Advanced Lane Finding Project Write-up

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

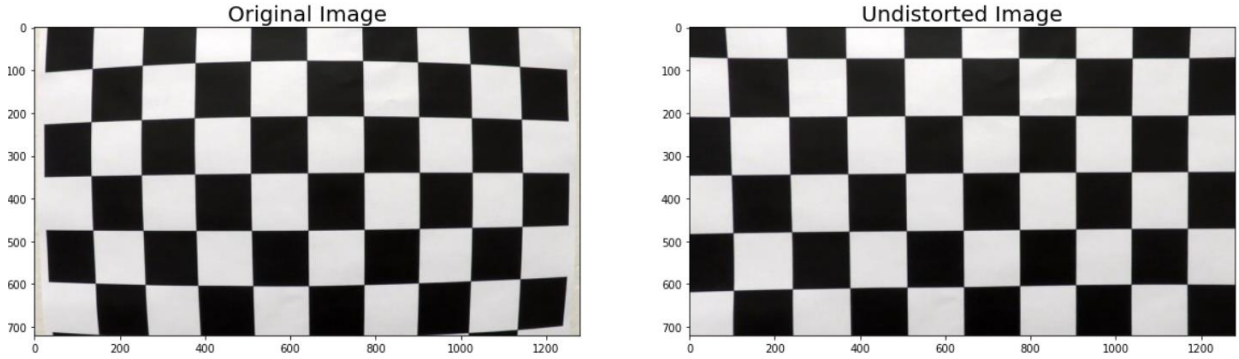
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the first and second code cell of the IPython notebook located in "Advanced_lane_finding_project_code_final.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.

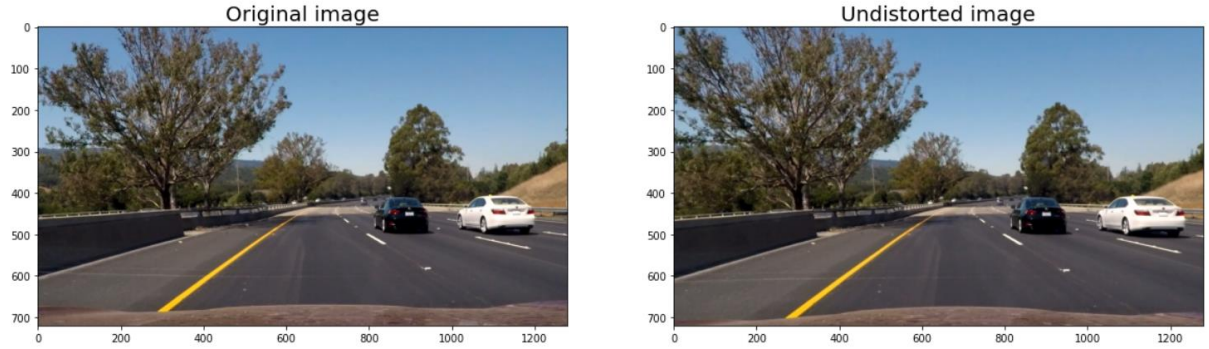
I then used the output `objpoints` and `imgpoints` to compute the camera calibration matrix and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

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

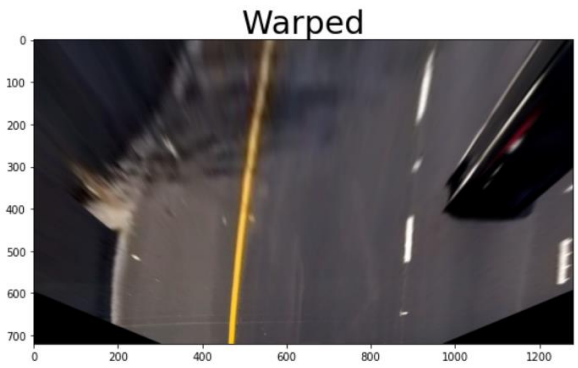
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



I have dumped the `camera calibration matrix (mtx)` and `distortion coefficients(dist)` to a pickle file `calibration.p` from camera calibration. I have created the function `cal_undistort()` function which contains `cv2.undistort()`. The `cal_undistort()` function takes image as a input(`img`), and returns undistorted image.

