Advanced Lane Finding 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.

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 cell #2 of the jupyter notebook.

First, I prepared object points, which describes the real-world position of the grid point of the chessboard, and stored them in list objpoints. Then, I converted the color image to gray scale. Afterwards, I used cv2.findChessboardCorners function to detect grid point locations in the distorted gray image, and stored coordinates in the list imgpoints. Finally, the cv2.calibrateCamera function was used to calculate distortion parameters and cv2.undistort was used for distortion correction.

The example of a distortion-corrected image can be found in the folder "output_images/camera_cal/".

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.

The code is located in the cell #4 of the jupyter notebook. In the end, I combined red channel threshold, saturation channel threshold, gradient in the x direction, magnitude threshold, and direction of gradients threshold to detect lane lines. I found the following combination most useful: first, the pixels should fulfill color threshold (red and saturation) or magnitude threshold. Then, the remaining pixels must satisfy gradient in the x direction and direction of gradient threshold.

The example of a lane line detection image can be found in the folder "output_images/lane_detect_before_transform/"

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 perspective transform is in the cell #6 of the jupyter notebook.

First, it should be start with a straight-line image, since it is easier to find a rectangle on the image. I used “straight_lines1.jpg” as the starting image. I found four points on both lanes:

```
src = np.float32([[272, 673],
     [1035, 673],
     [602, 447],
     [682, 447]])
```

and warped them into the four points of a rectangle:

```
dst = np.float32([[250, 720],
     [1000, 720],
     [250, 0],
     [1000,0]])
```

I used cv2.getPerspectiveTransform function to get the transform and inverse transform matrix.

I tested perspective transform on both distortion-corrected original images and lane detection binary images. The examples can be found in the folder "output_images/perspective/” and "output_images/lane_detect_after_transform/".

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

The code for this part is in the cell #10 and #12 of the jupyter notebook.

This section is divided into two parts:

1. When the polynomial fit lines are unknown, I started with the sliding window detection. First, the histogram across the width of the binary image was calculated. The left and right based points were chosen to be the maximum of the histogram. Then, two sliding windows, which were centered at the left and right base points, were used to detect relevant lane points in the window. If the number of points in the window exceeds the threshold, the left and right base points were re-calculated. The sliding windows were shifted up and the process was repeated until all windows were processed. Finally, a second order polynomial fit was obtained from detected points by using the np.polyfit function.
2. When the polynomial fit lines are known, I directly detected points that are within a small window around the polynomial fit lines. After all relevant points were detected, the polynomial lines were refitted.

Examples of images can be found in the folder "output_images/sliding_window/" and "output_images/poly_fit/".

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.

The code is located in the cell #9 of the jupyter notebook (curvature) and in the cell #10 and #12 (postion, in search_around_poly and search_from_window function).

Radius of curvature: First, the real-world distance of one pixel of the image was calculated. Then, the pixel coordinates were converted into real-world coordinates, and the real-world polynomial fit was calculated. Finally, the radius of curvature was calculated from the parameters of the fit functions.

Position of the vehicle: the lane mid pixel was calculated from most bottom points of the left and right polynomial fit lines. The camera center was just the mid point of the image. The difference between two, converting into real world distance, is the deviation from the lane center.

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

The code can be found in the cell #13 of the jupyter notebook. Example images can be found in the folder "output_images/plot_back/".

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!).

The video is in the same folder as the write up, and the video name is "project_video_output.mp4".

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 difficult part is to find the correct threshold to detect lane lines from a color image. I used almost all techniques covered in the lecture, including gradient in the x direction, red and saturation channel, magnitude, and direction threshold, and adjust parameters patiently to successfully detect lane lines.

The other issue is sanity check of the lane lines found. I need to use test images as example, and come up with parameters, such as radius of curvature difference and distance between lane lines, to judge if the lane lines found are good or not.

The potential fail of the pipeline may be some corner cases, such as no lane pixels are found in the image. I should consider those corner cases and return lane not detected immediately and skip the rest of the code.