Advanced Lane Finding

Write up of work done for Udacity CarND Advance Lane Finding Project P4, by Mani Srinivasan

Project Objectives

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!

Note: The code for the entire project was developed in Jupyter and the images and videos are embedded as part of the notebook “./Advanced-Lane-Line-Detection-P4.ipynb’. Each step of the code is marked up for easy understanding of the flow and the pipeline with images to visualize the intermediate results. I used the example code given in the lecture and added my own to complete the work.

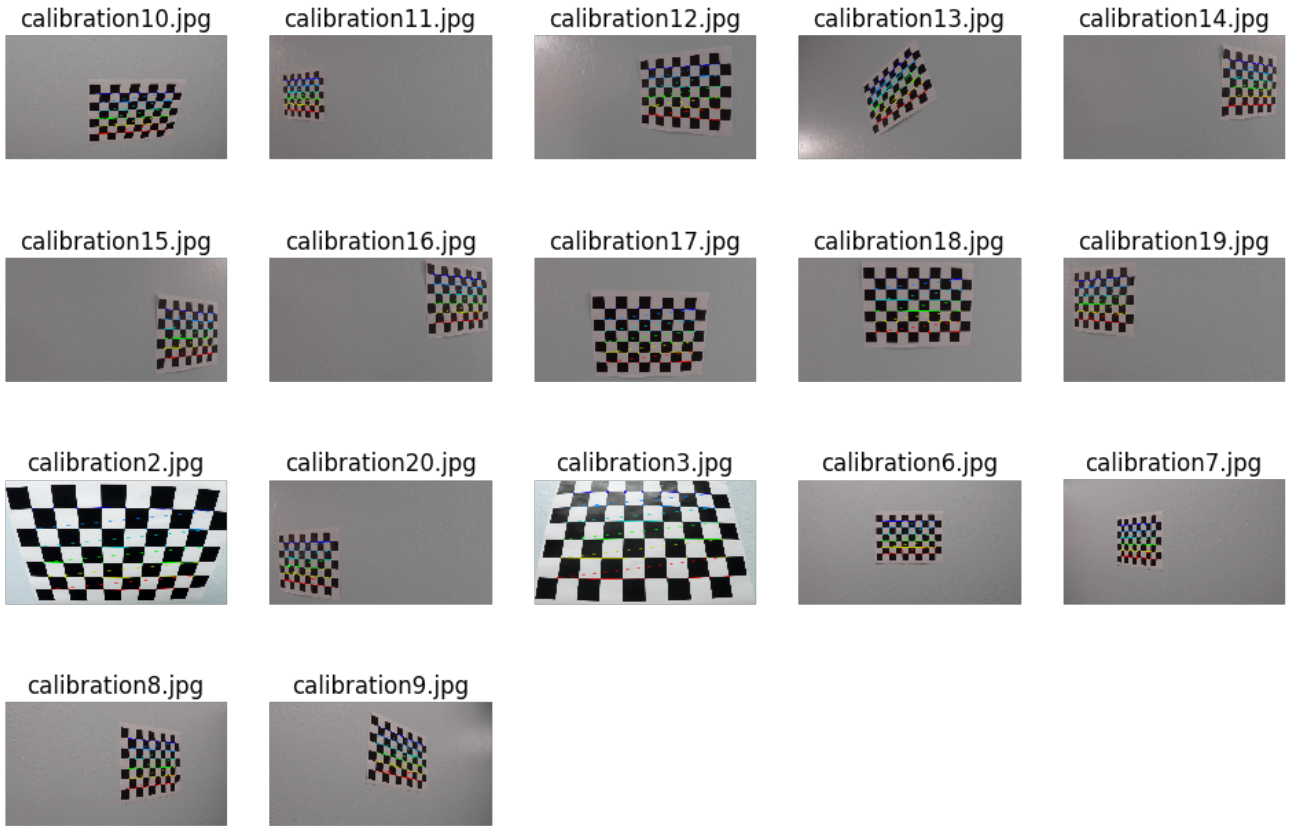
Camera Calibration

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

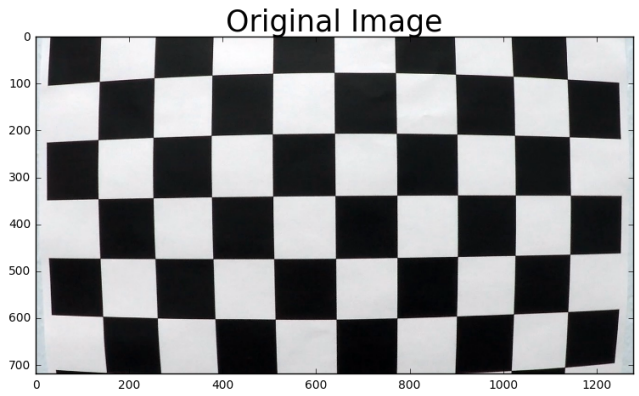
The code for this step is contained in the ***Step 1 on Camera Calibration*** cells in the Jupyter notebook located in "./Advanced-Lane-Line-Detection-P4"

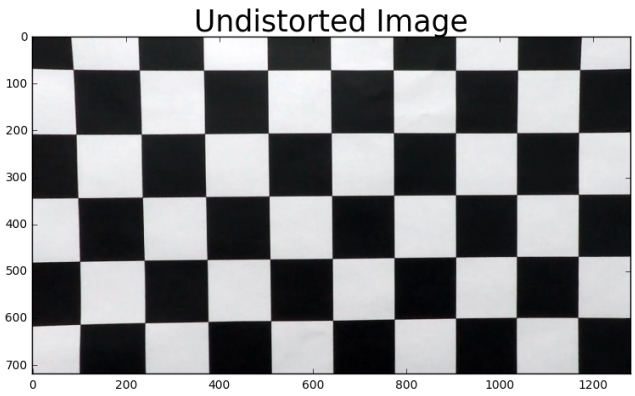
I started by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I assumed that the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array 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 used the **cv2.findChessboardCorner**s for detecting the corners.