***Note**: I have done perspective transform first and then I did color transforms and gradients for creating binary image in IPython notebook. So I follow the same order here too.*

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 appears in the 4th code cell of the IPython notebook. The `perspective_transform()` 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([(580,460),
                  (703,460),
                  (262,675),
                  (1036,675)])
dst = np.float32([(440,0),
                  (w-440,0),
                  (440,h),
                  (w-440,h)])
```

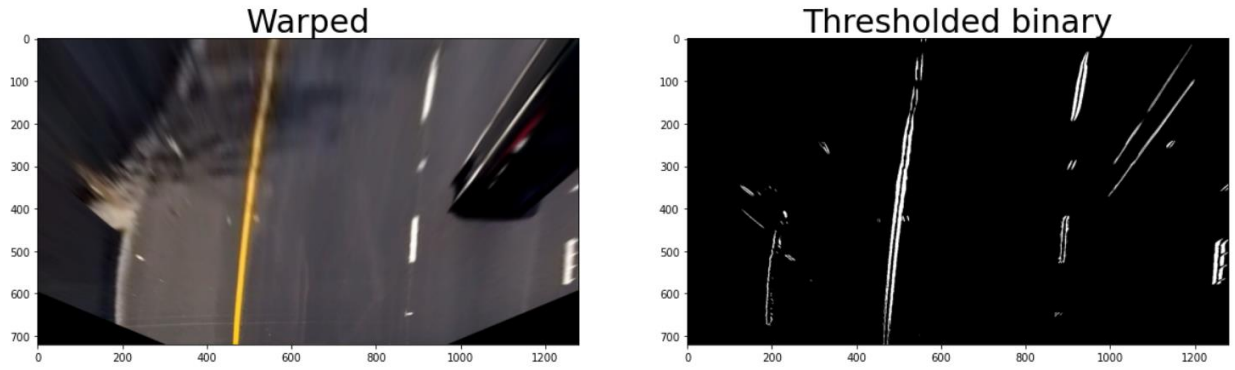
This resulted in the following source and destination points:

Source	Destination
580, 460	440, 0
703, 460	840, 0
262, 675	440, 720
1036, 675	840, 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.

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. The code for this step is contained in the sixth code cell of the IPython notebook located in "Advanced_lane_finding_project_code_final.ipynb" Here's an example of my output for this step.

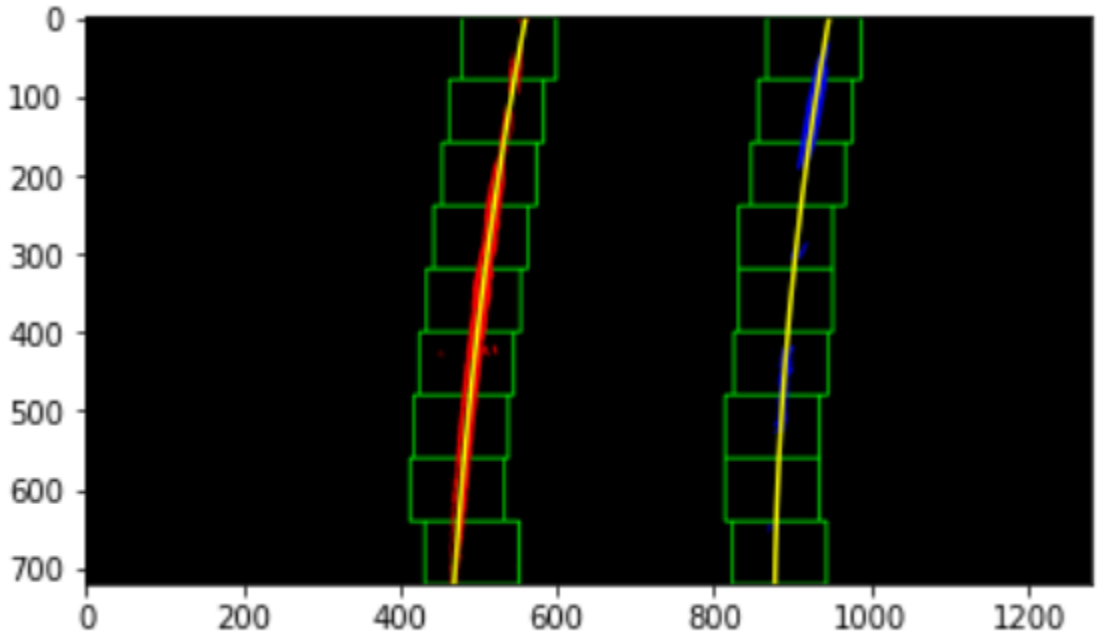


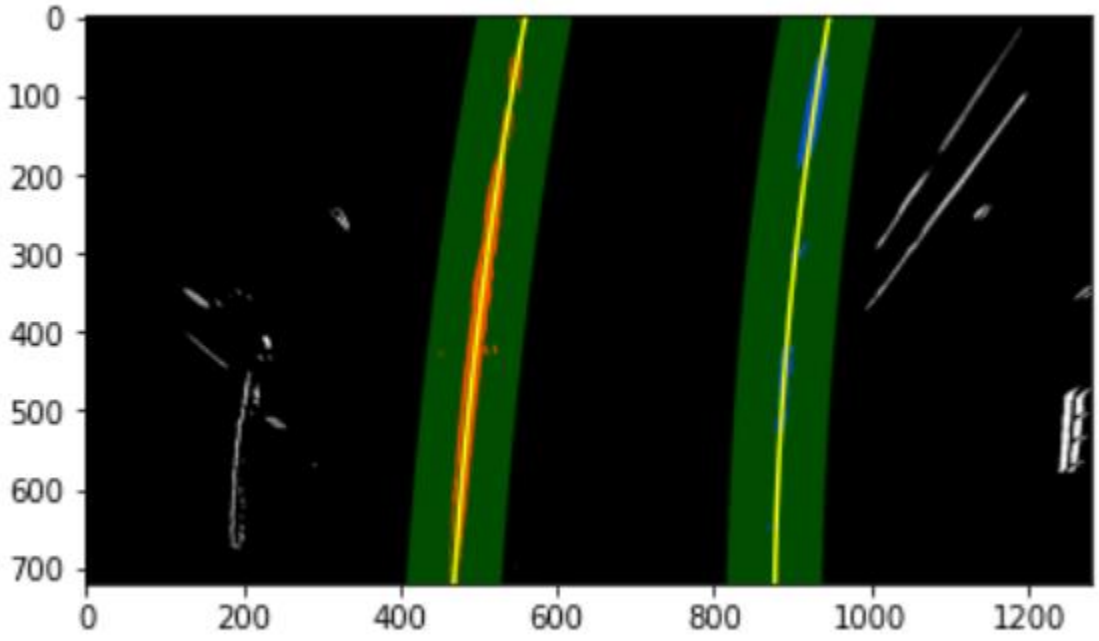
I've created `threshold_binary_img ()` function to which I pass an image to be perspective transformed. In case of gradient threshold, first I convert RGB image to GRAYSCALE using `cv2.cvtColor()` function. Then I take derivative in x-orient using `cv2.Sobel()` function. I have taken threshold value as `sobel_x_thresh = (25,127)`

In case of color transform I've converted RGB image to HSV color space and used V channel for color transform. I have taken threshold value as `hsv_v_thresh = (128,255)`.

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

