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.

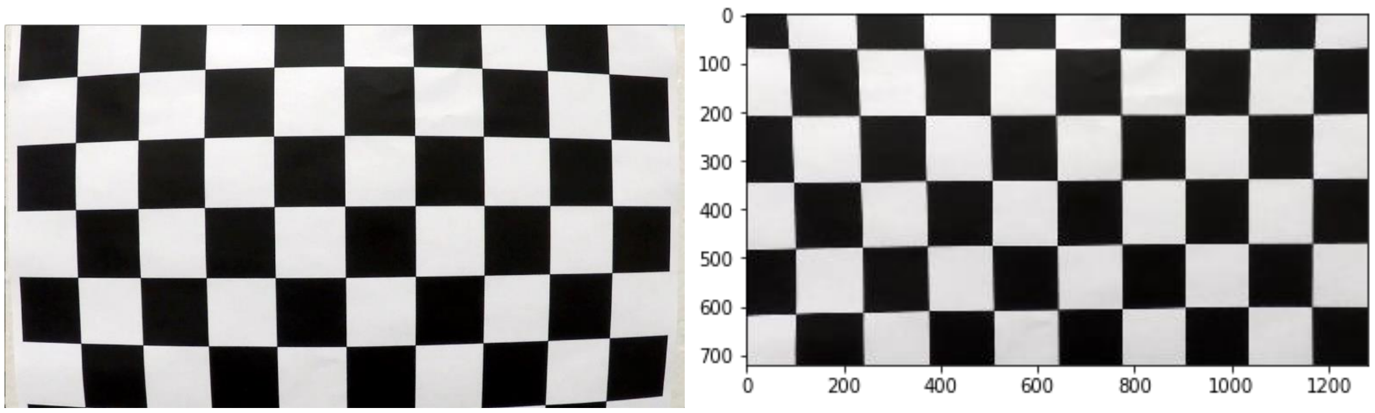
Camera Calibration

1. Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?

The code for this step is contained in the first code cell of the IPython notebook called `example.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. 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. To detect corners, I used OpenCV in-built function `findChessboardCorners` and drew the corners using `drawChessboardCorners`.

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:



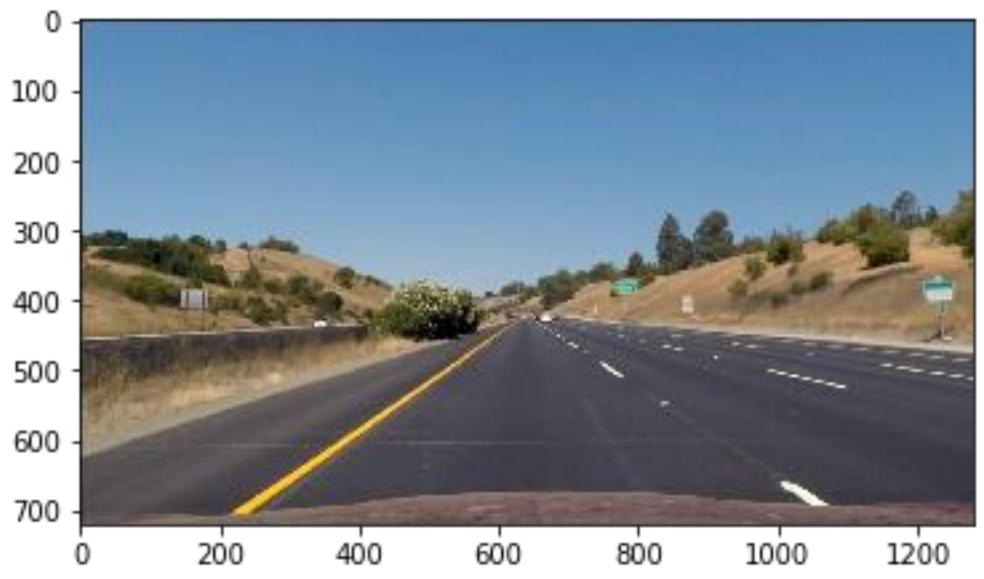
We can see that the algorithm to calibrate camera and undistort is working good as the resultant image have no visible distortion and all squares are of same size.

