

Project 4 Advanced Lane Line Finding

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

The codes for the project includes two parts: one is for single images in which a few image examples are included. The other one is similar to the first one but is organized for video processing. The two codes are written in Jupyter Notebook located in `"/Submission/Project 4_Single Image.ipynb"` and `"/Submission/Project 4_Video.ipynb"`


The examples given in the following writeup and in the codes may not be the same, since different images are used.

Camera Calibration

The code for this step is contained in the **Camera Calibration** cell.

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. The object points are in size of (9 * 6) (col * row). The third dimension is set to 0 under the assumption 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 `objpts` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image.

For "image points", I use `cv2.findChessboardCorners()` to find the coordinates of the corners of each cell of the chessboard, and append them to `imgpts` every time I successfully detect chessboard corners.

I then used the output `objpts` and `imgpts` to compute the camera matrix `mtx` and distortion coefficients `dist` using the `cv2.calibrateCamera()` function. Then, I save them to pickle file for later use, so that I do not need to compute them each time I do calibration. I only need to reload the saved pickle file. Next, I applied this distortion correction to the test image using the `cv2.undistort()` function with target image, `mtx` and `dist` and obtained this result: 

Pipeline (Single Images)

1. Distortion correction of road images

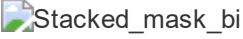
I use the same method to undistort road images as I use for the chessboard. One example is shown below:



2. Creation of Binary Images

I use a combination of shape-based feature (Sobel x gradient) and color-based feature (S channel of HLS color space) of images to generate binary images (the codes are in **Define Thresholds** cell, and executed in the **Creation of Binary Images Using Thresholds** cell). The outputs after threshold methods are shown below:



After combining the threshold methods, I filter the output image with a trapezoidal mask. The region of interest is shown in red dash lines in the image on the left, along with the masked binary image on the right. The mask is set larger than the real lane line region because the lane line region will extend when cars run into turns. The codes are in the **Define Region of Interest** cell. 

The vertex points are shown below:

Corners	Coordinates
up-left	(600, 430)
up-right	(680, 430)
down-right	(1200, 720)
down-left	(100, 720)

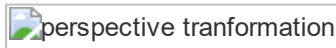
3. Perspective Transformation

After binary images is created, it needs to be converted to a bird-eyes view by perspective transformation. The transformation matrix M is computed by `cv2.getPerspectiveTransform()` function with source and destination points. By switching the source and destination points, $Minv$ is also computed.

The code for the perspective transformation includes a function called `perspective_trans()`, which appears in **Perspective Transformation** cell. The `cv2.getPerspectiveTransform()` function takes source (`src`) and destination (`dst`) points. The coordinates of source and destination points are given below:

Source	Destination
(600, 448)	(330, 0)
(230, 700)	(330, 720)
(1080, 700)	(950, 720)
(680, 448)	(950, 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.



Another example of right turn:



4. Lane Line Detection Using Sliding Windows

4.1 Single Images

After obtaining the bird-eyes view binary image I use sliding windows method to identify the lane line pixels. The codes are in the **Lane Line Detection** cell. The procedures are in the following:

1. Determine the base positions of the left and right lines by computing the histogram of the bottom half of the binary image. The peak positions of the left half and right half of the histogram are the base positions of the left and right lane lines, respectively.
2. Determine the window height by choosing the number of windows across the image.
3. Sliding the window from bottom to top of the binary image and store the positions of pixels inside the windows and with value of 1. The method is applied to the left and right lane lines separately. The center of the windows is initially set at the bases of the lane lines, and is updated with the mean of the positions of new detected line pixels.
4. After all good pixels are detected across the image, second degree polynomial curve fitting is applied to the positions of detected left and right line pixels.

The sliding windows (green box) and detected line pixels of left (red color) and right (blue color) lane lines are shown in the figure below. The fitted curves are shown in yellow.



