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.

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](https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

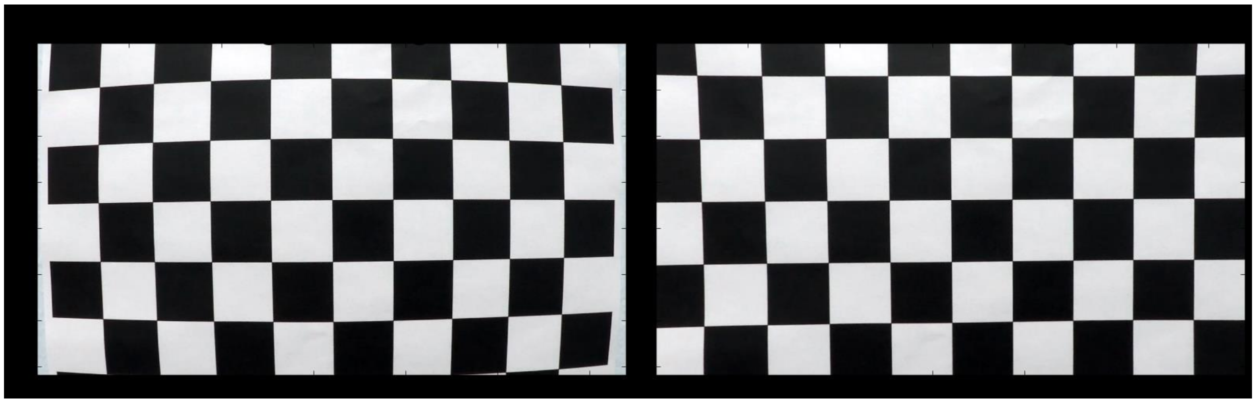
You're reading it! I use PDF generated with MS Word for simplicity attaching images. Less time doing markup, more time learning! 😉

2. Camera Calibration The code used is mostly from https://github.com/udacity/CarND-Camera-Calibration by Ryan Keenan

The code is at first cell of Advanced_lane_lines_code.ipynb

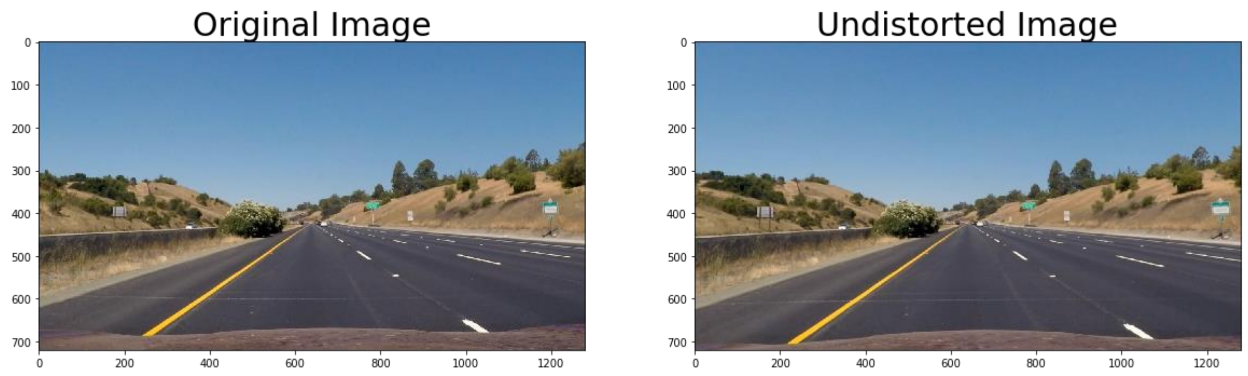
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of 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.