I ran this program thru' all 20 images and appended to the `objpoints` and `imgpoints`. The 20 images are given below.



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:





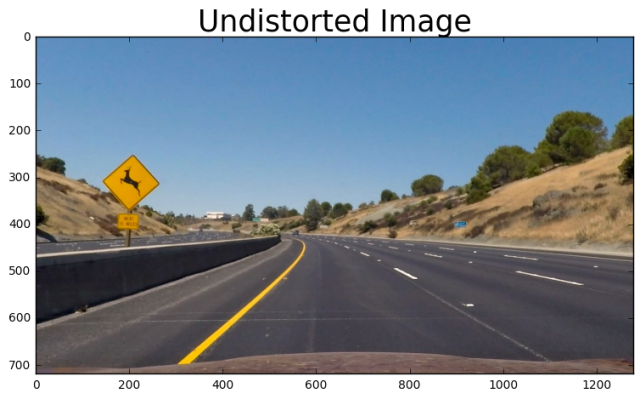
Pipeline (single images)

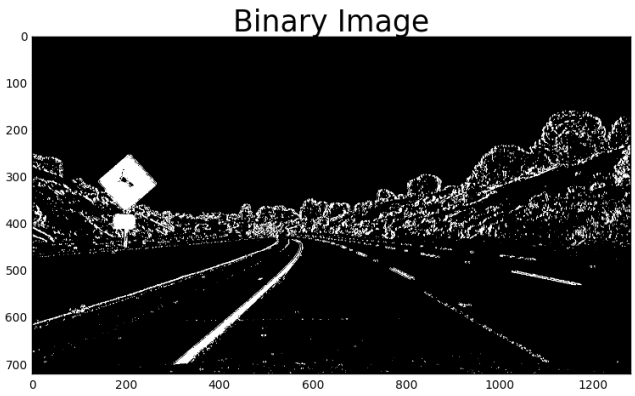
1. Provide an example of a distortion-corrected image.

To demonstrate this step, I used the function used in the previous stage and corrected one of the test images given. The cells **Step 2 - Distortion Correction,** in the Jupyter notebook give the code used for the correction. The example below is the test2.jpg, before and after applying the distortion correction.

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.

Step 3 - Color/ Gradient Threshold and do a binary transform of the image in the Jupyter notebook gives the code used for the thresholding of the binary images. I used a combination of color and gradient thresholds to generate a binary image. Here's an example of my output for this step. This again used the test2.png image, used in Step 2.





