Camera Calibration

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

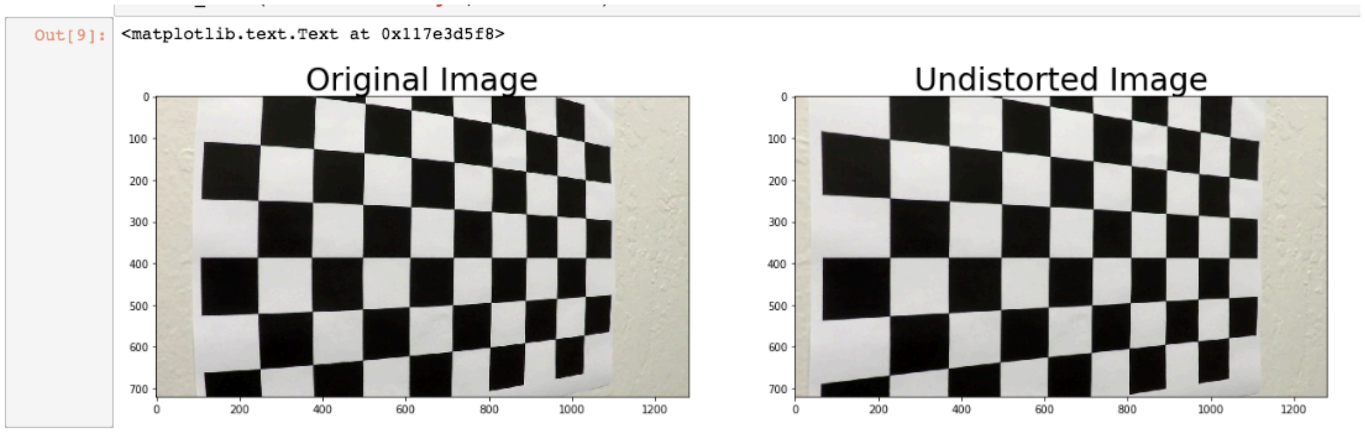
Code for this step is included in the 2nd and 3rd cells of my ipython notebook “Advanced_Lane_ver5 “.

I start by preparing the object points and image points by using several images of a chessboard. Chessboard have a predictable/known pattern, and hence can be used to detect the amount of distortion in the camera, and calibrate for it.

I adjust the dimensions to 6 x 9, convert image to gray scale, then run the cv2.findChessboardCorners to identify the image points, then feedboth to cv2.CallibrateCamera to calculate the camera matrix.

I noticed that some images where not detected by the “findChessboardCorners” function because the photos were taken from extreme angels and cropped several corners from the chessboard.

Next I pick a random chessboard image to test if the camera matrix works good, by calling the “undistort” opencv function and I get the below result.. clearly show distortion corrected, especially if you look at the left edge of the original image.



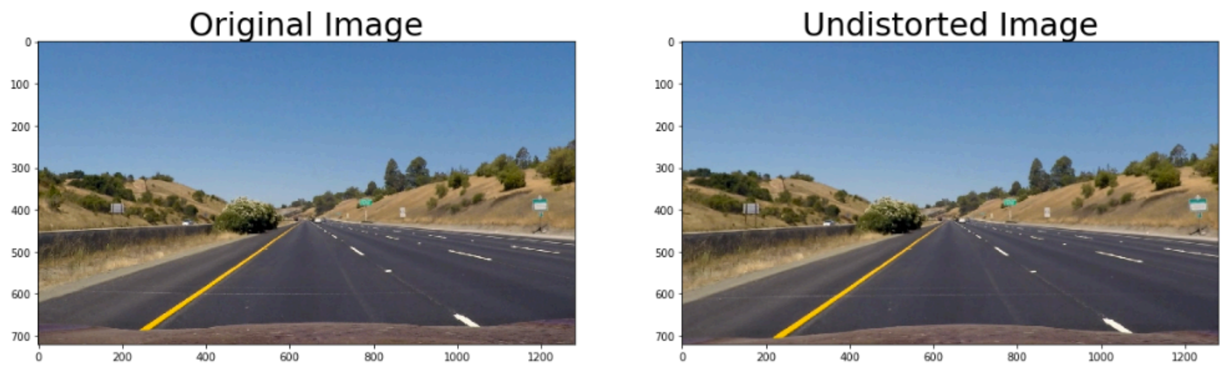
```
Out[9]: <matplotlib.text.Text at 0x117e3d5f8>
```

