Write Up for P2: Advanced Lane Lines

Introduction:

This document is a write up for the second project. As mentioned in the write up template, the objectives are:

- 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.

In order to make the report more consistent, I decided to make the description for code blocks that were already provided in the course brief and more detailed personal on code.
Project structure is as follows:
Code: /**P2.ipynb**
Videos:Input= **/test_videos** , Output= **/output_videos**
Images: Input= **/test_images** , Output= **/output_images**
Each cell in the pipeline is named and numbered, so code parts are referred to by cell name/number (+lines range if the cell contains more than one function). The code structure follows the rubrik steps: Camera calibration -> Image pipeline -> Video pipeline.

Camera calibration

I use the camera calibration function provided in the course without modification. The code is in script format (not a function) and takes as input $\mathbf{n_x}$ & $\mathbf{n_y}$. Below before/after example is provided here.
Code location: **First: camera calibration using chessboard images**

Image pipeline

The pipeline consists of one function: **image_pipeline** located in **4. Pipeline Function**
Dependencies (cell names):

- **Camera calibration parameters:** These are a one-time-cost, such as camera matrix and distortion coefficients.
- **1. Helper functions:** Contains all the helper functions developed for this project.
- **2. Source points for each road:** Used to perform perspective transform and contain the coordinates for 3 polygons, one for each video input.
- **3. Tracker class:** Contains project global variables, mostly useful for videos.

Side code:

- **5. Convert png images (from video) to jpg:** Converts video screenshots from .png to .jpg

- **6. Finding source points:** Manually get coordinates of lanes
- **7. Parameters tuning:** Self explanatory
- **8. Load and process image:** performs the pipeline on image then saves the result

Distortion correction

Function Name: None, it's one line of code: *cv2.undistort*
Location: Called directly within the **image_pipeline, line 6**
Input: Image + camera parameters
Output: undistorted image
Description: The before/after example provided below serves to illustrate camera calibration as well. Distortion correction is visible on the car's hood

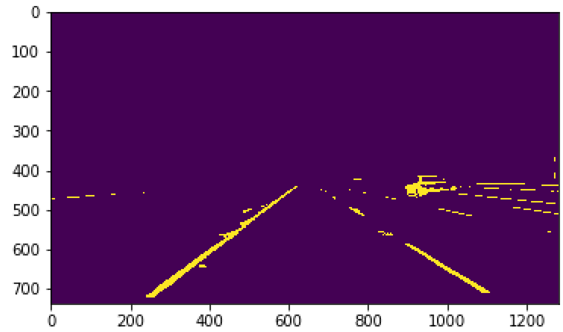
Before

After

Binary Transform

Function name: ***create_binary***
Location: **1. Helper functions, lines 3->21**
Functions called inside: **None**
Input: Image, l_channel threshold, b_channel threshold
Output: Binary image
Description: After trying several color spaces and gradient methods, I found that LAB colorspace provides the best overall performance. **L**, **B** channels to detect white, yellow colors respectively. I discarded gradients method because I found that they were very prone to noise which disturbed the lane finding algorithm.
My approach was to favor cleanness of prediction with a risk of an empty lane to robustness with a risk of high noise. I have to be clear though, image processing is a craft that I don't muster! My approach was empirical and not based on prior knowledge or experience but I'm thankful that I learned a bit in this project.

