Advanced Lane Lines Finding

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.

How to run

install conda, and create an environment based on the yml file provided

```
conda env create -f environment.yml
```

activate the environment and run the jupyter notebook

```
activate carnd-term1
jupyter notebook
```

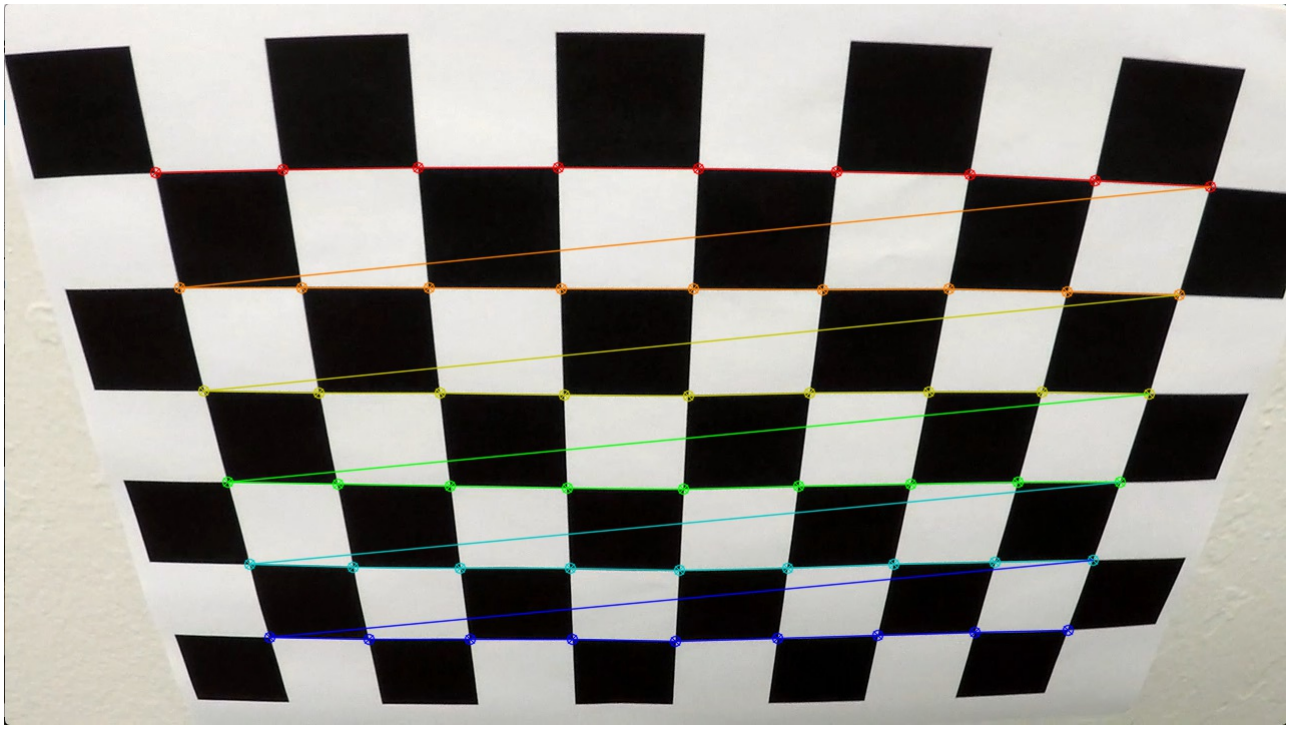
the project is implemented in the P2.ipynb

Camera Calibration

The code for this step is contained in the first (./P2.html#Helper-functions), second (./P2.html#Getting-the-object-and-image-points), and third (./P2.html#Calculating-the-camera-matrix-and-distortion-coefficients) code cells of the IPython notebook located in "./P2.ipynb".

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.

