Writeup

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.

You're reading it!

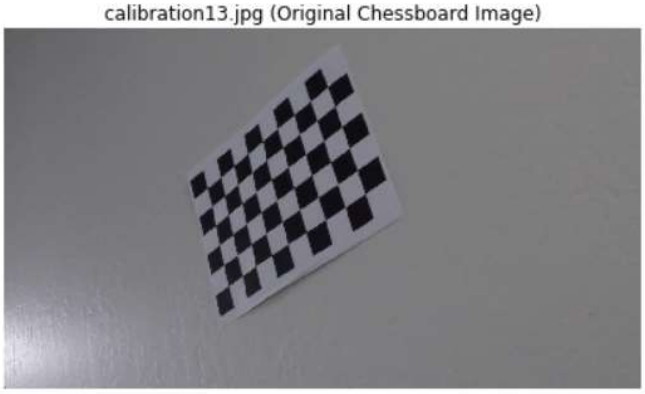
Camera Calibration

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 2nd and 3rd code cells of the IPython notebook located in "./P2.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, `objpoints` is just a replicated array of coordinates, and `objp_list` will be appended with a copy of it every time I successfully detect all chessboard corners(using `cv2.findChessboardCorners()` method) in a test image. `imgp_list` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

calibration13.jpg (Original Chessboard Image)

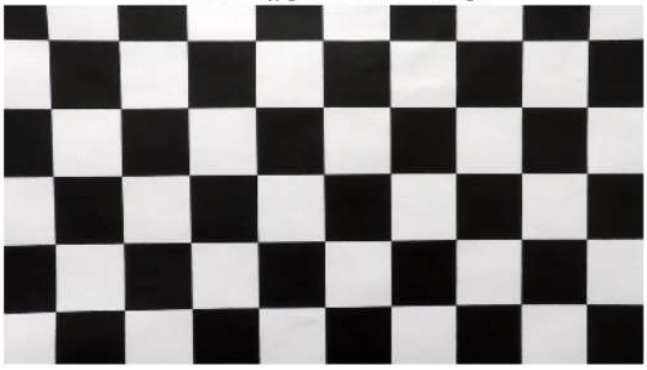


calibration13.jpg (Draw Chessboard Corners)

I then used the output `objp_list` and `imgp_list` 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(Please refer to the 5th and 6th code cell of P2.ipynb):

calibration1.jpg (Original Image)

calibration1.jpg (Undistorted Image)



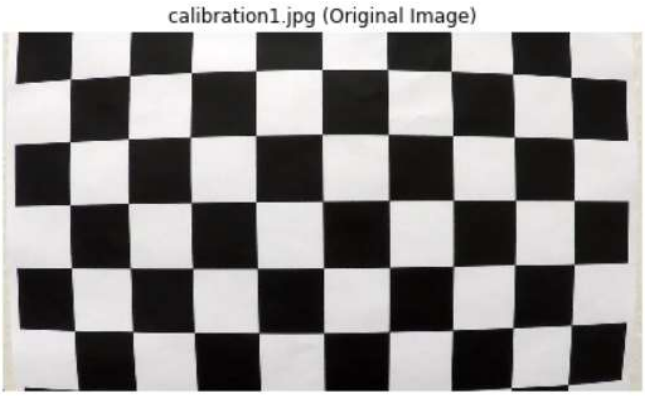
straight_lines1.jpg (Original Image)

straight_lines1.jpg (Undistorted Image)

Pipeline (single images)

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

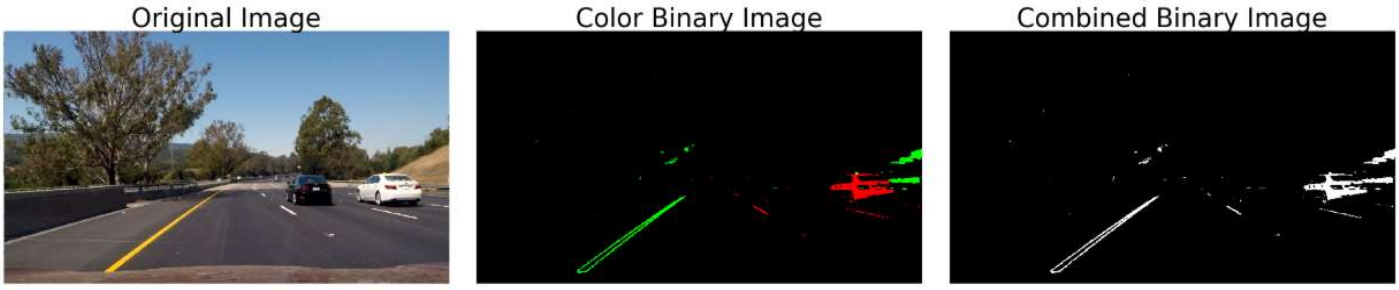
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