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:



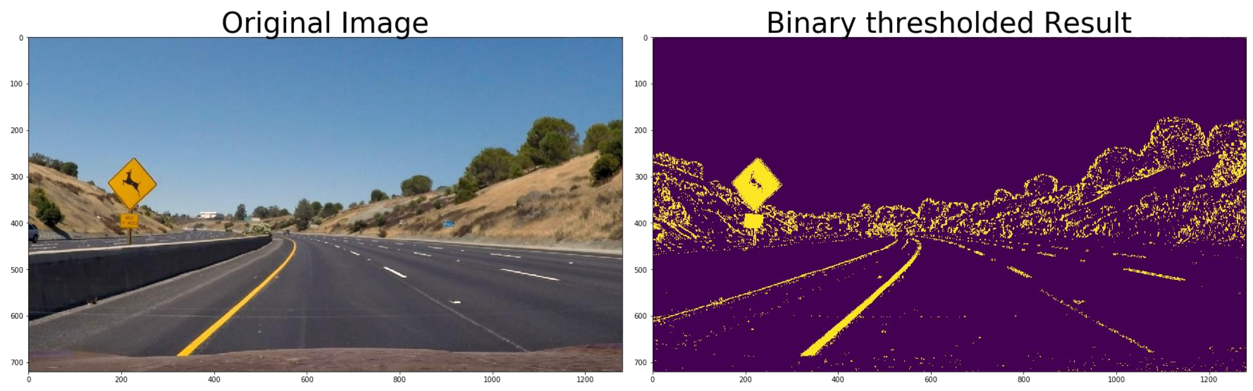
1.Provide an example of a distortion-corrected image.

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 color and gradient thresholds to generate a binary image (thresholding steps at sobel_schannel_thresholds function in `Advanced_lane_lines_code.ipynb`). Here's an example of my output for this step:



I used S channel and Sobel X gradient as suggested. Since filtering is just the basic to make it work, challenge video for example does have problems distinguishing the shadow (which passes X Sobel gradient filter) and the yellow lane.

I think this is the more workable area of this project in order to increase performance.

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

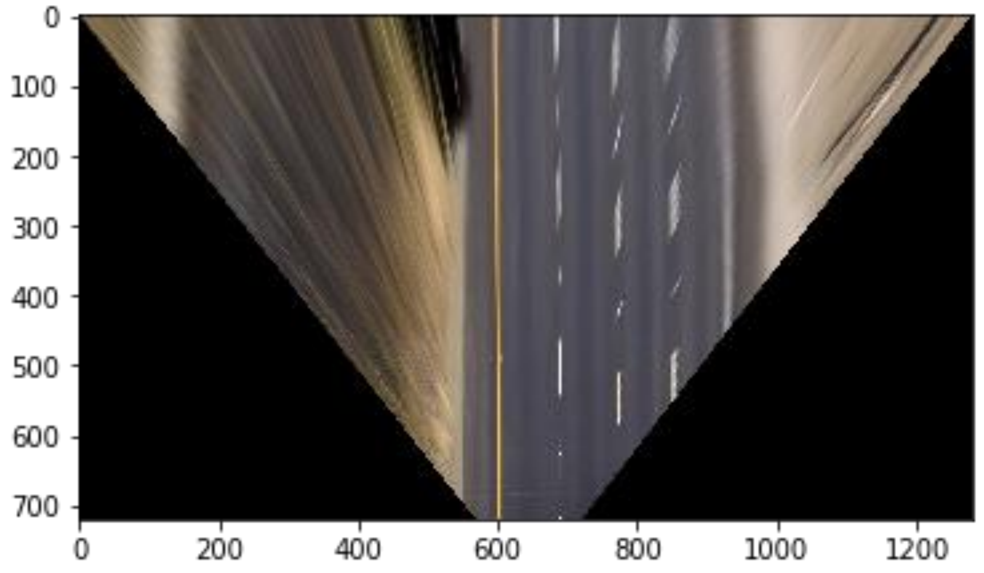
The code for my perspective transform includes a function called `warp_birds_eye ()`, which appears in part 2 in the file ` Advanced_lane_lines_code. The ` warp_birds_eye ()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points and a Boolean to choose to invert the transformation. I chose the hardcode the source and destination points in the following manner:

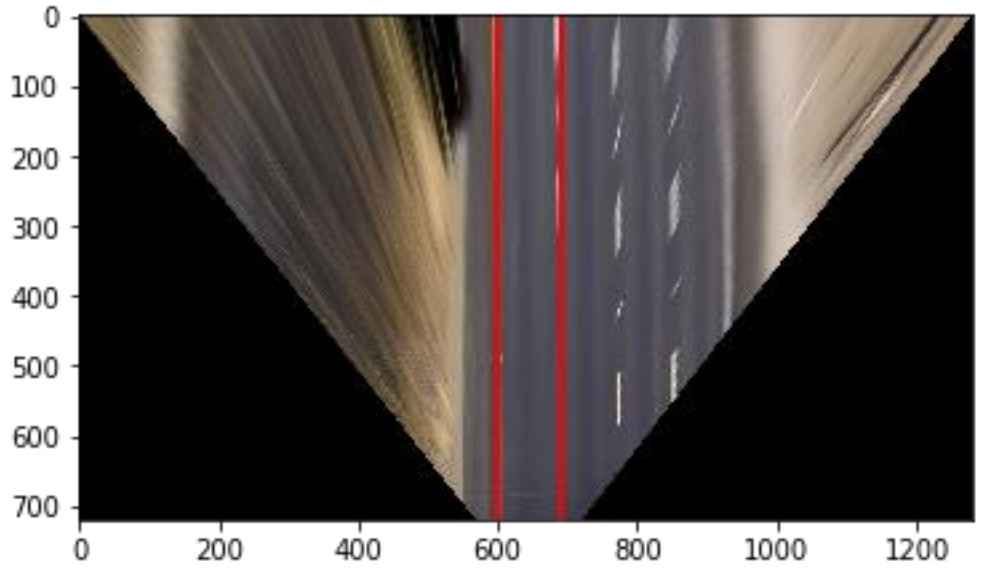
```
sourcepoints = np.float32([[0, 673], [1207, 673], [0, 450], [1280, 450]])

destinationpoints = np.float32([[569, 720], [711, 720], [0, 0], [1280, 0]])
```

This resulted in the following source and destination points:

Source	Destination
0 , 673	569,720
1207,673	711,720
0,450	0,0
1280,450	1280,0

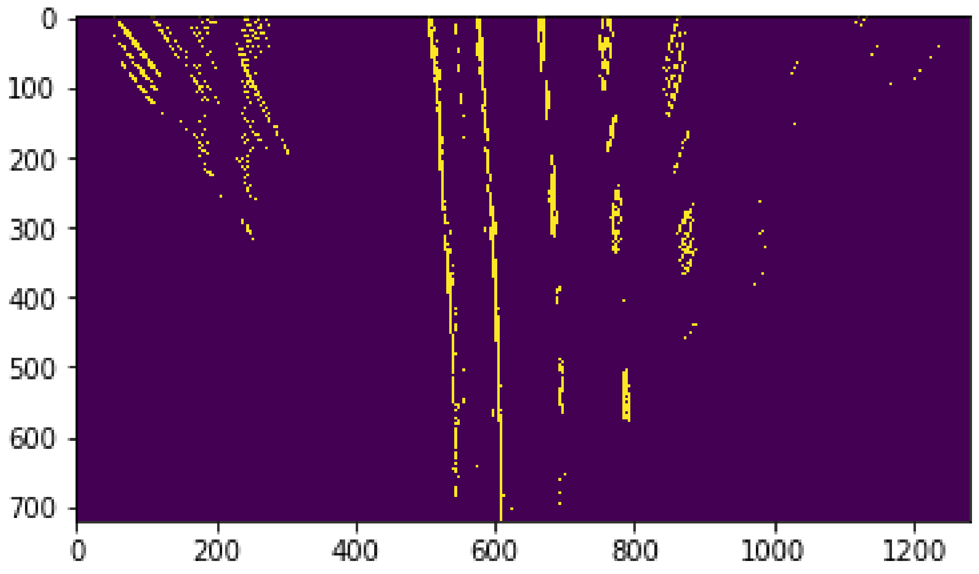
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