4.2 Curvature and Offset Calculation

The curvature of the lines are calculated by $R = (1 + (2Ay + B)^2)^{1.5} / |2A|$ for 2nd order polynomial $f(y) = Ay^2 + By + C$. The codes are in the **Curvature Calculation** cell. The offset is calculated by comparing the horizontal midpoint of the image which is assumed as the center of the car and the mean of the base positions of lane lines which is the center of the road. The curvatures and offset are converted from pixels to meters, so that it is easy to check if they have reasonable values. The conversion codes are shown below:

```
# Unit conversion
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
# Calculate the new radii of curvature
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)
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])
# offset
offset = (midpoint - np.mean(np.array([leftx_base, rightx_base]))) * xm_per_pix
```

4.3 Sanity Check

After fitting the good pixels with polynomials we have to check if the detected lines are real. The codes are in the **Sanity Check** cell. The following are the criteria I use for sanity check:

- If the bases of the left and right lines are separated too close (< 2.8 m) or too far (>4.2 m).
- If the middle point of the left and right lines are separated too close (< 2.8 m) or too far (>4.2 m). This criteria can also check if the lines are roughly parallel.
- I also tried to check if the top of the lines satisfy the above threshold, but it will filter out many good lines since the top of the lines usually separate from each other after perspective transformation, so I deprecate this criteria. But another option is relaxing the thresholds.

Only the lines that pass the above criteria can be considered as detected lines and used to update the line classes and warped back onto road images.

4.4 Skipping Windows and Smoothing for Video Processing

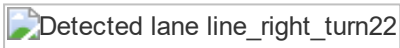
Sliding window method for each frame in a video is computationally intensity and will limit the algorithm to real-time application. To improve the efficiency, we search in a margin around the previous line positions rather than blindly searching the whole image when we are able to detect the lane lines in the last frame. An example of the searching zone defined by the margins is shown below. The codes are in the **Skip Searching Window** cell.



When processing a video, the sliding window algorithm could fail to detect lane lines in some frames due to bad lightening, lack of lines or other issues. To improve the robustness of the algorithm, a smoothing technique is used. First, a class of lines which includes line information of last N frames is created, and updated each time new lines are detected by using the method `self.update()`. The class of left and right lane lines are defined separately. The line information includes number of buffer frames `self.buffer`, recent fitted positions of lines `self.recent_xfitted`, recent fitting coefficients `self.recent_fit_coef`, best fitted positions `self.bestx`, best fitting coefficient etc.. The best fitted position and fitting coefficients are the average of positions and coefficients of last N frames. After defining the line class, whenever no lines are detected in a video, stored best fitted positions are used as the current lane lines. In this way, bad frames are smoothed by averaging over last N frames. The codes for lines class portion is shown in **Lane Line Class** cell.

5. Warping Back Detected Lane Area to Road Image

I implemented this step in **Warped Back** cell. I first fill the polygon defined by the detected left and right lane lines, and then did inverse perspective transformation with inverse transformation matrix `M_inv`. The function I used is `cv2.warpPerspective()`. An example of my result on a test image is shown below:



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 process the video using the codes in **Video Process** cell, and 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?

- One issue I had when I implement the codes is defining the output image `out_img` for visualization. I had to use `(0, 1, 0)` to draw the searching window rather than `(0, 255, 0)`. After trial and error, I found two things went wrong. One is when I apply mask in **Region of Interest** cell. The `ignore_mask_color` should be set to 1 rather than 255 since I use binary images. The other one is when I define the 3D `out_img`. I should set the `out_img` to `uint8`.
- Another issue is converting the color space. I found that only when one wants to display the image using `plt.imshow()`, the image needs to be converted back RGB, i.e. `plt.imshow(cv2.cvtColor(test1, cv2.COLOR_BGR2RGB))`. If we are processing a video, there is no need to change the color space. I presume some video functions do it implicitly.
- For the thresholds method, I also tried gradient direction but it did not give me reasonable output. Using magnitude gradient can improve the results with correct thresholds, but I need time to play with the thresholds.
- For the video, the detected lines are inside in the real lane lines sometimes when the pavement changes color from black to light gray. The reason could be other color gradients close to the real lines. It could be avoid by defining an inner polygon mask, or add other thresholds. But I am not sure what the correct approach. I am looking forwards to some suggestions.
- Another question is how to increase the processing speed.
- Sometimes the left and right curvatures are much different, I am not sure how to improve it.