The output images of this stage can be found under this directory (./output_images/CameraCalibration/Corners/)

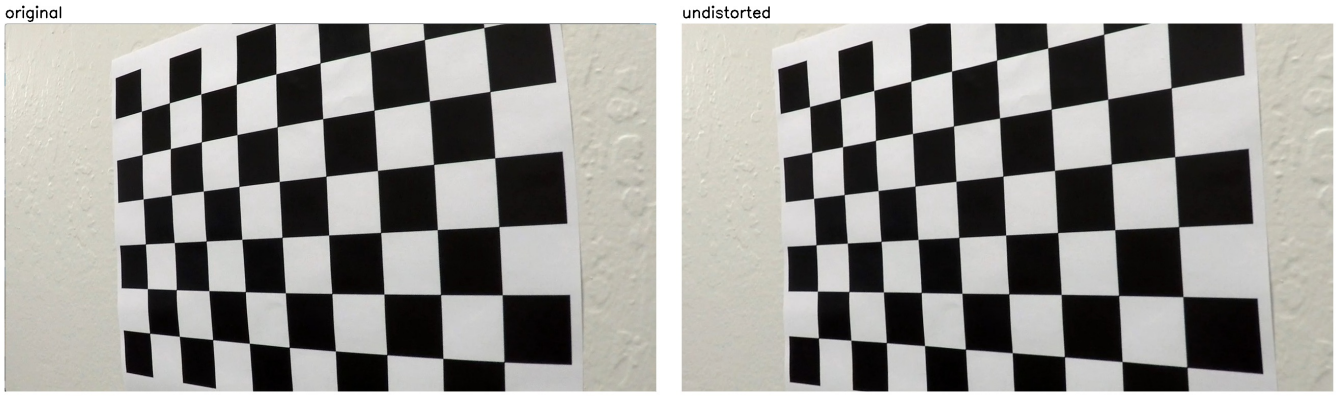


calibration2_corners.jpg (./output_images/CameraCalibration/Corners/calibration2_corners.jpg)

`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. these are stored in a pickel file named `CameraMatrix_DistrotionCoefficients.pickle`. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

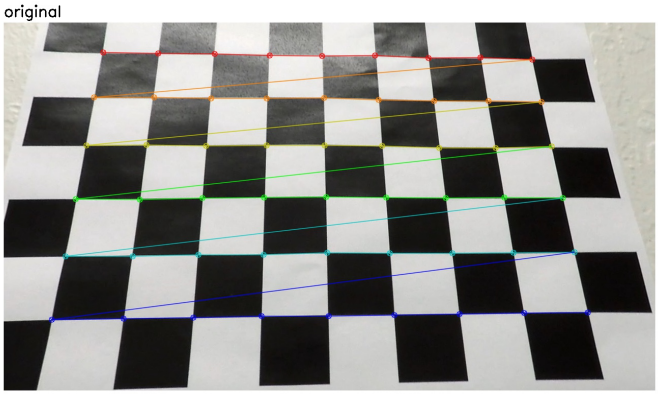
The output images of this stage can be found under this directory (./output_images/CameraCalibration/Undistorted/)

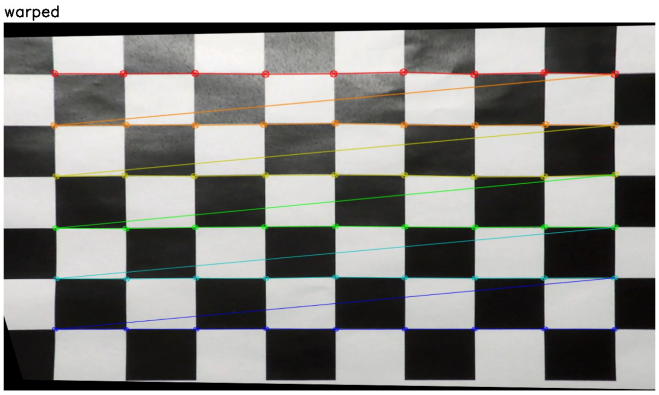


calibration4_undistorted.jpg (./output_images/CameraCalibration/Undistorted/calibration4_undistorted.jpg)

and, finally if the point to be found on the undistorted image, the image is warped using the corners as the reference points and `cv2.getPerspectiveTransform` function. the warped chessboard is as it seen from the front.

