Udacity SDCND Advanced Lane Finding Project *(Hai Xiao)*

The goals / steps of this project are the following:

- *Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.*
- *Apply distortion correction to each raw image to generate undistorted image.*
- *Apply a perspective transform to rectify binary image ("bird-eye view").*
    - *I think do this first then color and gradient threshing is logical, as this step will bring up more focused area (view) to next steps, easier debug as well.*
- *Use color transforms, gradients, etc., to create thresholded binary image.*
- *Detect lane line pixels and fit curve lane lines with polyfit, track and reject noise.*
- *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 consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

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.

This is the Writeup for Project 4: Advanced Lane Finding.

Please also notice other **submitted files** (extract P4.zip):

- project4.ipynb — project ipython notebook file (executable)
- project4.html — exported HTML file from notebook execute
- output_images/calibration*-corners.jpg — calibration images with chessboard corners
- output_images/calibration*-undist.jpg — undistorted calibration images
- output_images/test*-birdeye.jpg — rectified/birdeye test image post undistortion
- output_images/test*-bin.jpg — color and gradient filtered binary images
- output_images/test*-warp.jpg — color warped test image with curvature, etc.
- output_images/project4.mp4 — processed video from ‘project_video.mp4’
- output_images/test3-sliding-window-lane-finding.jpg — also provided as an example

Camera Calibration

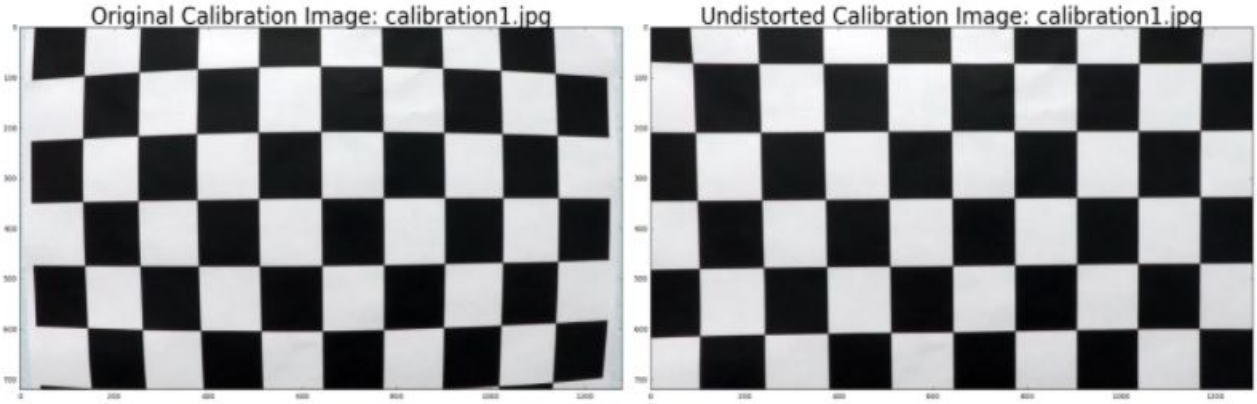
Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

Code for this step is contained in **Section 2** (project4.ipynb/html), specifically **Section 2.1** generated the camera matrix and distortion coefficients using standard OpenCV procedures.

- First use `cv2.findChessboardCorners()` to find pattern of corners from various different calibration images (some were close-up, some were partial), here I implemented good heuristic to automatically extract different corner patterns from all 20 calibration images (detail see `cameraCalibrate()`, `patternSz` in code cell [3])
- **Section 2.2** plot all original images with corners found drew. This verifies the good pattern extraction logic, all drew images were also saved as output_images/*-corners.jpg
- Still in `cameraCalibrate()` global camera matrix and distortion coefficients were generated using `cv2.calibrateCamera()` in code cell [3]
- Then **Section 2.3** calls `cv2.undistort()` and plot all undistorted calibration images, they were also saved as output_images/*-undist.jpg in code cell [6]

N.B. `patternSz = [(9, 6), (9, 5), (7, 6), (5, 6)]` patternSz is defined in code cell [2] of global definitions. My logic here is to iterate through them to find a best corners pattern match to each presenting calibration image, and derive `objpoints` and `imgpoints` accordingly in `cameraCalibrate()`.

Here is example of an original image and distortion correction image:



Pipeline (test images)

Provide an example of a distortion-corrected test image.

To demonstrate this, here is example of an undistorted test1 image (right side):

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

N.B. I swapped order of doing perspective transform and color/gradient transform, for the reason: perspective transform provides more focused area/region for later processing and it is easier to debug.

Code for this step is contained in **Section 3** (project4.ipynb/html), specifically **Section 3.1** defines the source and destination data points and produces the global Perspective Transform Matrix **(M)** and Inverse Matrix **(Minv)** for later use. While **Section 3.2** did the actual bird-eye transform using just generated Matrix **(M)** and provided visual validation for the correctness/robustness of prior selected coordinates.