Pipeline (single images)

1. Has the distortion correction been correctly applied to each image?

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

Uncalibrated

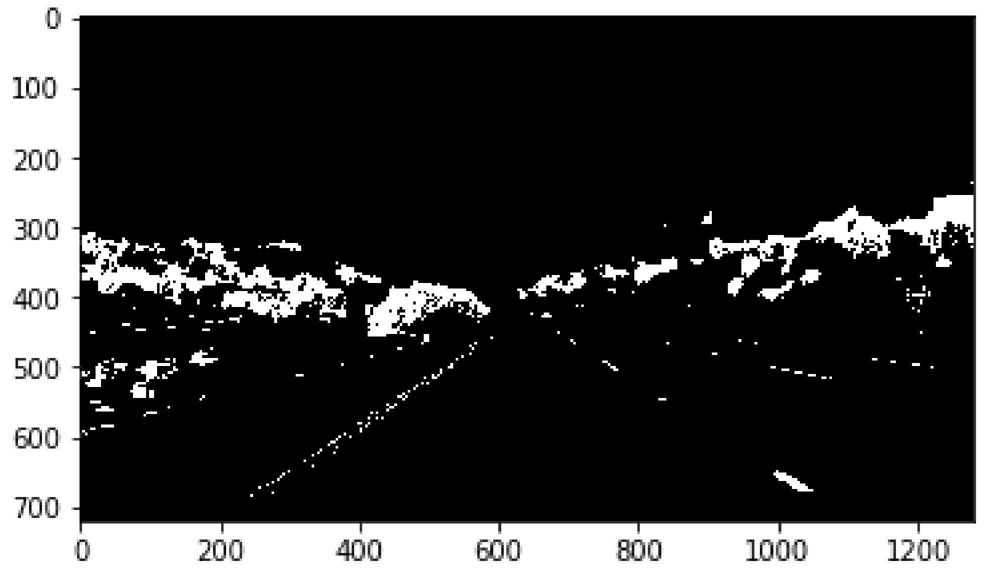


Calibrated

The two images almost look similar near the center of the image. But as we move towards the edges, the distortion seems to be increasing. This is common among many cameras and we can see this effect of distortion clearly on sign boards in the left and right corner of the image. The two sign boards seem to be facing camera directly after removing the distortion.

2. Has a binary image been created using color transforms, gradients or other methods?

A color transform was applied to the image to change the color space from RGB to HSL. This conversion was made to detect lane lines irrespective of the illumination conditions and altering color shades of lane lines. A min and max threshold were set to identify the lane lines from the rest of the image. I have used two separate thresholds for yellow and white lines and combined both detections. The resultant binary image was able to detect lane lines

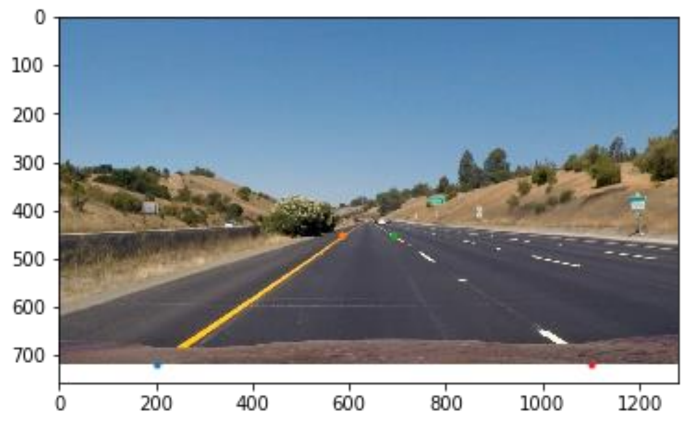


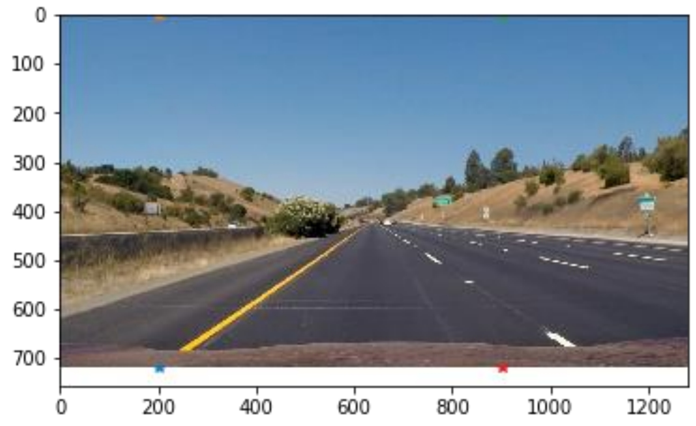
3. Has a perspective transform been applied to rectify the image?