The output images of this stage can be found under this directory (./output_images/CameraCalibration/Warped/)





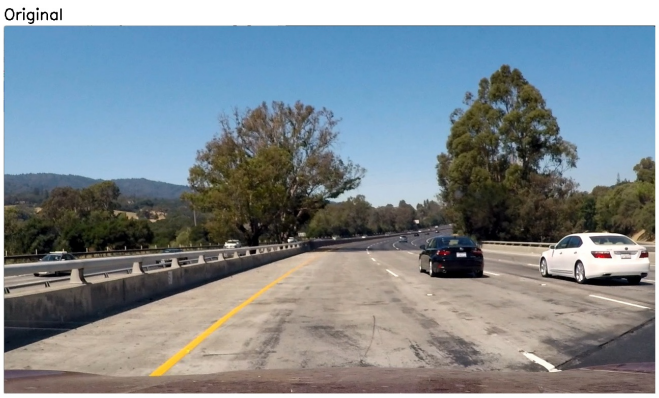
calibration3_warped.jpg (./output_images/CameraCalibration/Warped/calibration3_warped.jpg)

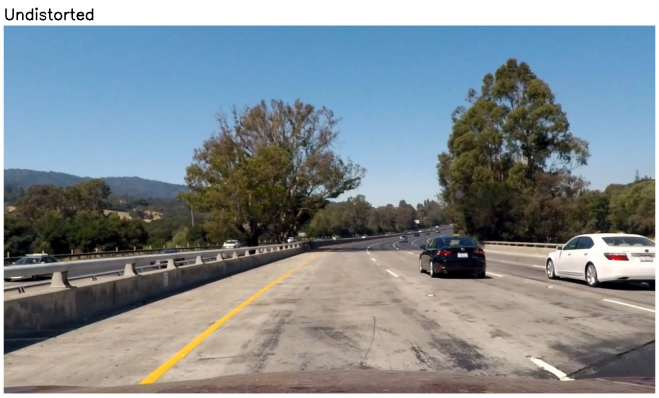
Pipeline overview

the pipeline is defined in the fifth cell (./P2.html#Defining-the-pipeline) of the IPython notebook located in "./P2.ipynb" as the function called `advanced_pipline` in the `RobustLaneLineDetection` class. The `RobustLaneLineDetection` class contains the pipeline, lane line sanity check, and necessary definitions like the instances of the left lane, right lane, and the average of these two called lane. these are instances of the `Line` class. this class holds the memory of the last 10 successful fitted polynomial and x values of the fitted line. also, whether or not the line is detected in last try and number of the total and consecutive failed fits are stored in this class. the main use of this class is in the video process. However, the `RobustLaneLineDetection` and `Line` classes are designed so they can be used for single image as well. simply, for each new image, a new instance of the `RobustLaneLineDetection` should be created.

1. Distortion correction

The camera calibration matrix and distortion coefficients obtained during the camera calibration step are passed to the constructor of the lane line detection class. these are used to undistort the image passed to the pipeline since the calibration parameters are independent from the camera angle. like before, using `cv2.undistort()` function, the undistorted image is created:





test1_stage1_undistorted.jpg (./output_images/test1_stage1_undistorted.jpg)

2. Color and gradients transforms

the goal of this stage is used to generate a thresholded binary image using the combination of color and gradient thresholds .

Color transform and color thresholding

Color transform is defined in the `advanced_pipeline` function in the fifth cell (./P2.html#Defining-the-pipeline) of the IPython notebook located in "./P2.ipynb". first, the image is converted to the HLS color space. this gives three channels: l-channel, s-channel, h-channel. the s-channel is good at detecting the light color where l-channel or gray-scale transform loses the information. minimum 170 and maximum 255 thresholds is applied on the S-channel and the binary output as follows: (these values are chosen so the lane lines are preserved while the other colors are canceled)

test1_stage3_clrThrsh.jpg (./output_images/test1_stage3_clrThrsh.jpg)

