Advanced Lane Lines 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.

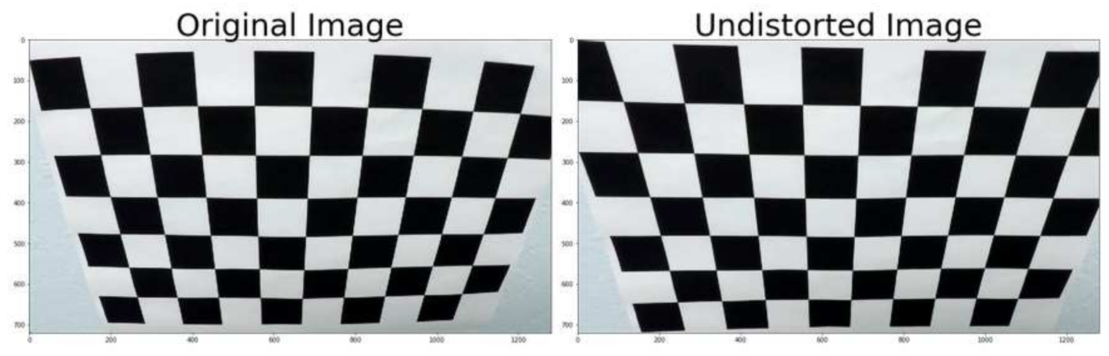
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Camera Calibration

1. Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?

The code for this step is contained in the first code cell of the IPython notebook.
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, 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 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:



`--> Please see the IPython notebook for more examples.`

Pipeline (single images)

1. Has the distortion correction been correctly applied to each image?

To demonstrate this step, I will describe how I apply the distortion correction to the the test images which can be found in the IPython notebook:
a) The previous step has yielded the camera calibration matrix and distortion coefficients, i.e. ret, mtx, dist, rvecs, tvecs
b) I used these coefficients to apply undistortion to each image with cv2.undistort(img, mtx, dist, None, mtx)

Example:

```
Original image:
```

```
Undistorted image:
```

`--> Please see the IPython notebook for more examples`

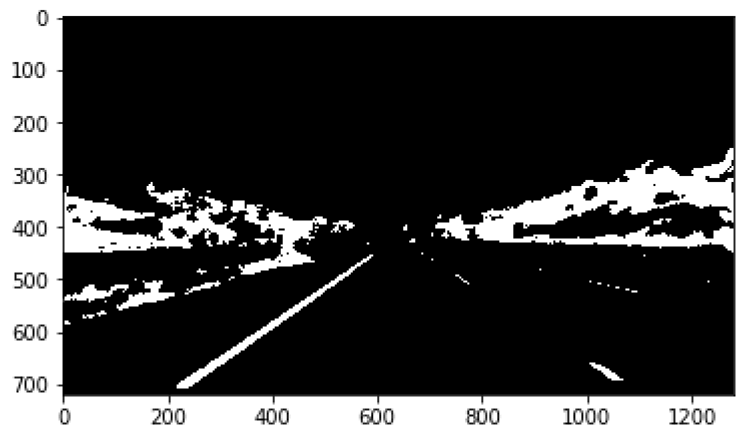
2. Has a binary image been created using color transforms, gradients or other methods?

I have used a combination of thresholding L in LUV for detecting white and B in LAB for detecting yellow.

In my code I'm using the function "l_and_b_filter()" for this purpose.

Example:

```
Combined thresholds:
```



--> Please see the IPython notebook for more examples

3. Has a perspective transform been applied to rectify the image?

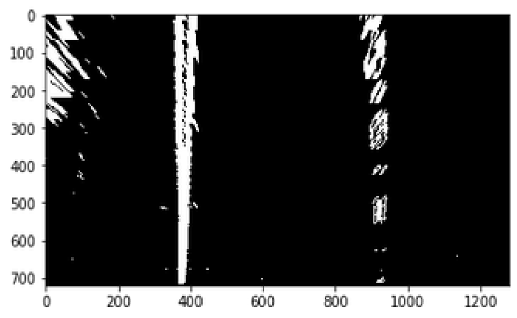
The code for my perspective transform is defined in the function "perspective_transform()". The perspective_transform() function takes as inputs an image (img). I chose to hardcode the source and destination points in the following manner:

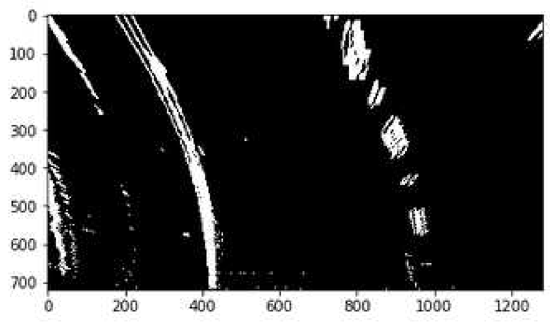
src = np.float32([[200, 719], [1108, 719],[535, 490],[750, 490]])
dst = np.float32([[370, 720], [929, 720],[370, 490],[929, 490]])

I verified that my perspective transform was working as expected by warping a number of test images and checking whether the lines appear parallel.