Pipeline
1- Single images

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

Then I run the undistort function on the test images from a camera fixed in the middle front of a car, and I could see a correction in effect, by just comparing, before and after, the traffic sign at the right of the image, and its distance from the edge of the image.
Also the front of the car looks different at the edges of the undistorted image.

```
Out[11]: <matplotlib.text.Text at 0x11d6797f0>
```



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

I spent some time in this step to gain some intuition from all the available tools we have to detect lines, and get the best result. Therefore, you will find in my ipython notebook, and extensive trial and error attempts to choose the best and cleanest way to detect the lines

This was an intirative process, where I sometimes returned to it even after I proceeded to warping / perspective transform and line detection to get better results.

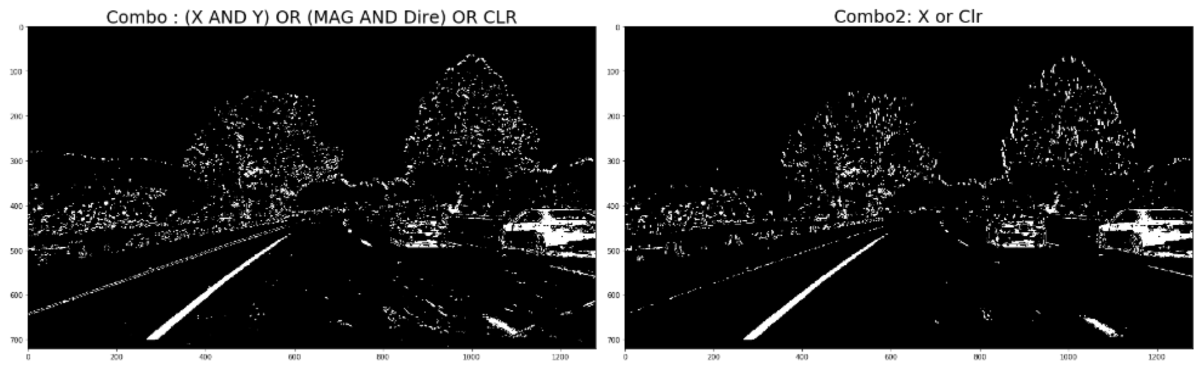
I simply displayed all available ways to detect lines:

- X gradient
- Y gradient
- Magnitude
- Radial / Directional
- Color threshold (S-Channel)

