Advanced Lane Line Finding

In this project, the challenge is to create an improved lane finding algorithm, using computer vision techniques.
This project has following steps

1 Camera calibration

2 Color and gradient threshold

3 Birds eye view

4 Lane detection and fit

5 Curvature of lanes and vehicle position with respect to center

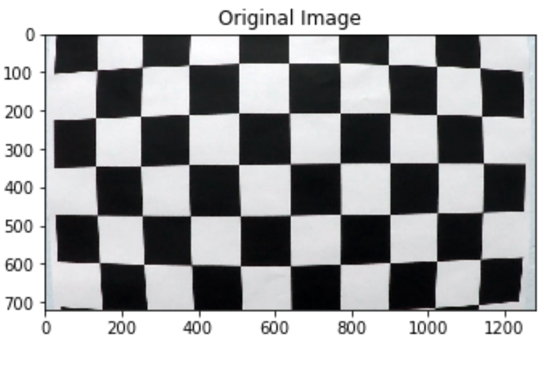
6 Warp back and display information

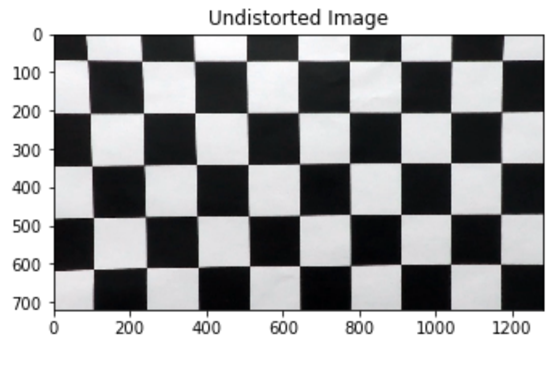
7 Sanity check

Camera calibration

The first step of the project is to do the camera calibration. It is necessary because the lenses of the camera distort the image of the real world. The calibration corrects this distortion.

In our case, we will use chessboards. Chessboards are great, because it is a regular pattern of squares. We know how is the original pattern and how the camera is showing it. Having at least 20 images of chessboards in different angles, we can use opencv function cv2.calibrateCamera() to find the arrays that describe the distortion of the lenses. Then, we can apply cv2.undistort() function.





Here are an example using test images.

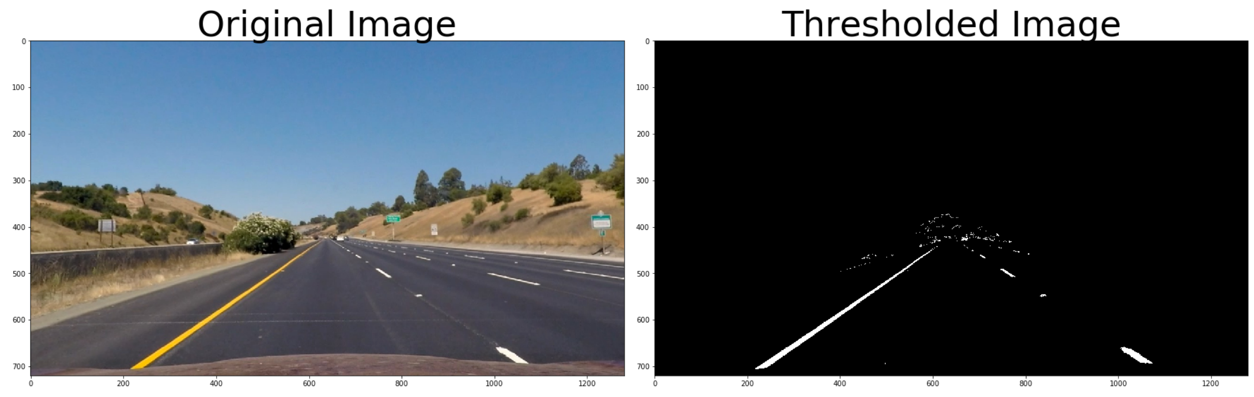
Note the difference in the corners of images

Color and gradient threshold

We use color and gradient thresholds to filter out what we don’t want. We know some features of the lanes: they are white or yellow. There is a high contrast between road and lanes. And they form an angle: they are not totally vertical or horizontal in image.

We do a color threshold filter to pick only yellow and white elements, using opencv convert color to HLS space (Hue, Lightness and Saturation).