Section 3.1 in code cell **In [7]**, I have

```
src = np.float32([[530, 480], [ 770, 480], [50, 720], [1250, 720]])
dst = np.float32([[ 50,   0], [1250,   0], [50, 720], [1250, 720]])

M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)
```

Source/Destination points are hand picked while validating all the rectified images via visualization in **Section 3.2**, even though they were not automatically generated, they

were approved to be robust across all these test images. In the ending discussion, an adaptive region selection to this transform is considered important in complex conditions

The bird-eye perspective transform (Code Cell In [8]) works as expected with given `src` and `dst` points on all test images from **Section 3.2** plot. Here is an example (note the src region selection within `blue` line segments in the Left image):

N.B. all bird-eye images were also saved as output_images/*-birdeye.jpg

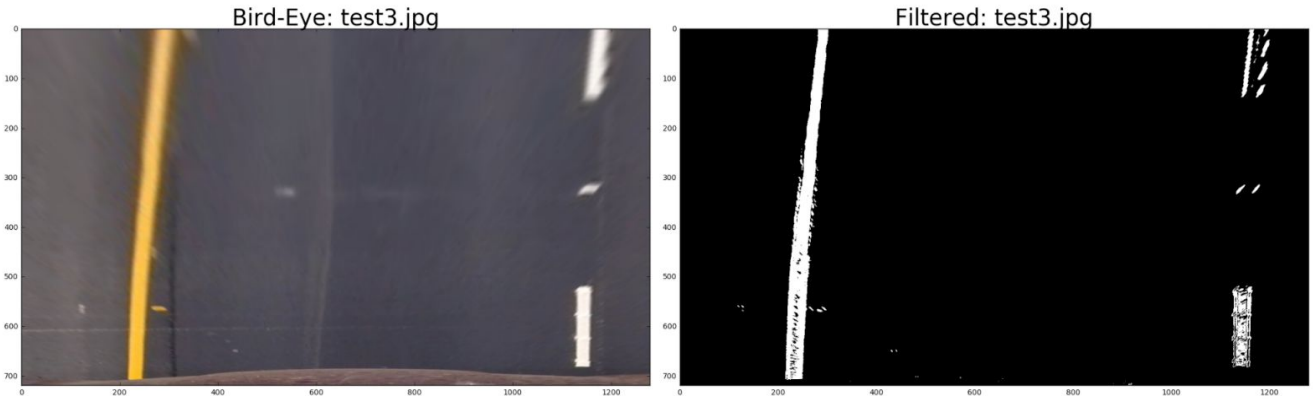
Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

N.B. With each image rectified, the task of doing color and gradient transform and filter are much more visible (earlier debug), since birdeye image provides much more simplified scene.

Code for this step is contained in **Section 4** (project4.ipynb/html), specifically **Section 4.1 and 4.2** implemented and visualized HLS color space S-Channel threshing. **Section 4.3 and 4.4** implemented and visualized Sobel-x gradient threshing.

Combined color and gradient threshing is implemented in **Section 4.5** to generate final binary images for next step (line find) in pipeline, and they are visualized in **Section 4.6**.

Here's an example of my output binary image. (***N.B.*** all final binary images were also saved as output_images/*-bin.jpg)



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

On final binary image (rectified, color/gradient threshed), **Section 5** (code cell **[16]**) implements:

- Find X-bases of Left/Right Lanelines using histogram of lower half Y[360:], in **Section 5.3**

  frame_xbases(img, minheight=0)

- An utility function to produce a selected binary patch (each Red or Blue rectangle in the example picture below), given prior X-base & Y, also return new X-base for next sliding window search above, new X-base is based upon points distribution of current window. ***N.B.*** *sliding window is the small rectangle patch (vs. whole horizontal stripe) in the following example image*

  sliding_win(img, xroot, ybase, xdelta=50, ydelta=60, dir=0)