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

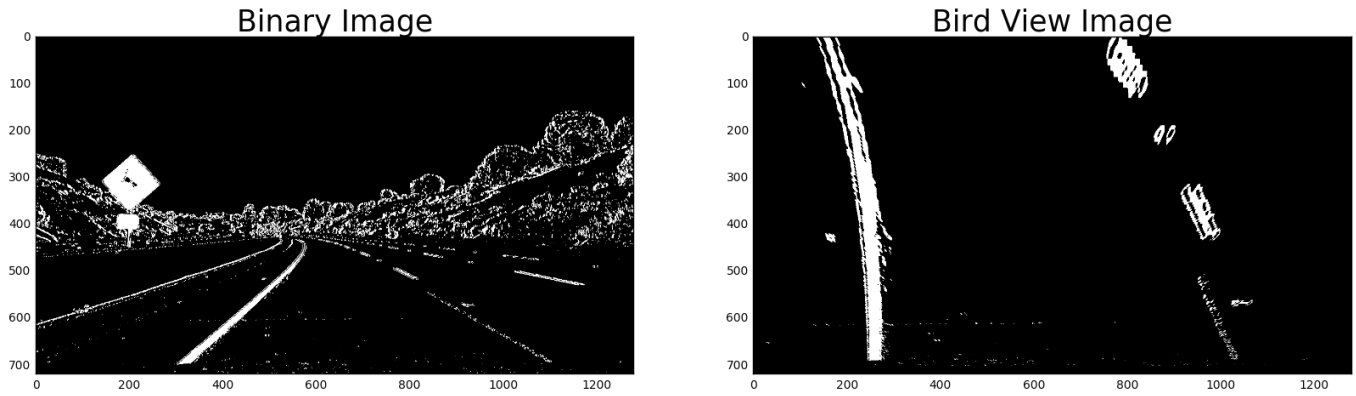
The code for my perspective transform includes is given in the code cells named in **Step 4 - Perspective Transform - Get a bird's eye view of the road,** of the Jupyter notebook, using a function called `perspective_xform(img)`. This takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

```
def perspective_xform(img):
    """ Apply perspective transformation on input image.
        Returns: The transformed input image (Bird's Eye) as uint8 type.
    """
    img_h = img.shape[0]
    img_w = img.shape[1]

    top_left     = [580,  460]
    top_right    = [740,  460]
    bottom_left  = [280,  680]
    bottom_right = [1050, 680]

    src = np.float32([top_left, bottom_left, top_right, bottom_right])
    dst = np.float32([[200,0], [200,680], [1000,0], [1000,680]])

    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)

    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(img, M, (img_w, img_h), flags=cv2.INTER_LINEAR) # Change
    return img_as_ubyte(warped), M
```

The code used the following source and destination points, by eye-balling the image:

Source	Destination
580, 460	200, 0
280, 360	200, 680
1050, 680	1000, 0
695, 460	1050, 680

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.

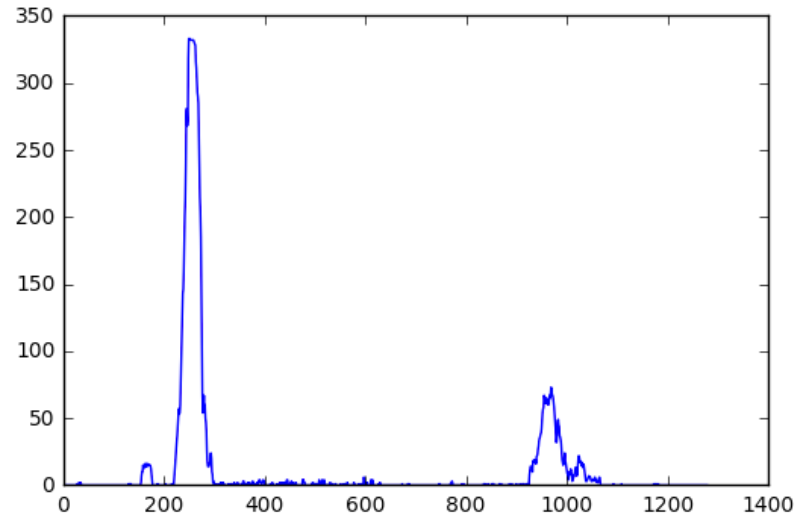


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

After applying calibration, thresholding, and a perspective transform to the road image, the resulting binary image had the lane lines standing out clearly. However, I still needed to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

Pl. see cells corresponding to **Step 5. Detect lane pixels and fit to find the lane boundary,** of the Jupyter notebook.