Sobel operator and gradient thresholding

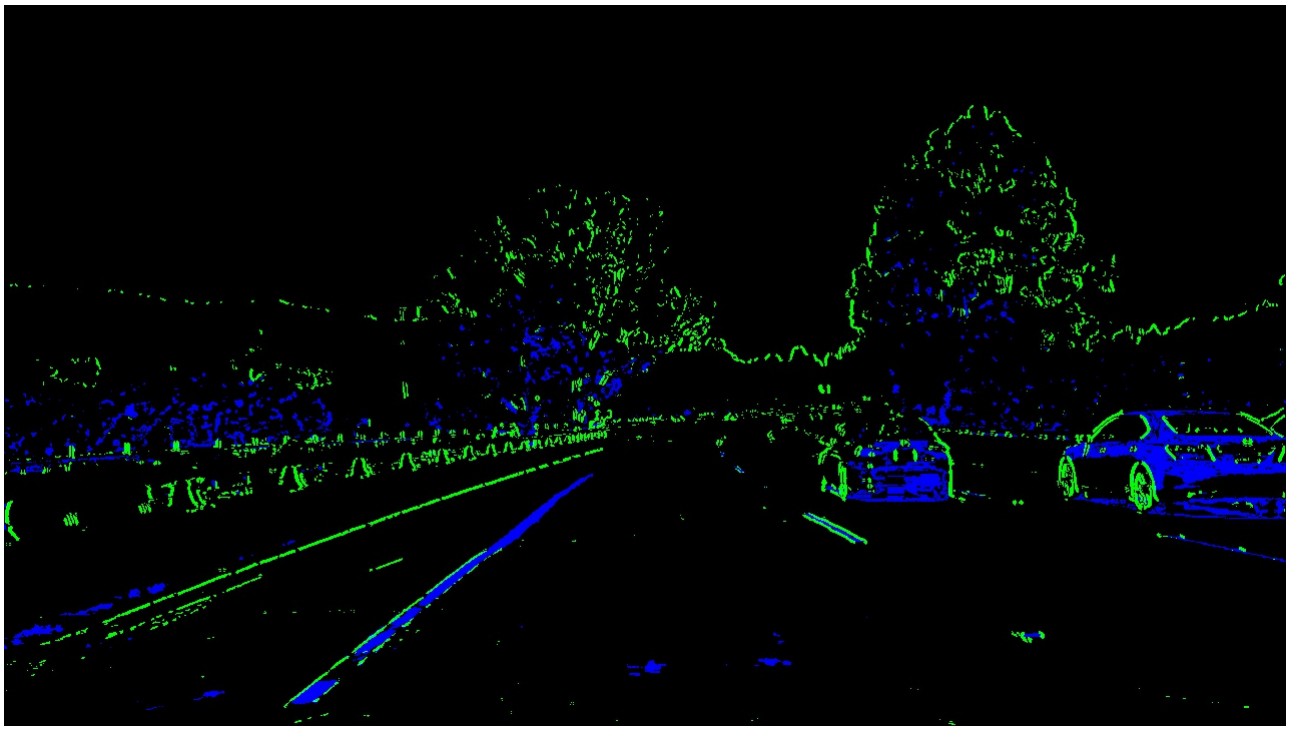
the Sobel operator performs gradient on the l-channel to detect the edges of the objects. The Sobel operator and gradient thresholding are defined in the `abs_sobel_thresh`,`mag_thresh`, and `dir_threshold` functions in the forth cell (./P2.html#Helper-Function) of the IPython notebook located in "./P2.ipynb". these functions are calculating the sobel operator thresholding over x or y, gradient magnitude thresholding, and gradient direction thresholding respectively. Sobel operator is done by `cv2.Sobel` function which accept the gradient kernel size too. with a larger kernel size the gradient would be smoother.

for the purpose of lane line detection, first sobel operator is calculated over the x axis. this detects the roughly vertical edges in the image. then gradient magnitude and direction are combined, detecting the almost vertical gradients which are large enough. the result of this two operation are added and the output of the gradient thresholding is like the following image:

straight_lines1_stage2_gradThrsh.jpg (./output_images/straight_lines1_stage2_gradThrsh.jpg)

Combinition of the color and gradient thresholding

the following image illustrates how these two complete each other: where color thresholding(in blue) fails to detect the lane line gradient thresholding(in green) finds the lane and vice versa.



test4_stage4_clrGradThrsh.jpg (./output_images/test4_stage4_clrGradThrsh.jpg)

the thresholded binary combinition image looks like this:

test3_stage5_binary_combo.jpg (./output_images/test3_stage5_binary_combo.jpg)

3. Perspective transform

The goal of this stage is to transform the front-facing camera image to the birds-eye view. in the front-facing image, the lane lines converge because of the perspective and depth of the scene. this transform makes it possible to fit parallel lines to the lanes and measure the lane cuvature.