We used the combination of Sobel magnitude, sobel direction, RGB color threshold and HLS color threshold.

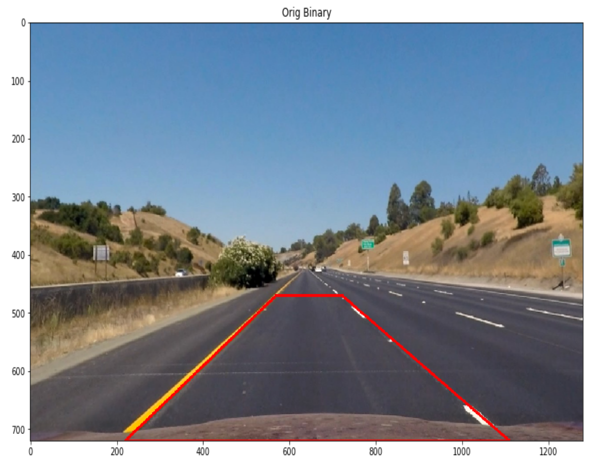


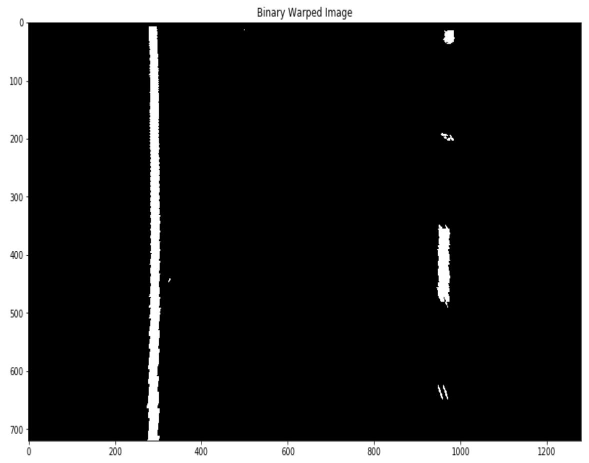
Birds eye view

The idea here is to warp the image, as if it is seen from above. That is because makes more sense to fit a curve on the lane from this point of view, then unwarp to return to the original view.

Here we are considering the camera is mounted in a fixed position, and the relative position of the lanes are always the same.

The opencv function warp needs 4 origins and destinations points. The origins are like a trapezium containing the lane. The destination is a rectangle.

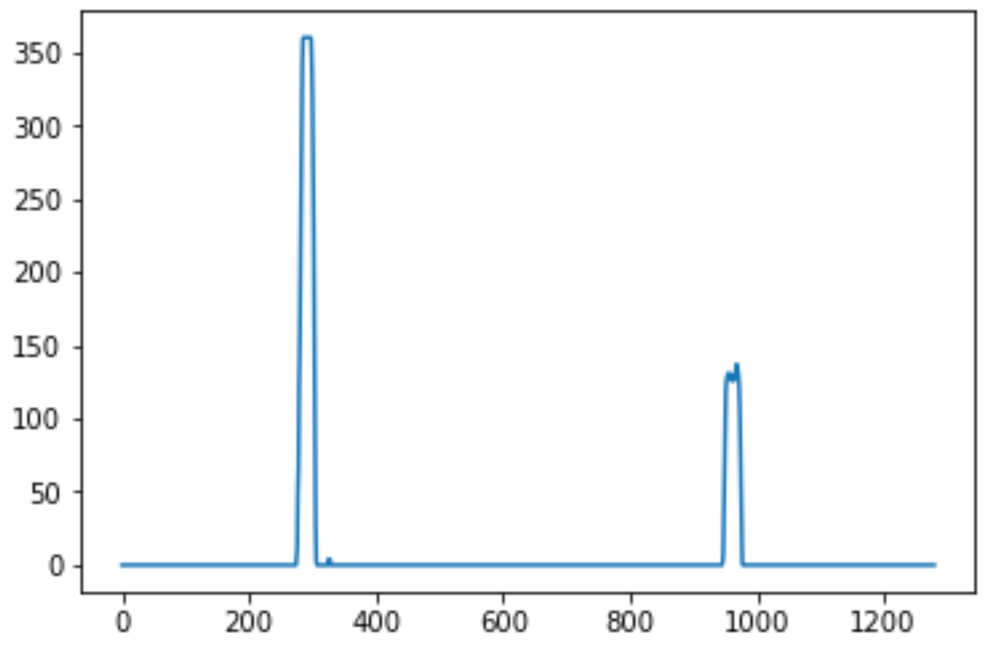




Lane detection and fit

We are using second order polynomial to fit the lane: x = ay**2 + by + c.