Using the `cv2.calibrateCamera()` functionto compute the camera calibration and distortion coefficients, and then applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result

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 LUV colorspace and LAB colorspace to generate a binary image (thresholding steps at 7th and 8th code cells in `P2.ipynb`). The L channel from LUV with lower and upper thresholds around 225 & 255 respectively works very well to pick out the white lines, even in the parts of the video with heavy shadows and brighter pavement. And the b channel from Lab which does a great job with the yellow lines (you can play around with thresholds around 155 & 200). Here's an example of my output for this step.



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

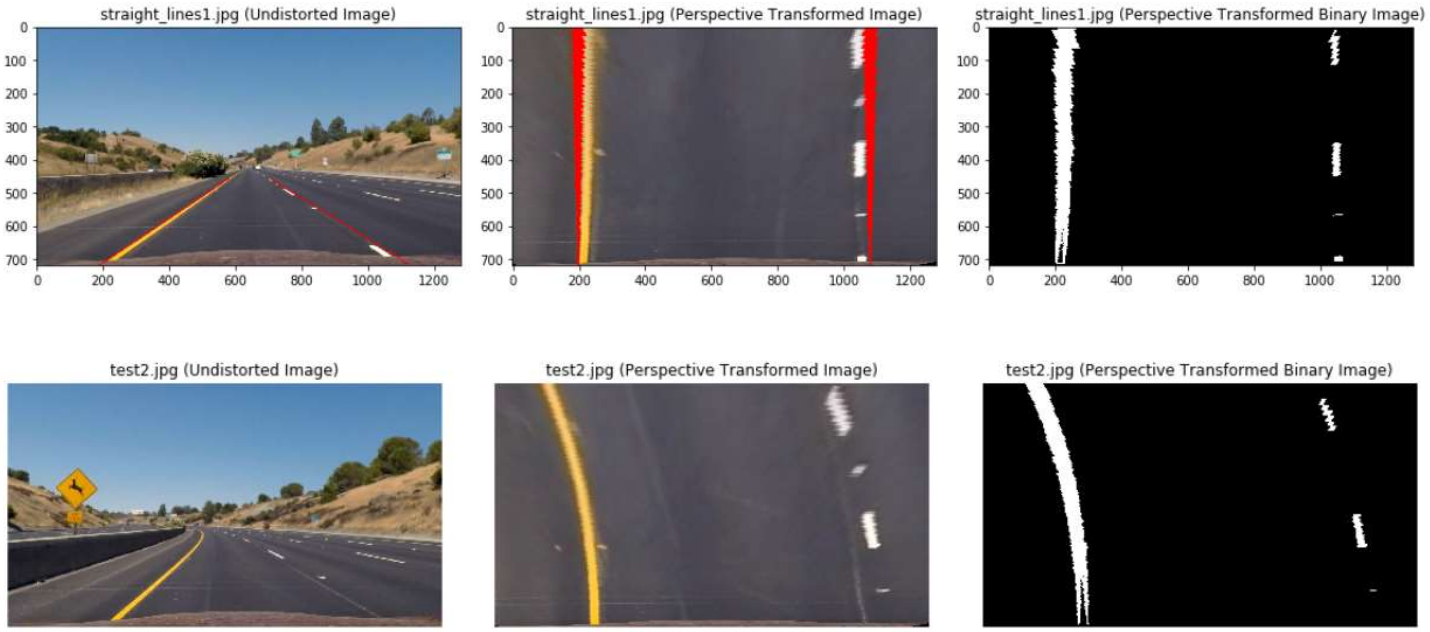
The code for my perspective transform includes a function called `Image_Warper()`, which appears in 9th code cell in the file `P2.ipynb`. The `Image_Warper()` function takes as inputs an image (`img`), as well as source (`src_points`) and destination (`dst_points`) points. I chose the hardcode the source and destination points in the following manner:

```
src_points = np.float32([[190, 720], [578, 460], [706, 460], [1130, 720]])
dst_points = np.float32([[200, 720], [200, 0], [1080, 0], [1080, 720]])
```

This resulted in the following source and destination points:

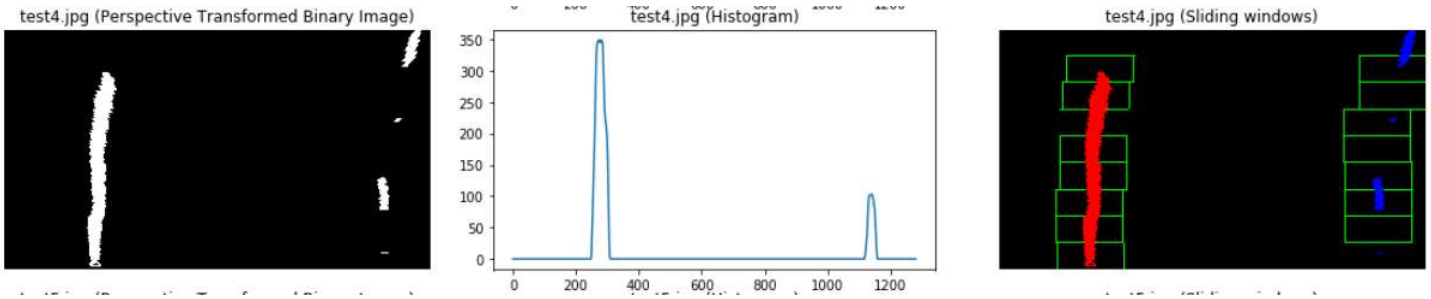
Source	Destination
190, 720	200, 720
578, 460	200, 0
706, 460	1080, 720
1130,720	1080, 720

I verified that my perspective transform was working as expected by drawing the `src_points` and `dst_points` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image (Please refer to the 9,10,11,12 code cells in `P2.ipynb`):



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

Then I did some other stuff(using histogram and sliding windows technique) and fit my lane lines with a 2nd order polynomial kinda like this (Please refer to the 13,14,15,16,17 code cells in `P2.ipynb`):



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 in the 18,19,20,21,22 code cells in `P2.ipynb`. The image below is an example. I define the conversions in x and y from pixels space to meters as below: I assume the camera is mounted at the

center of the car, such that the lane center is the midpoint at the bottom of the image between the two lines you've detected. The offset of the lane center from the center of the image (converted from pixels to meters) is the distance from the center of the lane.

```
# Define conversions in x and y from pixels space to meters
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/880 # meters per pixel in x dimension
# determine the curvature of the lane in real world (unit: m)
def measure_curvature_real(left_fit_cr, right_fit_cr, ploty,ym_per_pix = 30/720):
    # Calculates the curvature of polynomial functions in meters.
    # Define y-value where we want radius of curvature
    # We'll choose the maximum y-value, corresponding to the bottom of the image
    ploty_cr = ploty * ym_per_pix
    y_eval = np.max(ploty_cr)
    # Implement the calculation of R_curve (radius of curvature)
    left_curverad = (1+(2*left_fit_cr[0]*y_eval + left_fit_cr[1])**2)**(1.5) /np.absolute(2*left_fit_cr[
    right_curverad = (1+(2*right_fit_cr[0]*y_eval + right_fit_cr[1])**2)**(1.5) / np.absolute(2*right_fi
    return left_curverad, right_curverad
def measure_offset_to_center(ploty, left_fit, right_fit, xm_per_pix = 3.7/880):
    y_max = np.max(ploty)
    left_x_coor = left_fit[0] * y_max**2 + left_fit[1] * y_max + left_fit[2]
    right_x_coor = right_fit[0] * y_max**2 + right_fit[1] * y_max + right_fit[2]
    lane_center_px = 1280 / 2
    center_offset_img_space = (left_x_coor + right_x_coor)/ 2 - lane_center_px
    center_offset_real_world = center_offset_img_space * xm_per_pix
    return center_offset_real_world
```

Then I use `cv2.putText()` to put the measurment onto the image as below:

test3.jpg (Measure Radius of Curvature and offset to lane center)

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 23,24,25,26 code cells in `P2.ipynb` file in the function `Plot_Lane_Lines_Single_Image()`. Here is an example of my result on a test image:

test5.jpg (Final result)

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!).

Here's a link to my video result

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?

- There are some outliers bothering me and my algorithm maybe fail in some bad lightning coditions, especially in challange_video.mp4.
- I need to do some outliers rejection to avoid the wobbly lane lines in the future.
- In order to improve the efficiency of my codes, I think I should add some lane lines' confidence coefficients to reuse the lane line history, instead of just searching blindly for the lane lines in

each frame of video later.
- In order to better deal with varying lightning conditions, I will explore the deep learning techniques to find the lane lines. I think deep learning will work better if I collect enough good data.