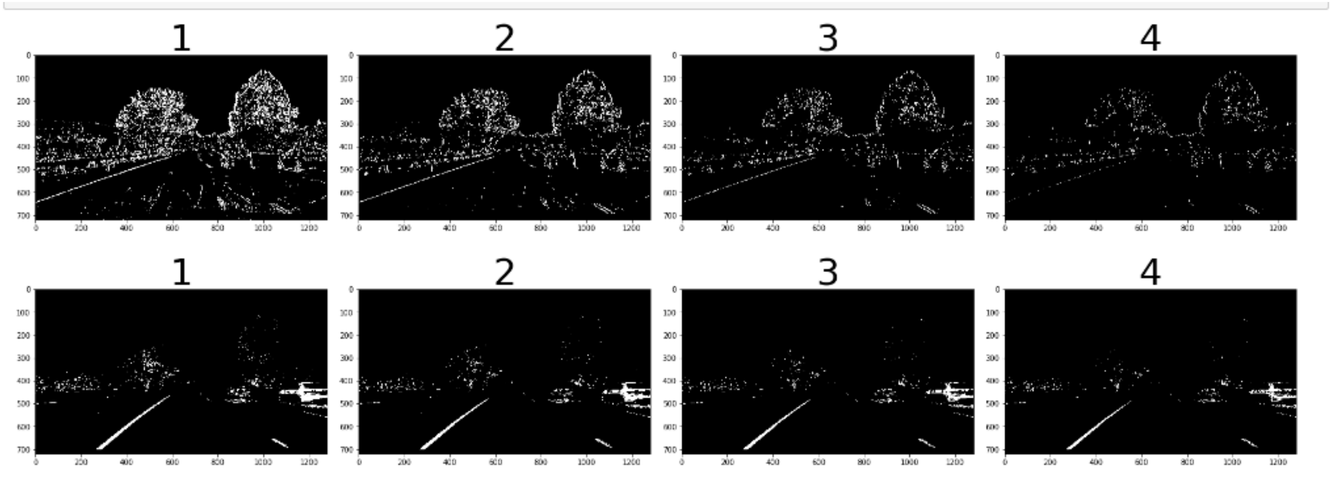
I adjusted:

- Threshold values
- Combination of different methods

Conclusion:
I used a combination of (X gradient) OR (Color threshold), as I found that this had the cleanest output, and complemented each others detection capability.



I implemented many threshold values, and judged the best one by just viewing them with my bare eye



```
img_sobelx    = abs_sobel_thresh(testimage,'x',9,(50,100))
S, img_clr    = clr_threshold(testimage,(150,255))
img_1[((img_sobelx == 1)  | (img_clr == 1))] =1
```

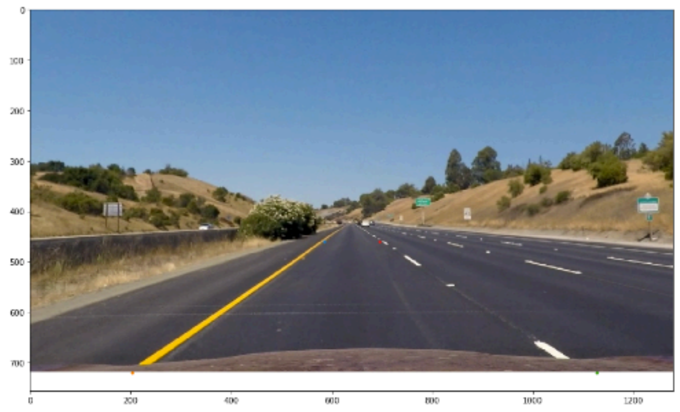
3. Has a perspective transform been applied to rectify the image?

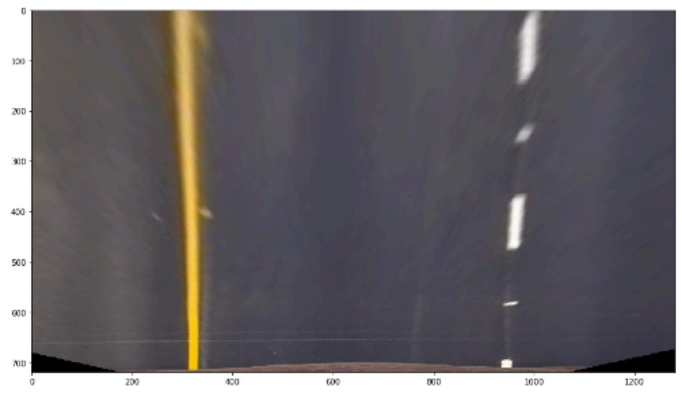
I spent some time here to manually choose the source and destination points to perform the perspective transform. And then my mentor suggested I use the method defined in the answer template to have a consistent result.