In order to better estimate where the lane is, we use a histogram on the bottom half of image.

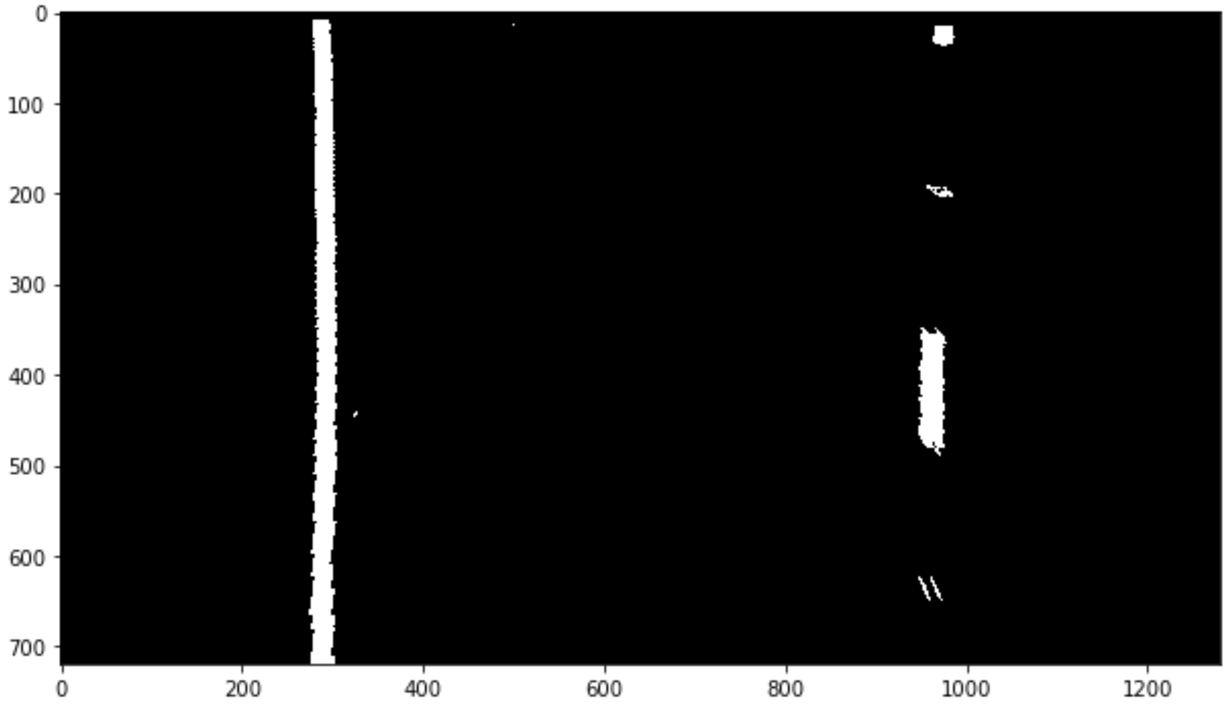


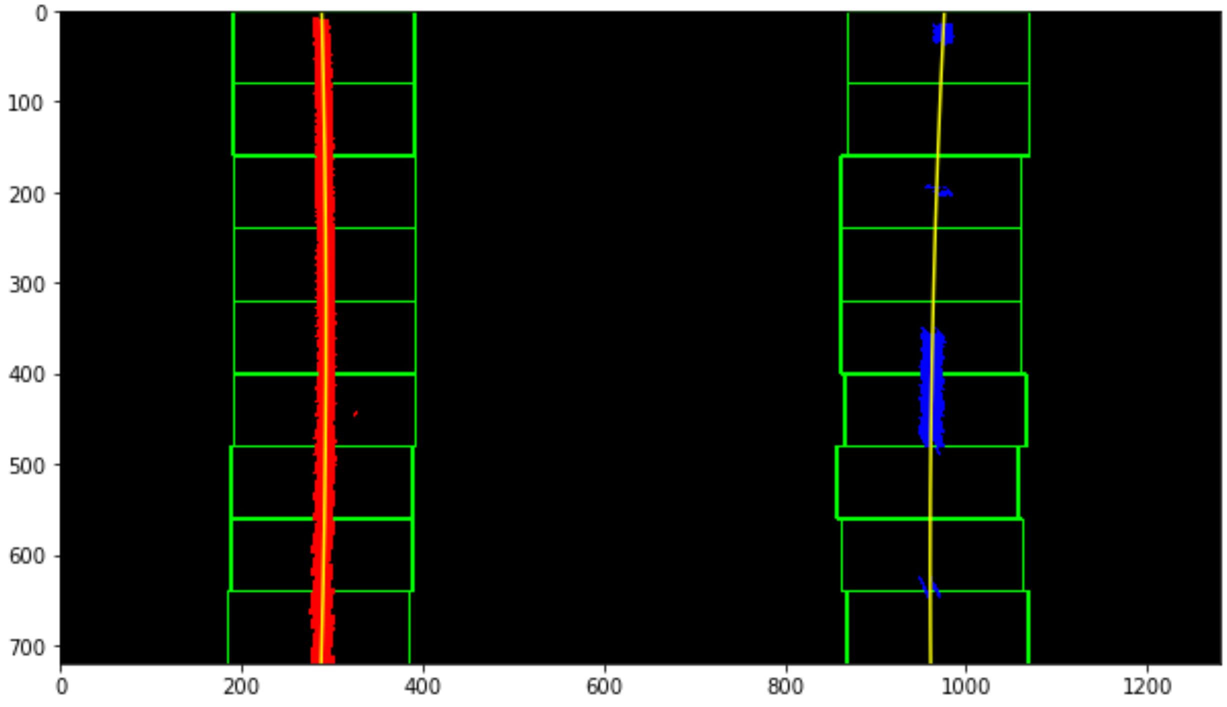
The idea is that the lane has most probability to be where there are more vertical points. Doing this, we find the initial point.

Then we divide the image in windows, and for each left and right window we find the mean of it, re-centering the window.

The points inside the windows are stored.

We then feed the numpy polyfit function to find the best second order polynomial to represent the lanes, as in image:


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Curvature of lanes and vehicle position with respect to center

In a given curve, the radius of curvature in some point is the radius of the circle that “kisses” it, or osculate it — same tangent and curvature at this point.

It is important to know this because it will be a indicative to the steering angle of the vehicle.

The radius of curvature is given by following formula.

Radius of curvature= (1 + (dy/dx)**2)**1.5 / abs(d2y /dx2)

We will calculate the radius for both lines, left and right, and the chosen point is the base of vehicle, the bottom of image.

x = ay2 + by + c

Taking derivatives, the formula is: radius = (1 + (2a y_eval+b)**2)**1.5 / abs(2a)

in the point where x = 0, represented as y_eval in the formula. Another confusion point is that x = 0 if the orientation is upside down, but the coordinates of image is downside - up. Then x = bottom of image, in the case, 720 pixels.

Another consideration. The image is in pixels, the real world is in meters. We have to estimate the real world dimension from the photo.

I’m using the estimative provide by class instructions:

ym_per_pix = 30/720 # meters per pixel in y dimension xm_per_pix = 3.7/700 # meters per pixel in x dimension

Applying correction and formula, I get the curvature for each line.

Example values: 632.1 m 626.2 m

The offset to the center of lane.