The code for my perspective transform is includes a function called `unwarp()`. The `unwarp ()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

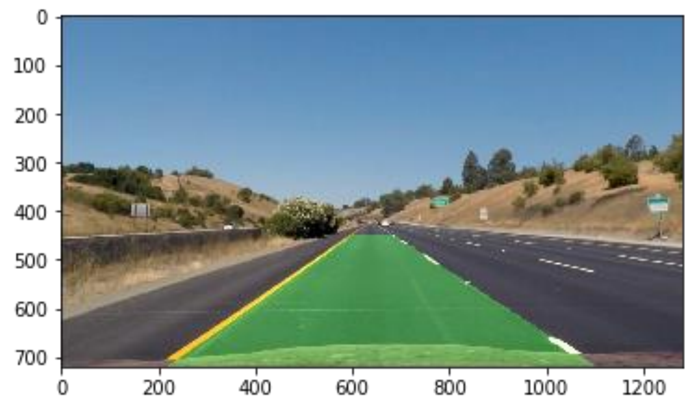
```
src= np.float32([

[200,720],

[585,450],

[690,450],

[1100,720]])

dest=np.float32([

[200,720],

[200,0],

[900,0],

[900,720]])
```

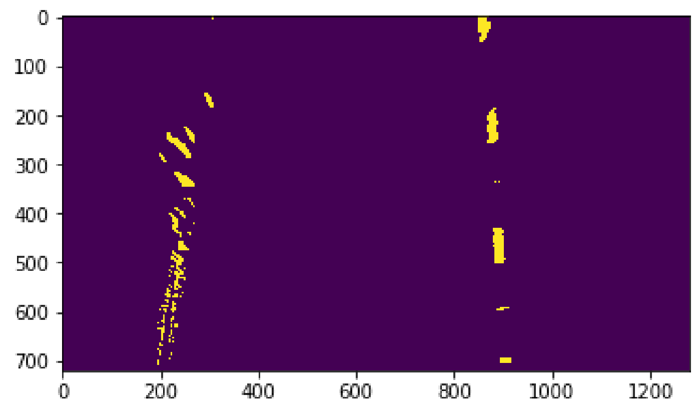
This resulted in the following source and destination points:





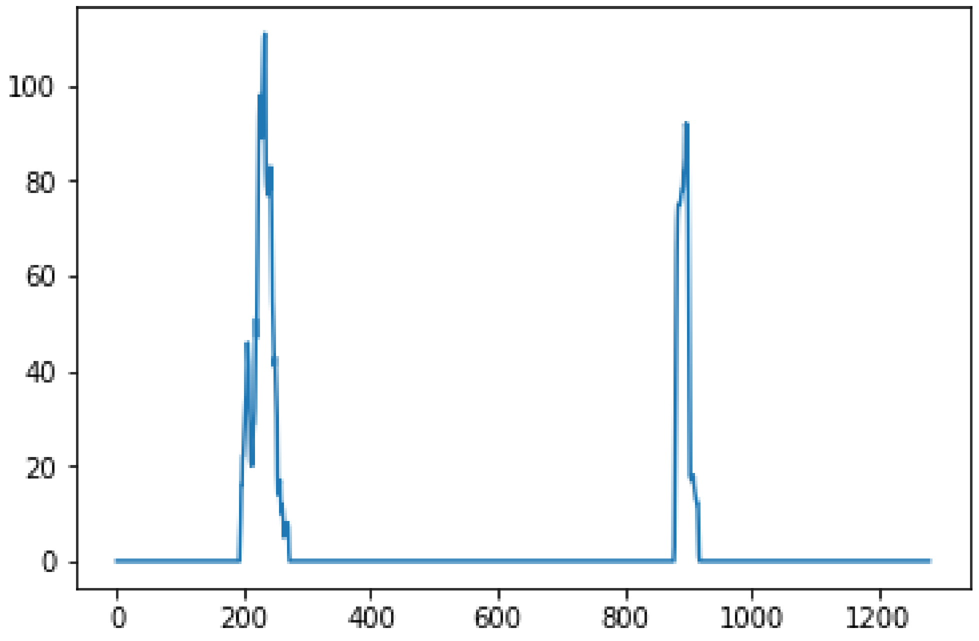
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