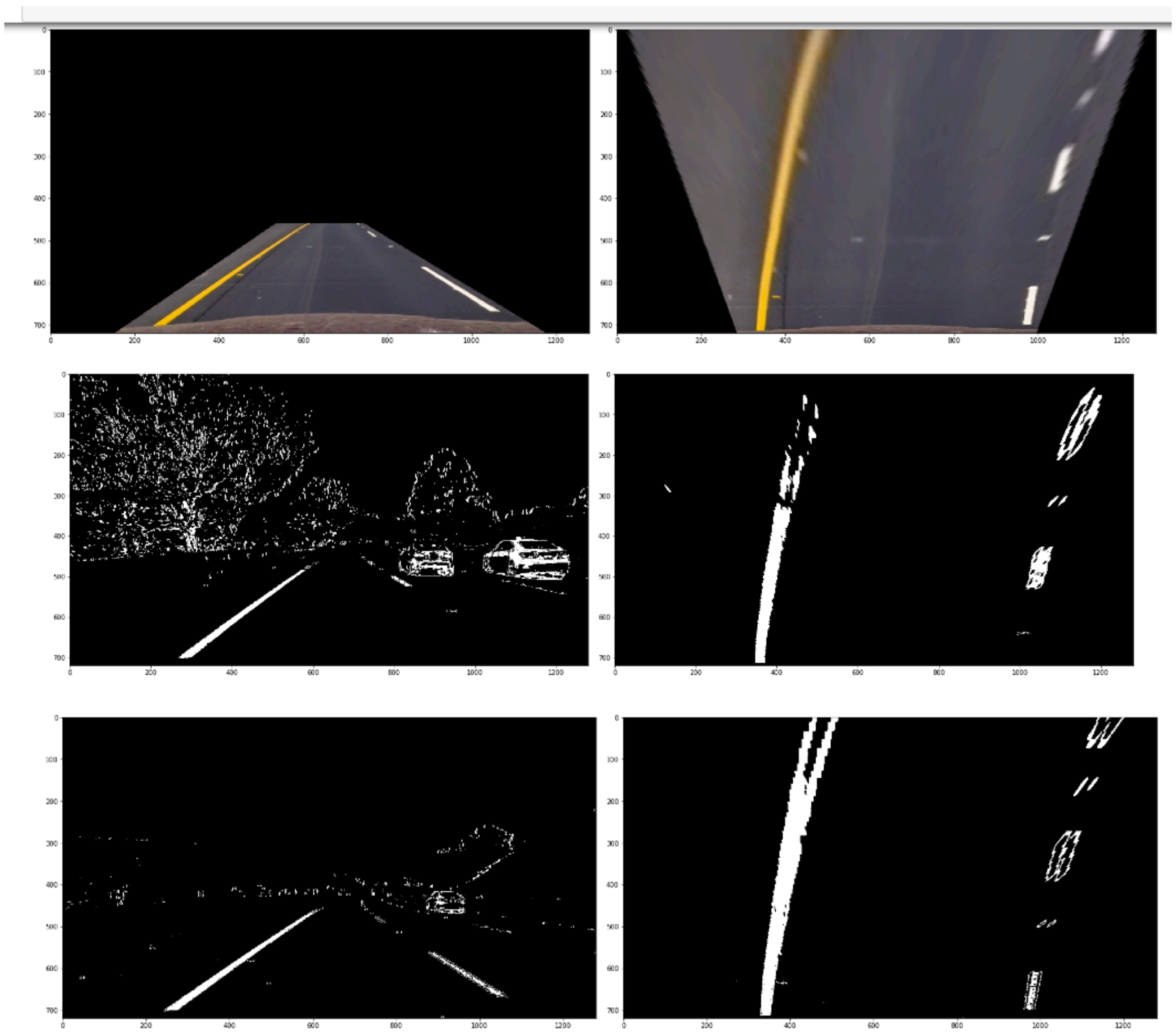
```
src = np.float32(
    [[585,460],
    [203,720],
    [1127,720],
    [695,460]])
    dst = np.float32
    [[320,0],
     [320,720],
     [960,720],
     [960,0]])
```