I've created `find_lane_pixels()` and `search_around_poly()` function for identifying lane-line pixels and the code for this step is contained in the 9th code cell of the IPython notebook located in "Advanced_lane_finding_project_code_final.ipynb". Under `find_lane_pixels()` function, I've implemented sliding window search method for finding lane-lines in the first frame of video and $f(y) = Ay^2 + By + C$ is the polynomial equation used for fitting lane-lines. For other consecutive frames, `search_around_poly()` function will be called and under this I've searched around the left and right side of polynomial calculated from sliding window with margin = 60. Here is an example of my result on a test image:





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've created `measure_curvature_real()` function for calculating the radius of curvature of lane & position of the vehicle w.r.t center and the code for this step is contained in the 14th code cell of the IPython notebook located in "Advanced_lane_finding_project_code_final.ipynb".

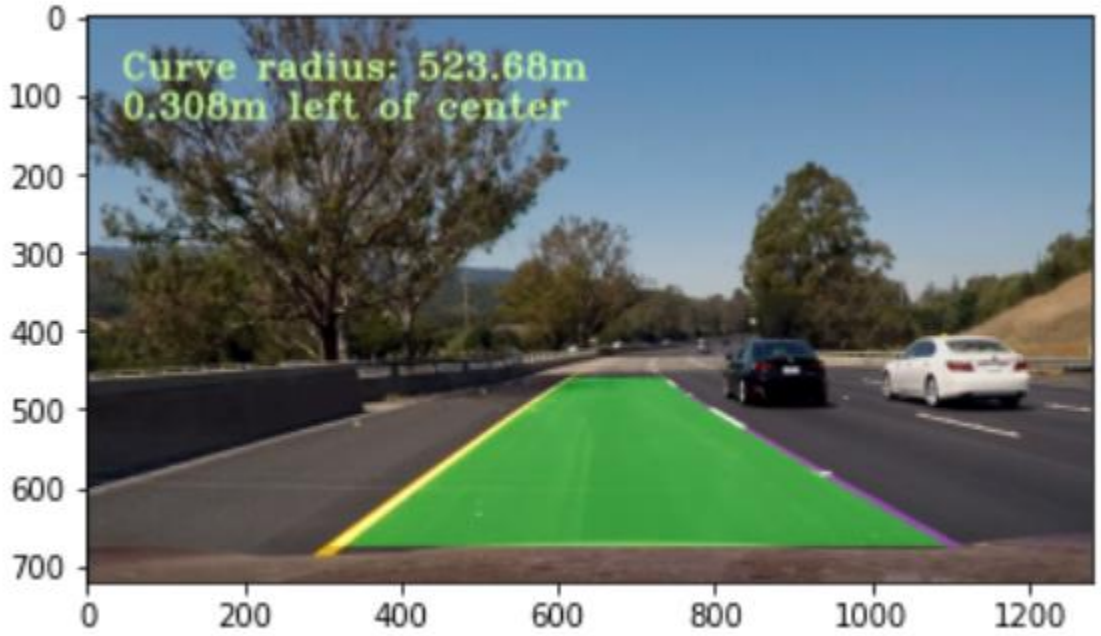
I used the equation for calculating radius of curvature = $Rcurve = \frac{(1 + (2Ay+B)2\ )3/2}{|2a|}$

In the above equation I've converted parameters from pixel space to real world space by multiplying constants. ym_per_pixel = 3.048/100 and xm_per_pix = 3.7/400, where 3.048 is the lane length in meters and 3.7 is the lane width in meters.

In case of finding position of the vehicle with respect to centre, I took average of left-lane x-intercept and right-lane x-intercept and subtracted it with midpoint of the image's horizontal axis.