Before **After**

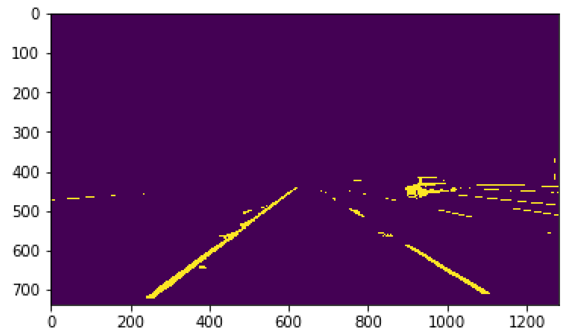


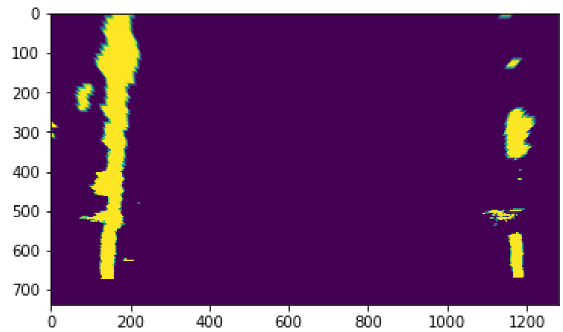
Perspective Transform

Function name: **transform_perspective**
Location: **1.Helper Functions, Line: 27 -> 36**
Functions called inside:**2. Source points for each road**
Input: image, source_point_coordinates, offset
output:warped_img, M, Minv
Description: The code for this function is identical to the course: Use a manually picked coordinate as input and “offsetted” polygon in the target image.
It’s important to report that offset + coordinates have to be adapted to the image otherwise the results may be very different, I report 2 reasons:

1. Lanes location can be different from one road to the other. That’s why I set 3 different coordinates according to the road.
2. A large offset implies fitting the polynome to a larger surface which means more wiggling in the fit, on the other hand, a small offset does not capture road curvatures

Before **After**





Identify Lanes

Function name: **find_lane_lines**
Location: **1.Helper Functions, Line: 39 -> 45**
Functions called inside:
-lanes_from_previous, located: 1.Helper Functions, Line: 47 -> 64
-sliding_window, located: 1.Helper Functions, Line: 66 -> 127
Input: image,nwindows,margin,minpix,margin2
Output: leftx,lefty,rightx,righty,left_found,right_found

Description: This function identifies lanes in the input image but does not fit. If lanes in previous image were found then I use **lanes_from_previous,** if not **sliding_window.** The output consists of x,y coordinates for left/right lanes + 2 binary variables: Lane pixels found or not. Almost all code is provided in the course, I have added 2 restrictions:
-If one lane is 5 times longer than the other ⇒ the short one is considered not found.
-If length of the longest lane < 1/2 image height ⇒ both are considered not found.
The reason is that most often, the polynomial is badly fitted if the length is too short.

Fit Lanes

Function name: **fit_lane_lines**
Location: **1.Helper Functions, Line: 130 -> 306**
Functions called inside:
-Line class, located: 3. Tracker class
-compute_curvature, located: 1.Helper Functions, Line: 309 -> 315
Input:
-image,
-leftx, lefty, rightx, righty, left_found, right_found, #output from previous lane finding function
-smoothing (Use smoothing or not), xm=3.7/700 (width pixel resolution), ym=30/720 (height pixel resolution)
Output: image with fitted lanes
Description: The function can be divided into 3 distinct parts: Assess quality of found lanes → Decide what to do based on the quality → fit to the image either by smoothing or not. I'll go through each step.
NB: One key assumption I use is road width constance.

Quality assessment:

Location: **Line 142 → 205**
Input: Image, lefty, rightx, righty, left_found, right_found
Output:2 booleans: **good_left_fit** & **good_right_fit**
Description:
We define an error margin named "alpha" (around 30%) for the assessment of quality.

1. For each lane: Initialize to bad fit
   If we found pixels → Fit polynomial→ If previous lane was fitted→compare curvature + quadratic coefficient + intercept with previous → If OK for all→fit is good
2. For both lanes and if previous tests were successful:
   Compare quadratic coefficients → If NOK → Shorter lane is no good fit

Decision based on fit quality

Location: **Line 209 → 289**
Input: **good_left_fit** & **good_right_fit**
Output (approximate):fitted lanes + updated global coefficients
Description: The decision part is 2 folds:Find the best fit + update global variables