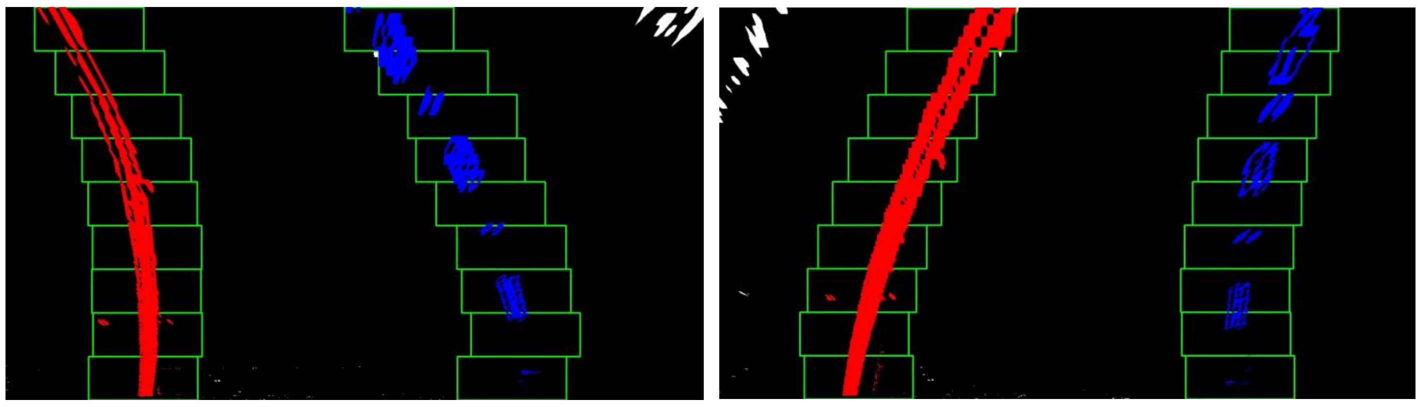


4. Have lane line pixels been identified in the rectified image and fit with a polynomial?

After perspective transform, I used histogram to identify the lane lines, which were the two peaks in the histogram plotted along x direction for a specific slice.



I then choose sliding window approach to get the center of the x points of both left and right lane line. The resultant was as follows



I later used polynomial fitting to draw the curve, by approximating it with a 2nd degree polynomial. The default number of sliding windows was chosen as ‘9’

5. Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?

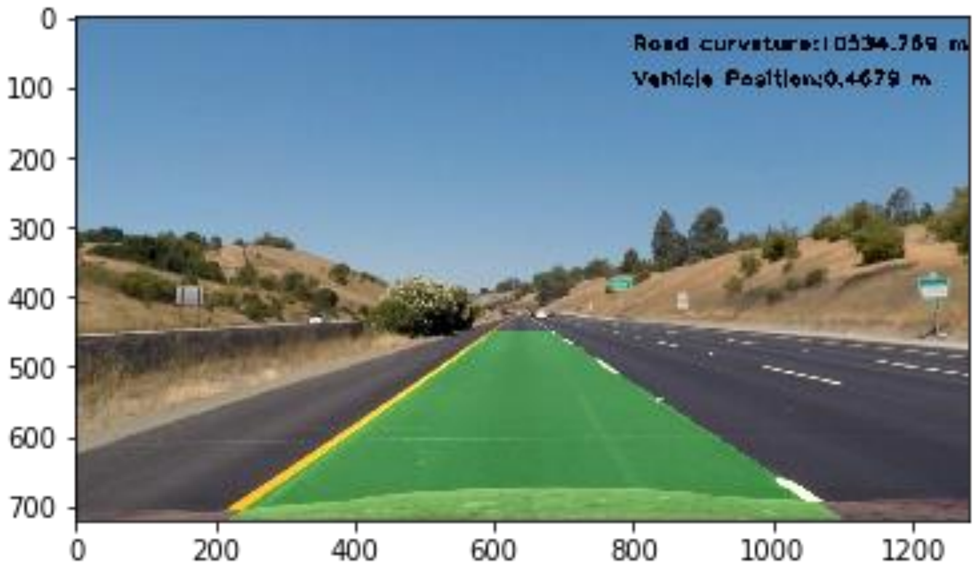
The radius of curvature was estimated using the formula

```
((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
```

And the vehicle positioning was calculated using

```
    lane_center = np.max((left_fitx + right_fitx) /2)

    car_center = 625

    xm_per_pix = 3.7/378

    drift = (car_center - lane_center) * xm_per_pix
```

Then I used OpenCV `putText` function to write the value of curvature and vehicle position on the image.



Pipeline (video)

1. Does the pipeline established with the test images work to process the video?

It sure does! Here's a [link to my video result]

README

1. Has a README file been included that describes in detail the steps taken to construct the pipeline, techniques used, areas where improvements could be made?

You're reading it!

Discussion

The most difficult problem encountered in the whole process was the different lighting conditions in the video. In order to address this issue, we used HLS color space and took only white and yellow color ranges from the image. Yet the algorithm faces difficulty in identifying lane lines at the end points. Also I have fixed the camera view to certain pixels in the image (hard coded). This might not work in some cases like extreme sharp turns or when the elevation of car changes like when the car passes on speed breakers.

The thresholding in the image was also manually set. I believe that dynamic thresholding would work lot better for this kind of scenario where the lane line color, brightness, etc. change quite frequently. In sliding window approach, I used 9 sliding windows. Increase this number might increase computing time but might lead to a better fit. Also using information from previous frame to make the algorithm implicit would give better results.