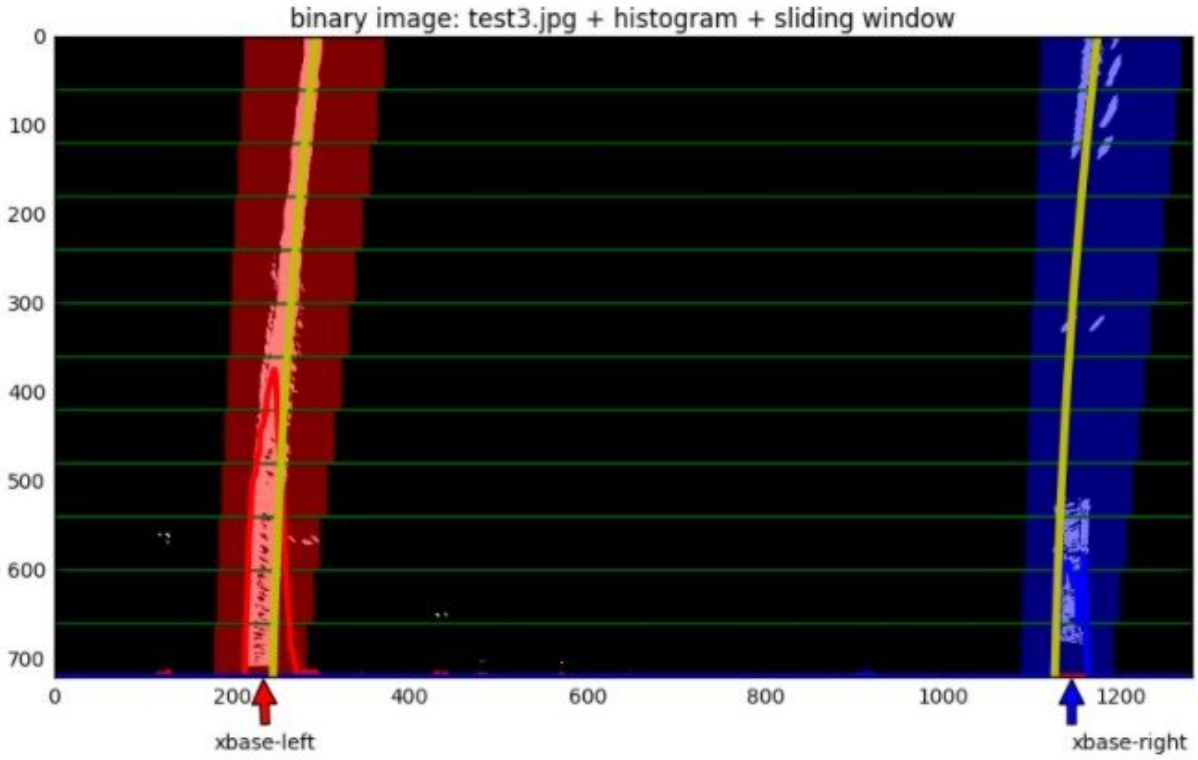
- Use both X-bases and sliding window to find all qualified L/R lane line pixels to fit later, still in **Section 5.3** code cell **[16]**

  findingLine(img, llane=**None**, rlane=**None**, minsample=0, debug=**False**)

- Use quadratic polynomial fit to produce fitted lane line once lane lines pixels produced, it is implemented in function colorwarp() for **pipeline** usage in the same **Section 5.3** code cell **[16].** But to the test images workflows, lane fit is actually done in **Section 5.4** code cell **[17].**

  N.B. *both* findingLine() and colorwarp() *provide lane pixels/lines* ***tracking*** *and noise* ***rejection*** *mechanism, when they are invoked with a corresponding lane line class instance. Please see a very first version of* ***Laneline() class*** *implementation* in **Section 1** code cell **[2]**

Here is an example image to demonstrate all steps done in **Section 5.1 - 5.4** for lane lines finding and fitting:



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.

This is done in **Section 5.3** code cell **[16],** function:

`curvature_centeroffs(leftx, lefty, rightx, righty)`

Primarily it does:

- calculate an L/R averaged polyfit curvature radius in meters (use estimate), using equations from course lecture.
- calculate an estimated vehicle lane center departure in centi-meters, (-) means vehicle is shifted left to lane center, vice versa. It is computed according to the distance between pixel of fitted lane center in camera view and the image center of camera view itself.

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

Final color warped images w/. lane lines are produced in **Section 5.5** and code cell **[18]**

Here is an example of color warped test6 image:

Pipeline (video)

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!).

Implementation is in **Section 6.** Generated project video is located in submission as:

output_images/project4.mp4

Discussion

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?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

Primarily I took a straightforward approach to build image pipeline for this project, it works well in the provided project video, but not so great in harder challenge ones, here are some summary, after thought and some highlights to improvement moving forward:

- *It sounds logical to do Perspective Transform first then Color and Gradient filtering, provided more focus*
    - *And that is how I did*
- *Yet to synthesis vehicle theta angle (angle between vehicle heading and lane heading) that is computable*
    - *It is very useful feature to predict vehicle shift, lane center departure, and next lane line starting X-base, etc.*
    - *It is computable from the angle of tangent at (X-base, Ymax=720) on previous fitted lane lines*
- *Yet to synthesis left & right lines tracking, derive good polyfit coefficient from other side if one side is rejected*
- *Yet to support adaptive (vs. static) region selection for perspective transformation*
    - *This is very useful to work with different road conditions, such as shorter viewpoint, windier turns, etc.*
- *Yet to add better noise cancellation to filter objects inside fillPoly region, since those may fail lane finding*
    - *Those interferencing objects can be CARs ahead nearby, etc.*
- *Yet to harness discovered lane line curvatures and lane center departures*
    - *They can be used to derive steering angles (with other sources, such as DNN trained signals) for example.*
- *Yet to refactor code further to more sophisticated tracking, rejection and smooth*
    - *Move more data and add more logic inside classes*

Thanks