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I've created `merge_lane_detected()` and `write_data()` function for drawing detected lane& radius of curvature and the code for this step is contained in the 16th and 18th code cell of the IPython notebook located in "Advanced_lane_finding_project_code_final.ipynb". Under `merge_lane_detected()` function, I've used `np.dstack()` for depth stacking of binary image and used `cv2.fillPoly()` for plotting lane area in green. Also I've used `cv2.polylines()` for highlighting lane lines in yellow and purple. Finally using `cv2.addWeighted()` lane is drawn onto the test image. Similarly under `write_data()` function, I've used `cv2.putText()` for writing the data onto the image. 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!).

I have attached the "`project_video_output`" in this folder and you can also find in jupyter notebook.

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've followed the course instructions until perspective transform. In gradient and color transform, I've explored different color spaces such as HSV, HLS, RGB and finally made use of HSV and Sobel X-gradient. I spent quite a lot of time in tuning these parameters since I faced issues in different lane lighting conditions. While thresholding lane lines, I could find other objects such as car whose edges is also detected and interfering with lane lines. In order to eliminate that, I've searched for lane lines from midpoint to either side of quarter of the image in sliding window polyfit. By this I could able to eliminate the 50% of unwanted pixels generated from other objects on the road. The pipeline might fail in conditions where lighting is very poor and thresholding needs to be more robust and in order to improve it I going to explore more color spaces such as LAB, etc and going to test the pipeline on the video capture on my vehicle at different lighting conditions.