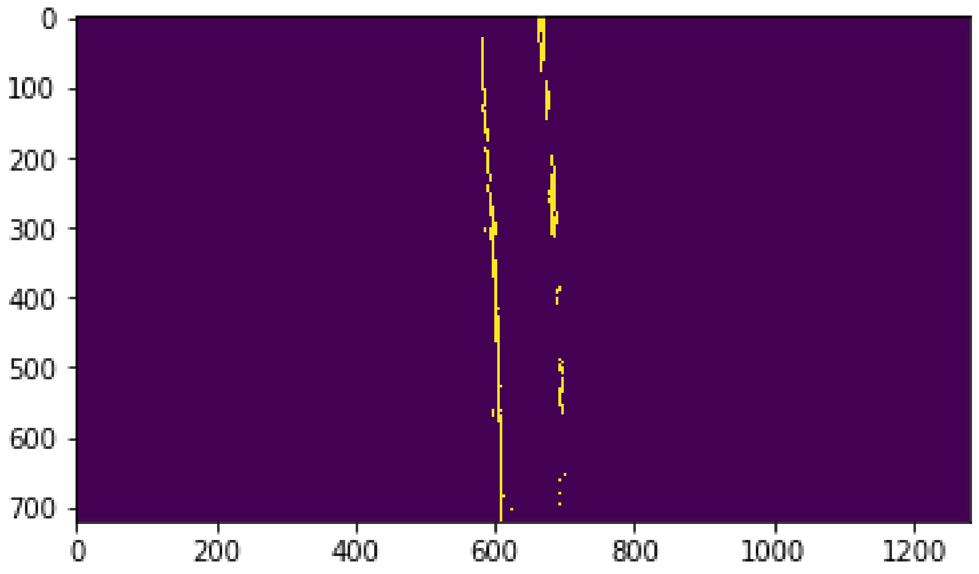




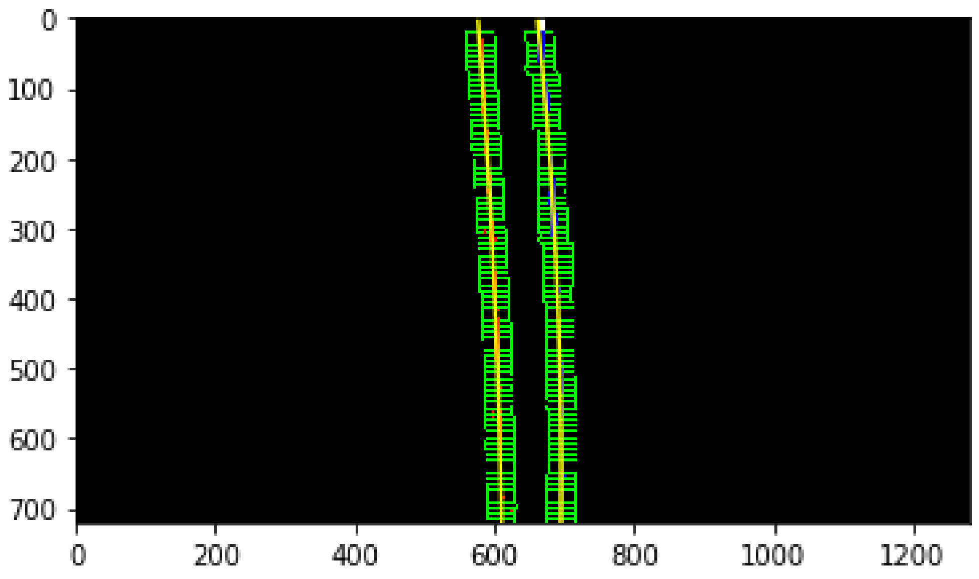
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I framed the region of interest to avoid detecting the neighboring lane lines.