1. If both fits are bad:
   Fitting: Last best fit stored in the global variable

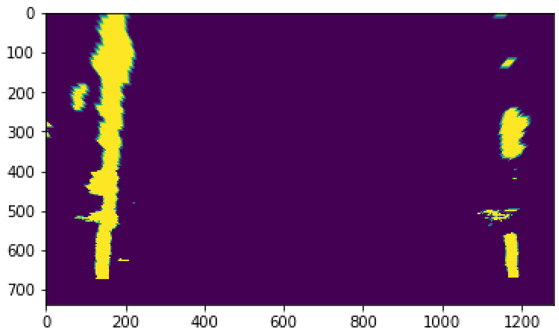
Update: None

2. If one fit is bad:
   Fitting:
   -If previous frame was well fitted for that lane → Use coefficients from previous frame
   -If not → Use current good frame coefficients + compute the road width → Adjust intercept according to the computed width
   Update:
   Road curvature from the good lane + polynomial fits for both lanes + set the bad lane to not found
3. If both fits are good:
   Fitting: Intuitive, Update: Analogous to previous

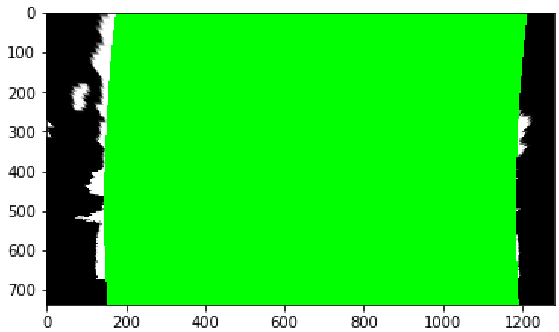
Fit lanes to the image

Location: **Line 290 → 304**
Input (approximate): Decided fits for current frame
Output (approximate):Fitted lanes to the image
Description: By now we have the fits resulting from the current frame. Depending on the decision to smooth frames or not + size of the smoother we output the final frames.

Before



After



Compute curvature

Function name: **compute_curvature**
Location: **1.Helper Functions, Line: 309 -> 315**
Functions called inside: None
Input:imY,ypix,xpix,xm,ym
Output: Radius
Description: The code for this block is identical to the one provided in the course.

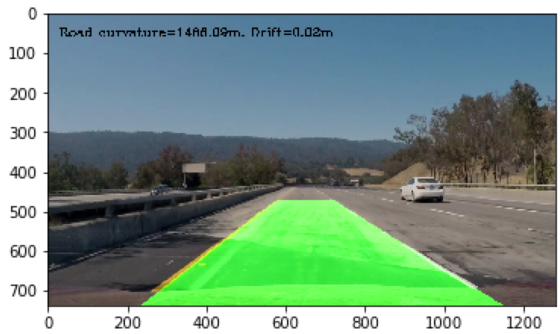
Mix project image + original

Function name: **weighted_img**

<u>Location</u>: **1.Helper Functions, Line: 318 -> 322**
<u>Functions called inside</u>: None
*<u>Input</u>***:**original image, projected image
<u>Output</u>: final image
<u>Description</u>: I add the radius + car drift during this step

Before

After



Video pipeline

Project video & challenge video link:
https://github.com/mmarouen/CarND-Advanced-Lane-Lines-master/tree/master/test_videos
Harder challenge lin:
https://drive.google.com/open?id=1B2gh8qizghYTNrWQ9u3e7G3uIWzUVHIu

Discussion

A word about difficulty

Besides the method, the list of parameters by step that impact final result includes:
Binary image: Thresholds
Perspective transform: Offset amount + coordinates of the source polygon
Lane finding & fitting: nwindows,margin, minpix, smoothing size, error sensitivity
Image processing is a craft in itself, the challenge for me was to connect the dots and come up with a "reasonable" result given my current knowledge.

Improvement areas

1. The harder challenge made me aware of potential improvement possibilities:
-When one lane is not in the image for high curvature
-Detect/Process high light images
-Detect double lanes and find the center
-Deal with high curvatures: When the polynomial fit crosses the vertical half of the image

2. Fixing smoother size inversely proportional to car speed