Examples:

```
Warped image / perspective transform:        Warped image / perspective transform:
```

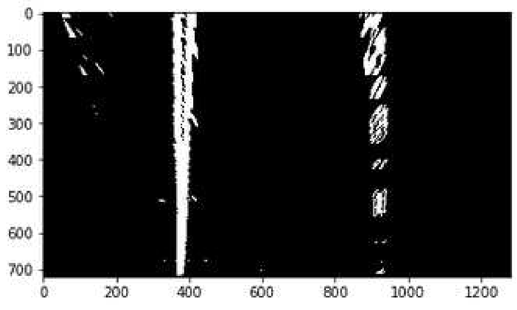




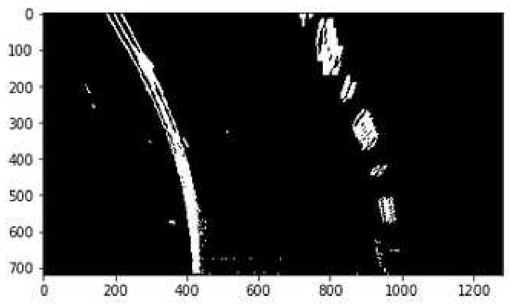
--> Please see the IPython notebook for more examples

As an additional step, I also masked out the left and right sides of the image to avoid "noise":

Masked warped image:



Masked warped image:

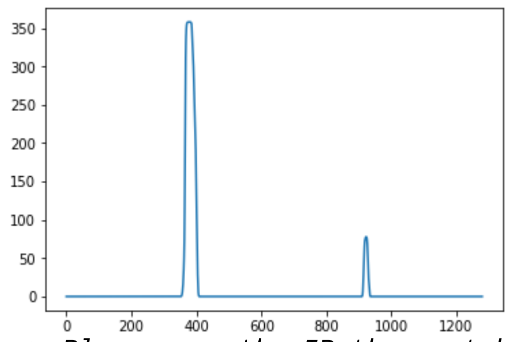


4. Have lane line pixels been identified in the rectified image and fit with a polynomial?

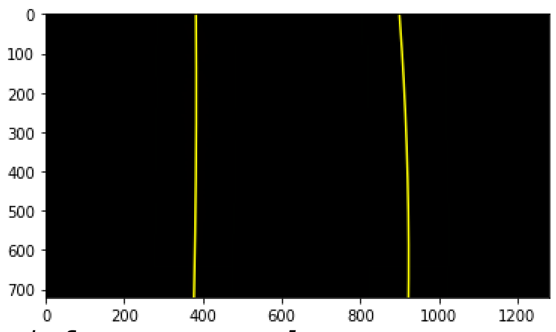
Polynomial fitting is defined in the function called "detect_lane_lines()". I applied the following steps in order to fit the polynomial:
a) Take a histogram of the white pixels in the bottom half of the image.
b) Identify the "peaks" and set them as lane lines bottom starting points
c) Use a sliding window algorithm to collect image points that belong to "white pixel clusters" (or non-zero pixels in the binary image).
d) Use the positions of identified non-zero pixels to fit the 2nd degree polynomials (using np.polyfit)
e) Calculate lane curvature radius (in meters)

Sample result:

Histogram:



Fitted polynomials:



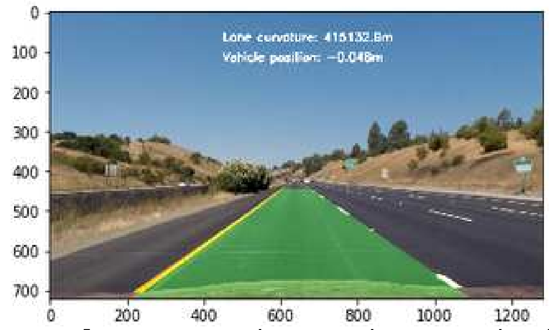
--> Please see the IPython notebook for more examples

5. Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?

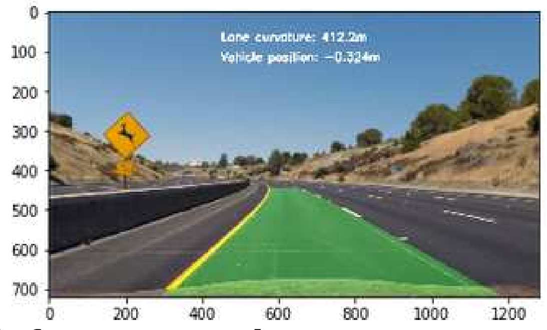
Yes, both values are calculated within the "detect_lane_lines()" function.

Sample results:

```
Final output:
```



```
Final output:
```



--> Please see the IPython notebook for more examples

Pipeline (video)

1. Does the pipeline established with the test images work to process the video?

Yes, the video is attached to the submission.

In order to smooth out the video I'm using a running average of polynomial-fitted x-coordinates and also discarding frames in which the detected lane lines are too close or too far from one another (indicating that they are not parallel).

In this second submission I have also added usage of an area around the lines fitted in prior images to search for lane pixels in a new image.

Discussion

In this second submission the video pipeline works clearly better on the "problematic" shaded or very bright frames from the previous submission. Using L (in LUV) and B (in LAB) channels to detect white and yellow colors (respectively) seems to work better than the gradient-based approach.

Still, even though on all test images my pipeline yields good results, it doesn't produce ideal results on all video frames. In order to improve my pipeline further I would need to further tweak the algorithm and try to optimize the filtering / thresholding to work better. Alternatively I could test different filtering approaches altogether.

Overall, solution design was an iterative process, with many steps until the final pipeline was defined. I believe that the initial steps, such as camera calibration, image undistortion and warping are rather straightforward. The tricky part is the choice of filters and thresholds to identify the line pixels. I experimented with pure sobel gradient, gradient magnitude, gradient direction and color space filtering. Finally I settled for L (in LUV) and B (in LAB) channels to detect white and yellow colors.