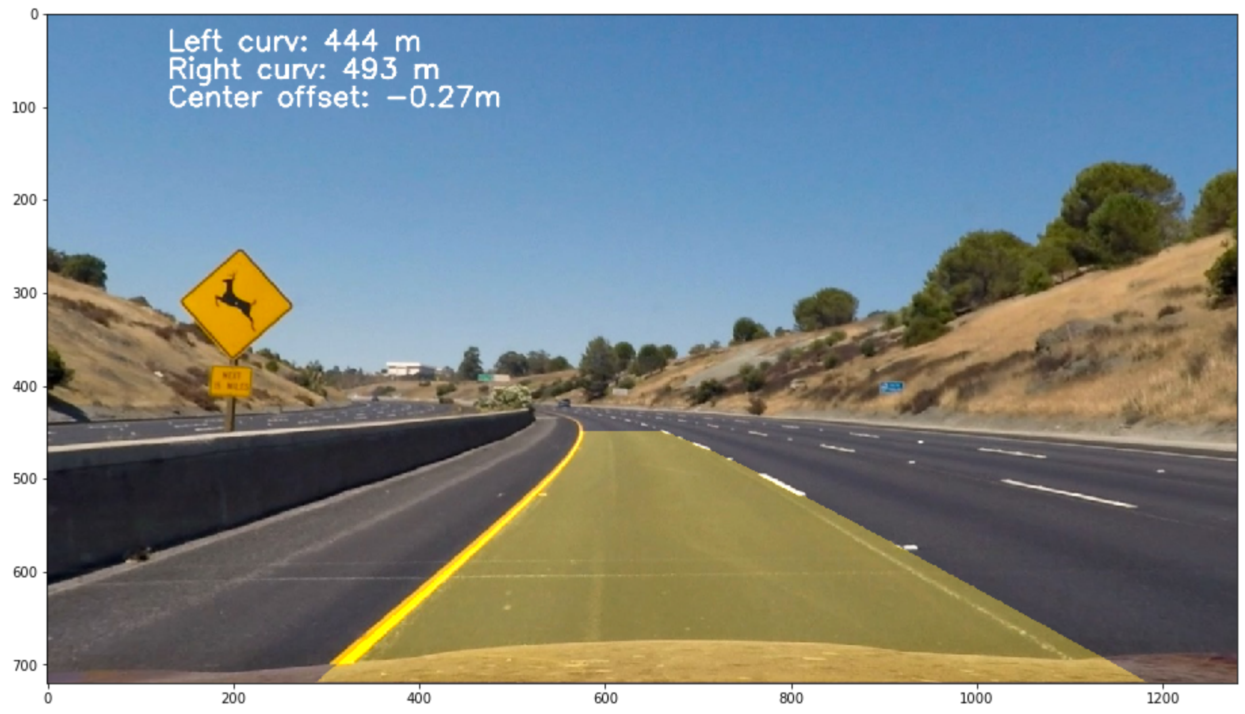
We assume the camera is mounted exactly in the center of the car. Thus, the difference between the center of the image (1280 /2 = 640) and the middle point of beginning of lines if the offset (in pixels). This value times conversion factor is the estimate of offset.

((xL(720) + xR(720))/2–1280/2)* xm_per_pix

Warp back and display information

Once we know the position of lanes in birds-eye view, we use opencv function polyfill to draw a area in the image. Then, we warp back to original perspective, and merge it to the color image.

We can compose the output image with some other images, to form a diagnostic panel. It is easy done, remembering that an image is just an numpy array. We can resize it, and position this resized image in the output image.



Sanity check

- I tried to calculate the difference between lines in 2 points. Did not work, because the width of project video and challenge video is different. So, this method should be tuned for every new lane. Therefore, it is not robust.

- Lines more of less parallel: so derivative in two points have to be about the same. I used this difference of derivatives as a sanity check.

If the lane fit don’t pass the sanity check, we use the last good fit.

Discussion

The technique shown works very well to the situation it was design for. For example, it picks the yellow and white lanes, so it will not work well in situations where panes are blue, or pink, for example. Or when the curves are outside the chosen boundary region (and a too broad region will introduce noise). The over-tuning of parameters will make it not able to generalize the method.

Computer vision techniques are straightforward, in comparison to recognition by deep learning for example. By straightforward, I mean we explicitly define the steps we want to take (undistort, detect edges, pick colors, and so on). By the other hand, in deep learning we do not explicitly choose these steps. Deep learning can make the algorithm more robust sometimes, other times make it fail for reasons nobody knows why.

Perhaps the best conclusion to take is that it is easy to create a simple algorithm that performs relatively well, but it is very hard to create one that will have a human level performance, to handle every situation. There are a lot of improvements to be done. Imagine a lane finding algorithm at night? Or under rain? Perhaps a combination of different approaches can make the final result more robust. Or the algorithm can have different tunings for different roads.

In developing nation, it is a common situation to do not have road lanes at all!

I would suggest if we combine computer vision techniques with deep learning models, it is possible to build a more robust pipeline.