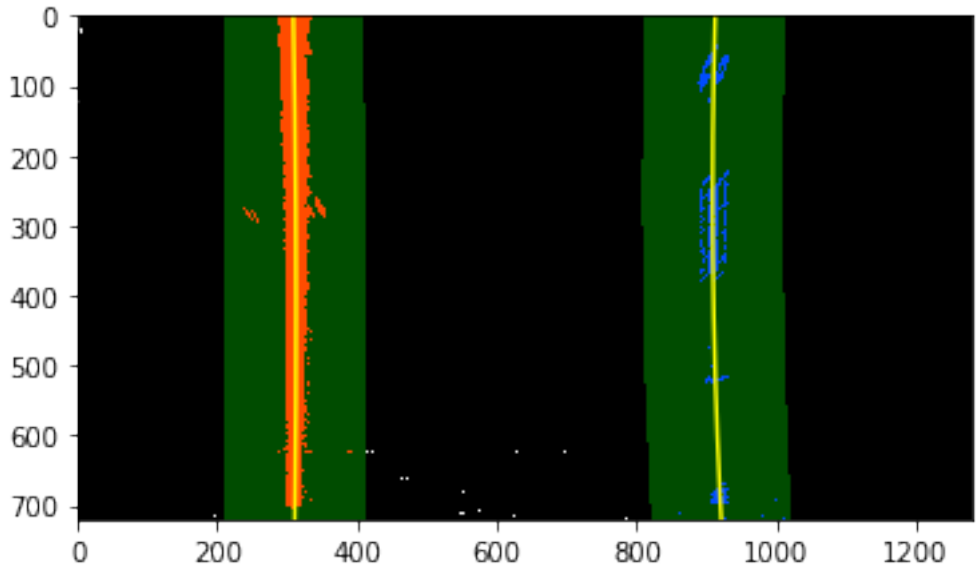
Then I found the left and right lanes from the histogram peaks as shown below.



Then by fitting a polynomial and using sliding windows, using the code in `findLanesPixelsUsingSlidingWindows` function I found the lane boundary, as shown.



Using lanes that have already been detected to find lane pixels, the next lane pixels were detected. This is more efficient as you do not have to search for sliding window again once the lane is detected. The `findLanesPixelsUsingPreviouslyDetectedLanesPixels` function was used for this and the following image was generated.



6. Determine Radius of Curvature and Center Shift (Offset)

The I determined the radius of curvature and center shift of the vehicle in cells 12 through 13 in my code in `Advanced_Lane_Finding.ipynb`. See next section for an example image.

7. Map the lane by drawing the Driving Area

I mapped the lane by drawing the drawing area onto the image. This was done by warping the detected lane boundaries back onto the original image and outputting a visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position. The following image was generated when the radius of curvature and center shift was determined.

Pipeline (video)

The above pipeline (single images) was applied to the project video and the lanes were mapped successfully.
The code can be found in the `adv_lane_detection_pipeline()` function.
Here's a link to my video result
Video location: https://youtu.be/1mOX0BjNaRo

Discussion Points – Problems and Observations

Finding the correct threshold took considerable number of trials because the images had varying degrees of color, shadows and curves on the lanes. Additionally the source and destination coordinates to do the perspective transform was hardcoded in the interest of time and making this dynamic would make this more robust.