In addition to this, I performed image masking to cancel out noise coming from other parts of the image. Such noise had a negative impact on detecting and fitting the line later-on. So I came back to this step and peformed masking

I took a margin left and right of the specified source points and used them as vertices for the masking function. Below are some results



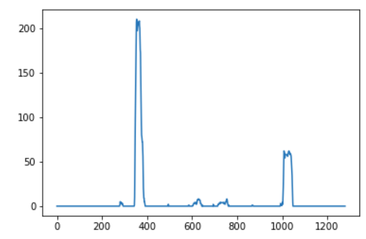
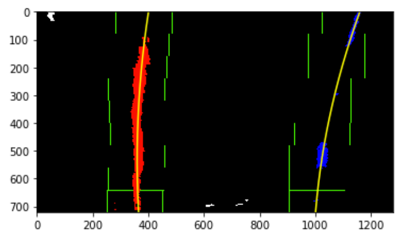
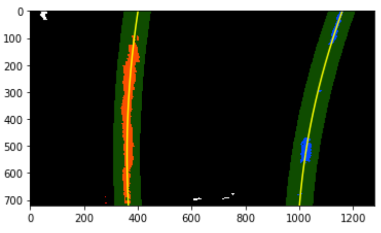
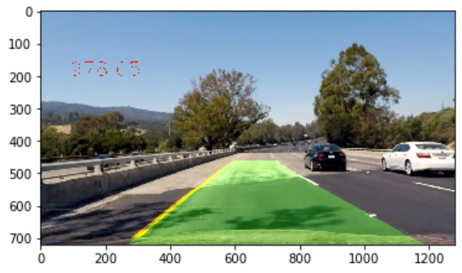




4. Have lane line pixels been identified in the rectified image and fit with a polynomial? Radius and Offset

After successfully implementing a perspective transform, I applied the sliding window method to detect and fit the detected lane lines following steps:

1- Sliding window: starts with a histogram to detect the x values of the lines, then move up the image step by step to detect the movement of the x value representing the center of the line.
2- Fitting a second degree polynomial to the detected lines
3- Calculating X-values in order to plot the line.
4- Calculate the radius and offset (distance between center of the car, i.e. center of the image, and the center of the lane.

In my ipython notebook, you will find that I tested my code on the test4.jpg image, and got the above results..

Radius = 1227 m
Offset = 0.2 m to the right of the center of the lane (positive value)

I calculated the radius in the pixel world then an approximation to the real world.

Radius of curvature equation at any point x in equation x=f(y): $R_{curve} = (1+(2Ay+B)^2)^{3/2} / |2A|$
And we perform the calculation at the bottome of the image : y_eval = np.max(ploty)

I already calculated the midpoint of the image and the center of the lane while using histogram function … so I calculating the offset by subtracting “Lane center” – “car center”. If result is +ve .. then we are closer to the right lane.

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 on git hub

Before applying my pipeline to the video, I had to re-organize my code into smaller functions and created a class to keep history of the information gathered about the lines for each scene.

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

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

When implementing the pip-line for the video I had to consider the following:

1- How to get a smooth detected lines, with slowly changing radius in curves: Here I considered giving higher weighted average of previous predictions in the system at (t-1)
   a. alpha * x_fitted (t-1) + (1-alpha)*x_fitted (t)
   b. Giving more weight to previous predictions, will make the system less vulnerable to rapid changes cause by new data points coming in.
   c. Eventually this system should converge towards the right solution, based on the weight.

2- Sanity check of predicted lines:
   a. I used the distance between the 2 lane lines as sanity check … if the distance change by more than 30% from one frame to another, then this data point is dropped … this is an indication off a bad detection of the lanes

3- Radius, offset and drawing the lines:
   a. Instead of performing calculation on the latest fitted line, I did it on the best fitted line. And after removing unwanted line detections after sanity check, the function produced more stable results.

4- I use sliding window method to detect new line, only when I fail to detect a line, or when I discard a line in the previous frame. The sliding window method is very heavy on the processing.