UDACITY

Back to Self-Driving Car Engineer

Advanced Lane Finding

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Awesome job with this review! You have clearly put a lot of hard work into it and it looks great! I hope the suggestions I have provided here can help you improve even more. Keep up the great work and stay Udacious!

Writeup / README

The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled.

Your write up is great! I encourage that in the future you try to create it in the md format and separate from using a ipython notebook to create it. This way you are able to publish work and projects on github for maximum visibility. If you don't have much experience with this, Udacity offers a course that I will link below.

https://www.udacity.com/course/writing-readmes--ud777

Camera Calibration

OpenCV functions or other methods were used to calculate the correct camera matrix and distortion

coefficients using the calibration chessboard images provided in the repository (note these are 9x6 chessboard images, unlike the 8x6 images used in the lesson). The distortion matrix should be used to un-distort one of the calibration images provided as a demonstration that the calibration is correct. Example of undistorted calibration image is Included in the writeup (or saved to a folder).

Camera matrix and distortion coefficients were correctly calculated.

Pipeline (test images)

Distortion correction that was calculated via camera calibration has been correctly applied to each image. An example of a distortion corrected image should be included in the writeup (or saved to a folder) and submitted with the project.

Camera matrix and distortion coefficients were correctly applied to each frame.

A method or combination of methods (i.e., color transforms, gradients) has been used to create a binary image containing likely lane pixels. There is no "ground truth" here, just visual verification that the pixels identified as part of the lane lines are, in fact, part of the lines. Example binary images should be included in the writeup (or saved to a folder) and submitted with the project.

Good job extracting the lane lines! Some advice for more robust extraction is to try color thresholding just in the RGB, HSV and HSL channels for your yellows and whites! By using color thresholding you are able to make the lane detection more robust by relying less on gradients for good results! Furthermore it is faster to compute!

Here is some sample code to play around with!

```
HSV = cv2.cvtColor(your_image, cv2.COLOR_RGB2HSV)

# For yellow
yellow = cv2.inRange(HSV, (20, 100, 100), (50, 255, 255))

# For white
sensitivity_1 = 68
white = cv2.inRange(HSV, (0,0,255-sensitivity_1), (255,20,255))

sensitivity_2 = 60
HSL = cv2.cvtColor(your_image, cv2.COLOR_RGB2HLS)
white_2 = cv2.inRange(HSL, (0,255-sensitivity_2,0), (255,255,sensitivity_2))
white_3 = cv2.inRange(your_image, (200,200,200), (255,255,255))


bit_layer = your_bit_layer | yellow | white | white_2 | white_3
```

OpenCV function or other method has been used to correctly rectify each image to a "birds-eye view". Transformed images should be included in the writeup (or saved to a folder) and submitted with the project.

Great job with the transform! A suggestion would be to first transform the image and then extract the lanes to get the binary image. This will provide cleaner results on the transformed image.

Methods have been used to identify lane line pixels in the rectified binary image. The left and right line have been identified and fit with a curved functional form (e.g., spine or polynomial). Example images with line pixels identified and a fit overplotted should be included in the writeup (or saved to a folder) and submitted with the project.

Nice job fitting polynomials to the extracted lane lines!

Here the idea is to take the measurements of where the lane lines are and estimate how much the road is curving and where the vehicle is located with respect to the center of the lane. The radius of curvature may be given in meters assuming the curve of the road follows a circle. For the position of the vehicle, you may assume the camera is mounted at the center of the car and the deviation of the midpoint of the lane from the center of the image is the offset you're looking for. As with the polynomial fitting, convert from pixels to meters.

Measurements look good. Consider averaging the values in a rotating fashion and update each frame so it appears more smooth to the user. Also use the average process to eliminate outliers by not using them in the average. Otherwise superb job!

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. This should demonstrate that the lane boundaries were correctly identified. An example image with lanes, curvature, and position from center should be included in the writeup (or saved to a folder) and submitted with the project.

Warp looks great, for the different colors, perhaps consider using them in a DEBUG mode oppose to the final product. Its a lot to take in :)

Pipeline (video)

The image processing pipeline that was established to find the lane lines in images successfully processes the video. The output here should be a new video where the lanes are identified in every frame, and outputs

the video. The output here should be a new video where the lanes are identified in every frame, and outputs are generated regarding the radius of curvature of the lane and vehicle position within the lane. The pipeline should correctly map out curved lines and not fail when shadows or pavement color changes are present. The output video should be linked to in the writeup and/or saved and submitted with the project.

Video looks great! There were some polygon errors like so:

In addition to other filtering mechanisms you can use cv2.matchShapes as a means to make sure the final warp polygon is of quality before using. OpenCV's matchShapes, compares two shapes and returns a similarly index, with 0 being identical shapes. You can use this to make sure the polygon for your next frame is close to what it is expected to look like and if not you can elect to use old polygon instead. This way you can let your algorithm be really good when it can and when it can just fake it until a few frames later when it produces good results.

Here is an example:

```
# Early in the code before pipeline
global polygon_points_old
polygon_points_old = None


# In the pipeline

if (polygon_points_old == None):
    polygon_points_old = polygon_points

a = polygon_points_old
b = polygon_points
ret = cv2.matchShapes(a,b,1,0.0)

if (ret < 0.045):
```

```
    # Use the new polygon points to write the next frame due to similarites of last su
cessfully written polygon area

    polygon_points_old = polygon_points

else:
    # Use the old polygon points to write the next frame due to irregularities
    # Then write the out the old polygon points
    # This will help only use your good detections
```

Discussion

Discussion includes some consideration of problems/issues faced, what could be improved about their algorithm/pipeline, and what hypothetical cases would cause their pipeline to fail.

Awesome reflection!

DOWNLOAD PROJECT

RETURN TO PATH