I used the sliding window technique. Since my transformation gives very long lines I used a lot of smaller windows.



Then I selected non-zero pixels and bucket into data structures. Then I fitted a standard second order polynomial.

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.

I did this at the function pipeline_adv_lane_lines_finding at Advanced_lane_lines_code.ipynb

For curvature radius I used suggested implementation at Udacity lesson.

```
# Fit new polynomials to x,y in world space
   left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
   right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)
   # Calculate the new radio of curvature
   left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
   right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
   radius = (left_curverad + right_curverad)/2
```

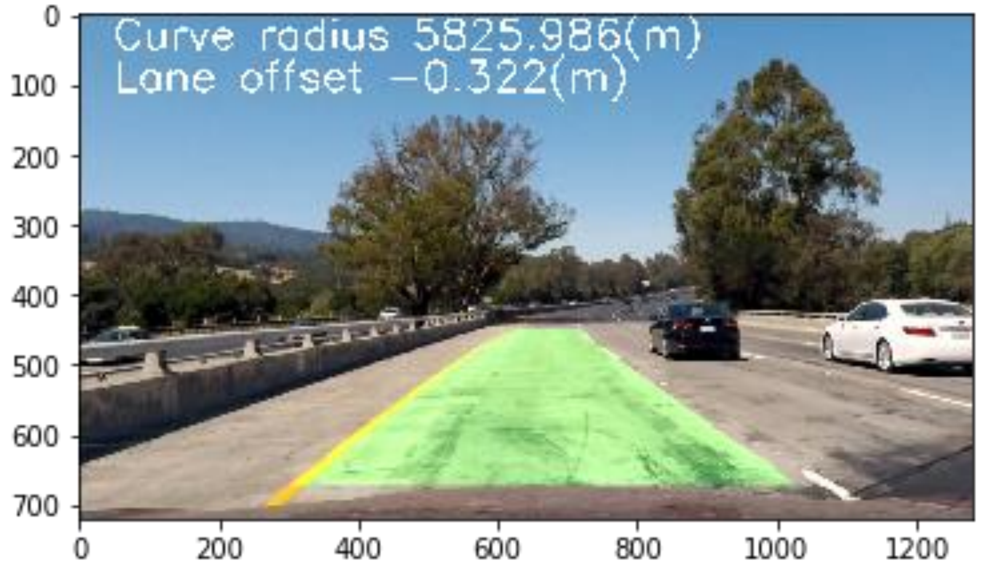
The code snippet uses calibration from pixel to meters to bring the second order polynomial to have meters units. Then uses some geometry math to calc the radius at the middle point.

Inner and outer lanes have slightly different radius since (they should have around 3 meter more). Since this is not very accurate, this will differ far more, and I just use the average between the two measures to display.

For the center lane deviation, I calculated the line polynomials at where the hood is, then I calc the median and substract it from middle point. At that point you have the deviation in pixels, so you have to apply the pixels to meters approximation.

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 lines # through # in my code in ` Advanced_lane_lines_code.ipynb` in the function ` pipeline_adv_lane_lines_finding ()` at section number 7. 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!).

See my github directory CarND-Advanced-Lane-Lines/test_videos_output

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?

I face some difficulties understanding how different colours space filterings affect the image. And since a intuition is difficult to build with those, I had to do a lot of tests.

There is plenty of room at that part of the project, my code is not filtering enough shadows, and not distinguishing only white and yellow lines perfectly. Also different contrasts affect a lot more than expected.

With steeper curves pipeline will fail too.

Radius measuring is very bad, is terrible and curve approximation polynomials maybe can be substituted with other geometry entity or model that civil engineers tend to use when building highways.