The code for my perspective transform includes a function called `unwarp()`, which appears in the first code cell (./P2.html#Helper-functions) of the IPython notebook located in "./P2.ipynb". The `unwarp()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

Source	Destination
229, 700	300, 720
595, 450	300, 0
1068, 700	1000, 720
687, 450	1000, 0

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

straight_lines1_stage6_warped_src_dest.jpg (./output_images/straight_lines1_stage6_warped_src_dest.jpg)

The binary image obtained from the previous stage after being warpped:

test3_stage7_warped_binary.jpg (./output_images/test3_stage7_warped_binary.jpg)

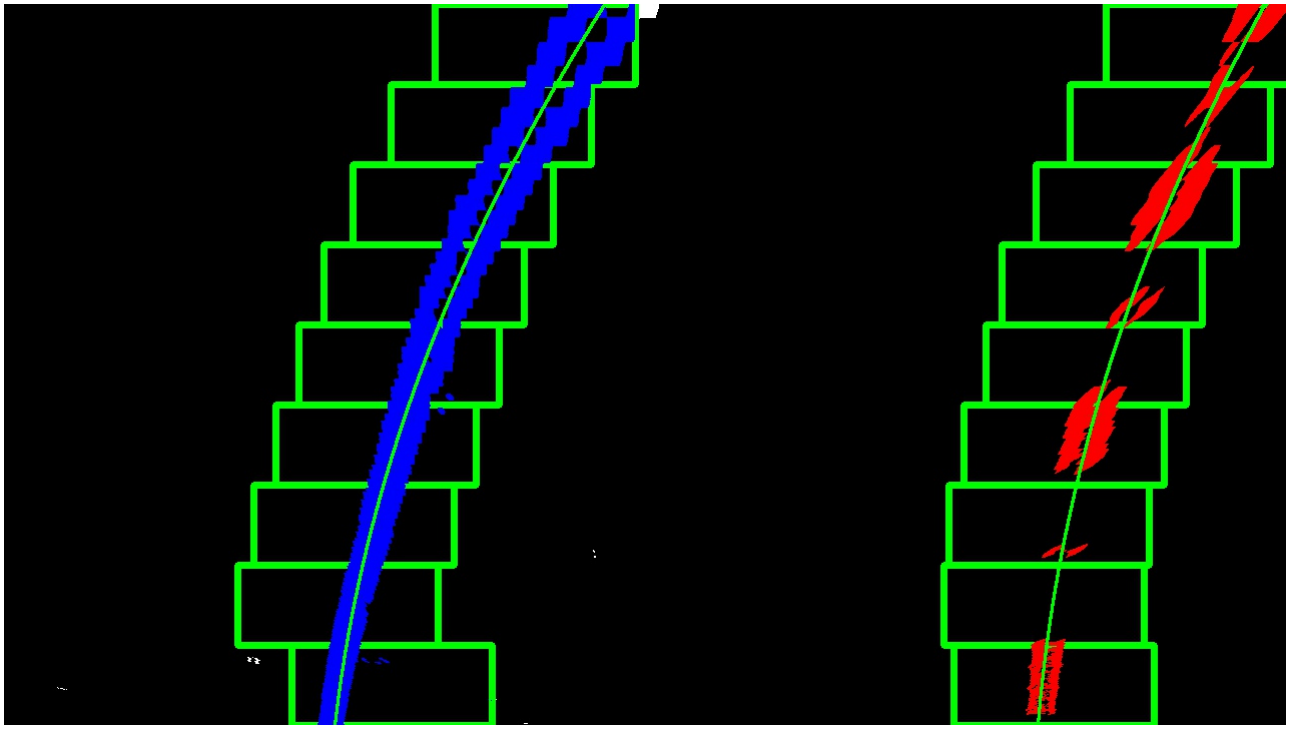
4. Lane line detection and curve fitting

this stage identifies the pixels for each lane line, then fits a 2nd degree polynomial to those pixels.

Identifying the lane line pixels: Sliding Window Method

the code for this method can be found in the `fnd_ln_pxls_sldng_wndw()` function in fourth cell code (./P2.html#Helper-Function) of the IPython notebook located in ./P2.ipynb (./P2.ipynb).

this function divides the warped binary image from the last stage into two halves and calculates the maximum over left section and the right section of the histogram of the bottom half since the lane line pixels usually are detected stronger closer to the car. for each lane line, this gives the starting point for sliding windows. the number of sliding windows determines the height, and the marging,how wide is the search, determines the width of each sliding window. at each iteration the active pixels enclosed in the sliding window are categorized as the lane line pixels, and if the number of these pixels exceeds the `minpx` parameter the center of the sliding points slides (hence the name) to the mean value of the found pixels. in this way, if lane line turns, the sliding window follows the change in x value.