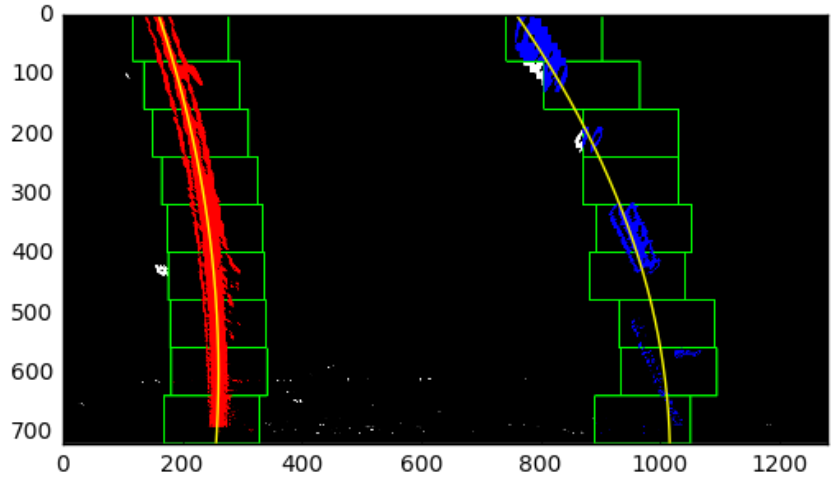
I first took a ***histogram*** along all the columns in the ***lower half*** of the image like this:



With this histogram, I added up the pixel values along each column in the image. In my thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. I then use that as a starting point for where to search for the lines. From that point, I could use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

Once I knew where the lines were, I had a fit! In the next frame of video, I did not need to do a blind search again, but instead just searched in a margin around the previous line position

The output looked something like this:



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.

Pl. see cells corresponding the function def locate_lane_lines_after(binary_warped) in **Step 5. Detect lane pixels and fit to find the lane boundary,** of the Jupyter notebook.

The radius of curvature at any point x of the function $x=f(y)$ is given by the following formula:

$$Rcurve=\frac{|dy2d2x|}{[1+(dydx)2]3/2}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y)=dydx=2Ay+B$$
$$f''(y)=dy2d2x=2A$$

So, our equation for radius of curvature becomes:

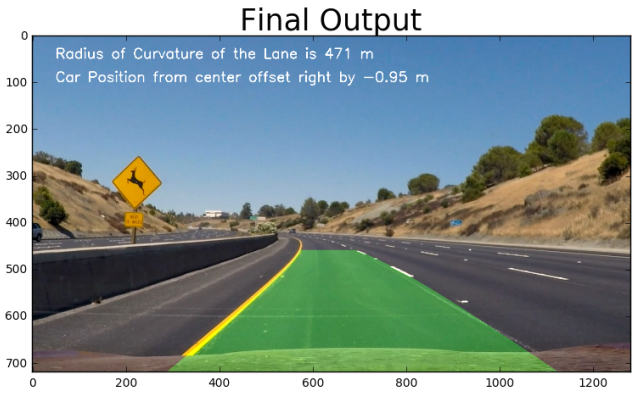
$$Rcurve=\frac{|2A|}{(1+(2Ay+B)2)3/2}$$

The code below implements the radius of curvature which also converts the pixel values to real world coordinates.

```
# Compute curve radius
# 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 = 3.7/700 # meters per pixel in x dimension

# Choose the maximum y-value, corresponding to the bottom of the image
y_eval = np.max(ploty)

# Fit new polynomials to x,y in world space
left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2)

# Calculate the new radii 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])
```

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

An example of the image (test2.png) with all the pipeline stages implemented and lane lines drawn with poly fill is given below. This is part of the **Test it on the final images,** cells in the Jupyter notebook.



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 the link https://www.youtube.com/watch?v=8on9smjldSw&feature=youtu.be to my video output, *project_video.mp4,* which is also part of the Jupyter notebook. I have tried the pipeline on the other 2 videos, *challenge_video.mp4* and *harder_challenge_video.mp4* as well. They are embedded in the 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?

The first issue was noticed when the image has sharp bends. The lane lines were not tracking properly. After adding the smoothing function, to a large extent the problem vanished. I still have issues for the two challenger videos, where the camera is mounted at a different place and the roads have a lot of sharp curves and alternating shadows and bright light. I guess, I need to work on the Color/ Gradient Threshold of the image with different functions and thresholds to get this working correctly.