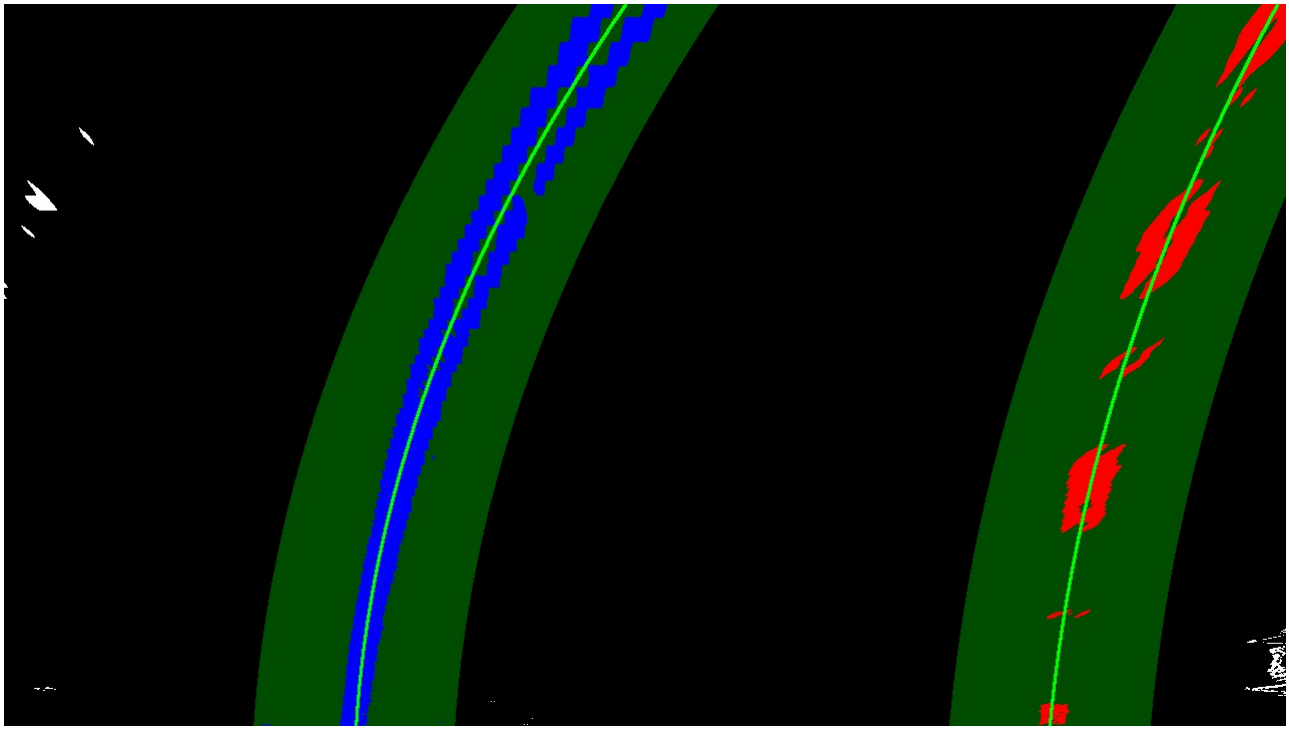


test3_stage8_fit_curve.jpg (./output_images/test3_stage8_fit_curve.jpg)

Identifying the lane line pixels: Prior Search Method

the code for this method can be found in the `fnd_ln_pxls_srch_prior()` function in fourth cell code (./P2.html#Helper-Function) of the IPython notebook located in ./P2.ipynb (./P2.ipynb).

this method expedites the lane line detection when the pipeline has detected the lane lines in the previous frames. it is worth mentioning here that the lane lines are instances of the `Line` class defined in the fifth cell code (./P2.html#Defining-the-pipeline) of the IPython notebook located in ./P2.ipynb (./P2.ipynb). using the best fitted polynomial (average over the last 10 fitted) stored in each lane lines instance, and knowing that the lane lines don't move that much between frames of the video, this function defines the search area around the best fitted curve and categorizes the active pixels as the left or right lane line pixels.



project_video_1_stage8_fit_curve.jpg (./output_images/project_video_1_stage8_fit_curve.jpg)

Curve fiting

the code for this stage can be found in the `fit_polynomial()` function in fourth cell code (./P2.html#Helper-Function) of the IPython notebook located in ./P2.ipynb (./P2.ipynb).

if enough pixels are found for each lane line, the `fit_polynomial()` function is called to fit the 2nd degree polynomial to this pixels. the output of this function is the array of three numbers for the polynomial and the warped image with the lane lines drawn.

Sanity check

the code for this stage can be found in the `sanity_check()` function in fifth cell code (./P2.html#Defining-the-pipeline) of the IPython notebook located in ./P2.ipynb (./P2.ipynb).

this function checks whether the lane lines are parallel, have the similar curvature, and in the right distance from each other.

parallel:the lane lines are consider parallel if the distance between them in the bottom of the image is roughly equal to the distance at the top of the image.

curvature: if the first and second coefficients of the polynomials are roughly equal, the lane lines consider to have similar curvatures. the lane lines are consider vertical if the curvature is bigger then 4km; in this case, they have similar curvature.

distance: the distance between two lane lines is measured in meters and compared to the US road regulation (3m to 3.7m), if the distance between lane lines is between these margins with some threshold, they are considered to have the right distance.

if the sanity check succeeds the value of current fit is stored in the line instance along with the x values. also, the average of the last fitted polynomials (up to 10) is calculated and stored as best fit.

if the sanity check fails more than 10 consecutive frames, the prior search might have lost track of the lane lines; hence, the history kept by the lane line is wiped out, and the sliding window method is called next time to find the lane lines again.

the result of the reset and lane line detection failure can be observed in the challenge video output (./challenge_video_output.mp4).

5. Calculating the radius of curvature of the lane and the position of the vehicle with respect to center.

the code for this stage can be found in the `calculate_curvature()` and `calculate_offset` functions in fourth cell code (./P2.html#Helper-Function) of the IPython notebook located in ./P2.ipynb (./P2.ipynb).

radius curvaure

the fitted polynomial to the lane line is expressed in pixels, so a conversion between these space has to be performed to get the raduis of the curvature in the meters. the lane line polynomial looks like this with the A, B, and C being the coefficients:

$$x_{pixels} = Ay_{pixels}^2 + By_{pixels} + C$$

and the conversion from pixels to meters is as following:

$$x_{meters} = M_x x_{pixels} \overset{where}{\rightarrow} M_x = \frac{3.7m}{700px}$$

$$y_{meters} = M_y y_{pixels} \overset{where}{\rightarrow} M_y = \frac{30m}{720px}$$

so, the resulting polynomial in meters would be:

$$x_{meters} = \frac{M_x A}{M_y^2} y_{meters}^2 + \frac{M_x B}{M_y} y_{meters} + M_x C$$

the above equation gives us the new polynomial coefficients expressed in meters. based on the proof mentioned here (https://www.intmath.com/applications-differentiation/8-radius-curvature.php) the radius of curvature then is calculated like this:

$$R_{curvature} = \frac{(1 + (2A_m y_m + B_m)^2)^{\frac{3}{2}}}{|2A_m|}$$

where

$$A_m = \frac{M_x A}{M_y^2}$$

$$B_m = \frac{M_x B}{M_y}$$

$$y_m = y_{evaluation(px)} M_y$$

the radius is measured at the bottom of the image so the y in pixels is equal to the height of the image.

offset

the offset from the center of the road is measured by comparing the image middle and the value of the polynomial at the bottom of the image. then this value is multiply by the converasion factor between pixels and meters.

6. Ouput

Here is an example of my result on a test image:

test4_stage10_lane_boundary.jpg (./output_images/test4_stage10_lane_boundary.jpg)

Pipeline (test on video)

Here's a link to my video result (./project_video.mp4)

some frames are lost but in total works fine.

Discussion

first, this approach is a bit slow, on my machine it takes around 5 minutes to process a 1 minute video; therefore, it is not suitable for the real time processing and decision making.

also, if the car is going uphill the birds-eye view of road is not accurate since the source points don't have similar z values. therefore, resulting fitted polynomials would give a wrong radius of curvautre and offset. a possible solution is to adjust the source points according to the road slope.

moreover, road conditions would drastically affect the lane line detection. like at night time, or precipitation.

if any sudden movements like a pot hole occurs, the lane lines will not consider to be ok since they don't match the previous detection.

at the sharp turn where the lane lines are out of the range of view, the lane lines are lost. one possible solution would be to use multiple cameras pointing at different directions on the car.

the other issue is not all active pixels in the bird eye view belong to the lane lines. some belong to other road features like the side guardrails or cracks on the road. one solution would be to detect those objects and

In [ ]: