Self-Driving Car: Advanced Lane Detection

1 Project Goals

1. Compute camera parameters(**distortion coefficients and intrinsics**).
2. **Remove radial distortion** from sequence images.
3. Use gradient and color channel **thresholding** to separate lines in binary images.
4. **Remove perspective distortion** from the ground plane in the binary images.
5. **Fit left and right line** as quadratics using the warped (rectified) binary images.
6. **Compute left-right line curvature** at the front of the vehicle (bottom of the image) in meters.
7. **Estimate vehicle offset** from the center of the lane in meters.
8. **Display** results in the original frame.

2 Camera calibration

Synopsis of calibration principles The idea behind camera calibration is to recover two set of parameters:

- The camera **intrinsic parameters** which map image locations to coordinates on a Euclidean projection plane.
- The **distortion coefficients** used to remove image radial distortion.

The idea behind calibration is to use captured images of a chessboard pattern serving as a putative set of planar coordinates in order to recover camera parameters. Chessboard corners are very easy to accurately detect in an image, so camera pose and parameters become the unknowns in a least squares formulation over the projective transformations that map the chessboard to each captured image (aka as *extrinsic parameters*) and the camera distortion and intrinsic parameters.

2.1 The calibrator

Calibration is performed by the *calibrator* class. The calibrator simply looks for a directory containing the images (default name is taken to be the one used in the project repository) and registers the file names. By sequentially loading these images and detecting the corner locations in each, the calibrator builds a list of correspondences between the local 2D coordinate frame of the chessboard and the various camera views obtained in different poses. OpenCV provides $cv2.findChessboardCorners()$ for corner detection and a nice visualization function $cv2.drawChessboardCorners()$; detected corners in a calibration image are shown in Figure 1.

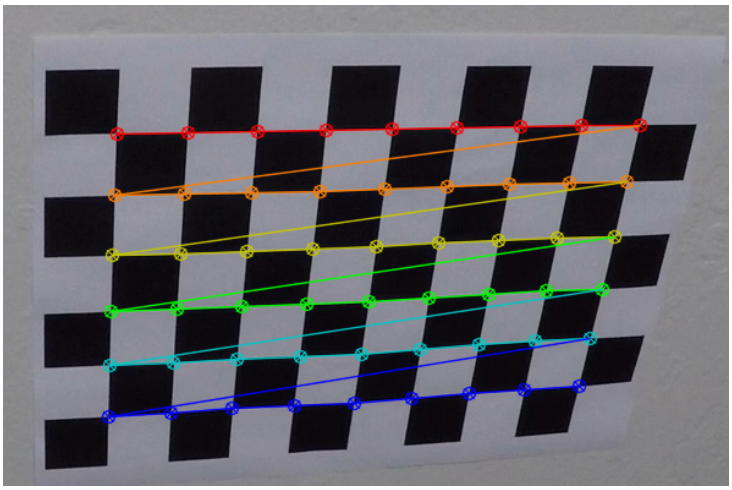
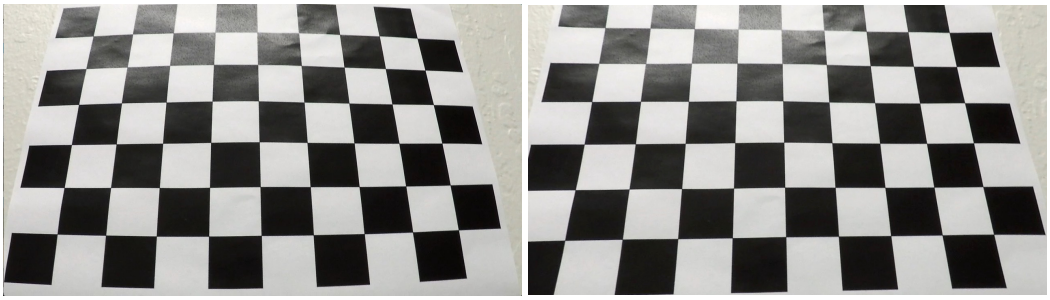


Figure 1: Detected corners in a calibration image.

The calibrator invokes $cv2.calibrateCamera()$ to optimize camera poses and parameters over the entire set of correspondences in the calibration sequence. The distortion coefficients can thereafter be used to remove radial

distortion. In fact, the calibrator offers a method *undistort*() which is invoked in the main loop to remove radial distortion from a frame. Figure 2 illustrates the distorted and undistorted versions of the 3^{d} calibration image.



(a) Distorted image. (b) Undistorted image.

Figure 2: Radial distortion removal from a calibration image (3^{d} image) .

Similarly, Figure 3 illustrates removal of radial distortion from sequence images. Note that lines are generally curved due to perspective, so undistortion is not so apparent to the human eye.

(a) Original sequence image. (b) Undistorted image.

Figure 3: Radial distortion removal from a road sequence image.

3 Method pipeline

The method essentially comprises 3 stages:

1. **Gradient and image channel thresholding** to obtain a binary image with the two lines (along with some residual stuff in other regions of the image).

2. **Cropping of trapezoidal regions** around the lines.

3. **Removal of perspective distortion** from the ground plane.

4. **Fit quadratics** to the left and right line pixels.

5. **Compute curvature and car offset**.

6. **Transform back to regular camera view** and display results.

Preprocessing of the frame is mainly done in the *frame_handler* class. This class is responsible for receiving an undistorted image and processing it in order to produce a projectively rectified binary image to track the lines. The overall binary image preparation is done inside a method *process_frame()*. The projectively rectified binary image is then passed-on to a *LaneTracker* class, which fits the left and right polynomials, computes radii of curvatures and vehicle offset. Method *track_lines()* is the function of *LaneTracker* that does the fitting and delivers an image with the lines drawn on it. Figure 4 illustrates this processing pipeline.

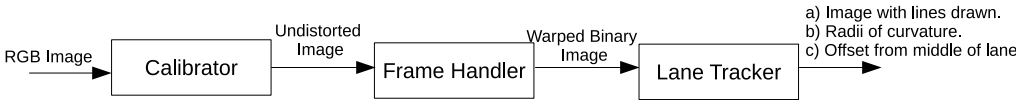


Figure 4: Objects involved and information flow in the pipeline.

3.1 Finding lines with x-gradient and s-channel thresholding

Although probably there is no best way about doing this, in my solution, the thresholding phase essentially involves:

1. Threshold the **grayscale gradient in the x-axis** in the range of $[40, 150]$ (Figure 5).

2. Threshold the **s-channel of the HLS image** in the range of $[100, 255]$ (Figure 6).

3. Crop trapezoidal regions around the lines (Figure 7)

Figure 5: Absolute gradient thresholding in the x-direction. Acceptable range, $[40, 150]$.

Thresholds were picked experimentally based on the overall response of the lane lines in all three videos. I decided on using the s-channel because it clearly enhances the lines, although it produces artifacts above the horizon as shown in Figure 6.

To remove redundant by-products of thresholding, the resulting binary image is cropped so that its contents that fall-out of trapezoidal regions that surround the lines (Figure 7) are set to zero. The red lines in Figure 7 indicate the boundaries of these regions. To compute these lines, we originally consider empirically recovered lines that roughly align with the lane lines when the car is cruising on a line-straight highway section.

$$y = A_l x + B_l \quad (1)$$

$$y = A_r x + B_r \quad (2)$$

where the subscripts l and r denote the left and right side respectively. For each line, we construct two more lines, one on the left and one on the right, by adding and subtracting a large margin from the at the bottom and a slightly smaller at the top (in order to form trapezoidal regions). We represent these lines using the subsripts ll for left-left, lr for left-right, rl for right-left and

Figure 6: Thresholding s-channel in HLS colorspace. Acceptable range, $[100, 255]$.

rr for right-right. Thus, the rule by which a location in the final image is masked, is the following:

$$\begin{aligned} mask\,(x,y) = 1- ((y < A_{ll}x + B_{ll})OR(y > A_{lr}x + B_{lr}))AND \\ ((y < A_{rr}x + B_{rr})OR(y > A_{rl}x + B_{rl})) \end{aligned} \tag{3}$$

The resulting binary image is illustrated in Figure 7.

4 Removing perspective distortion from the ground plane

The final step in the frame handling process is the rectification of the binary image so that the ground plane is displayed without perspective distortion. The idea here is to generate a projective transformation (aka *Homography*) which will transform two lines in the original image that are **known to be parallel in the ground**, into actually parallel lines in a new image. The latter is equivalent to finding 4 points that form a canonical trapezium in

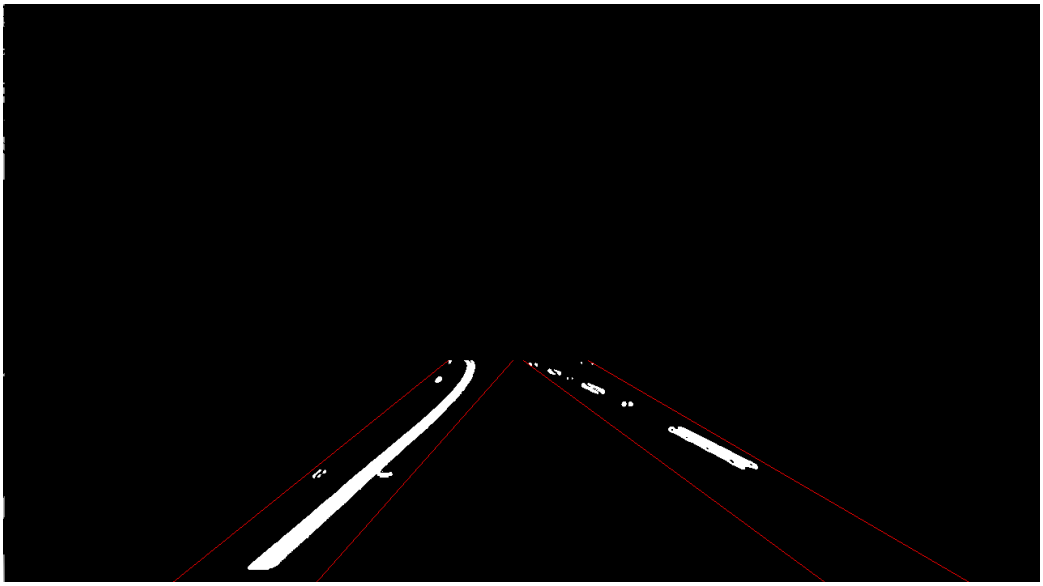


Figure 7: Cropping trapezoidal regions around the lines. The red lines indicate the left and right boundaries of the regions.

the original image and compute a transformation that transforms them into a rectangle. Figure 9 illustrates the transformed image, also known as *bird 's eye view.*

Choosing appropriate points . My strategy was to use a clear view of the left and right line when the car is moving on a section of the highway that appears almost certainly straight. In this case, the lines are going to be sufficiently straight and the respective estimated transformation will be accurate enough. Figure 8 illustrates this trapezium formed by the selection of four points, that, by all appearances lie on parallel lines on the ground plane. The point-coordinates are defined in the following lines (*lane_detector.py*):

```
#left trapezium side
x1 = (int)(cols/4) - 30
y1 = rows-10
x2 = (int)(cols/4) + 247
y2 = rows-(int)(rows/3)
# Right trapezium side
x3 = (int)(cols/4) - 30 + 840
```

```
y3 = rows-10
x4 = (int)(cols/4) + 250 + 160
y4 = rows-(int)(rows/3)

# Destination points (a rectangle)
dx1 = x1
dy1 = y1
dx2 = x1
dy2 = 150

dx3 = x3-200
dy3 = y3
dx4 = x3-200
dy4 = 150
```

and the respective transformation is obtained as follows:

```
H = cv2.getPerspectiveTransform(src, dst)
#Homography : [[ -5.12053432e-01  -1.29964866e+00   8.76345706e+02]
#[ -6.10622664e-16  -1.98959891e+00   9.35445438e+02]
#[  8.67361738e-19  -2.35502835e-03   1.0000000e+00]]
```

The warped image (bird's eye view) ratifies to a significant extent the selection of the 4 points in Figure 8. The right and left line on the lane appear parallel in the warped image in Figure 9 for all intents and purposes.

5 Fitting left and right line

The *LaneTracker* does all the lane tracking work on the warped image. It computes non-zero pixel counts along each row, incrementally along the y-axis in windows (usually 8). The lines are fitted along the peaks of these histograms and all the way to the last window. I haven 't made many changes to this mechanism, although my original intention was to modify the least squares formulation so that the fitted polynomials would be constrained, in, amongst other things, to have a certain minimum distance between them(see more comments in Section 6), but I decided to keep the original concepts intact, mainly because of limited time. There were certain provisions taken to somewhat prevent the search from skewing away from the actual position of

Figure 8: The four chosen points to generate the rectification homography forming a canonical trapezium (in blue color) was to be used by all appearances. The vehicle is cruising steadily in what appears to be a straight section of the motorway.

the line when the non-zero pixel threshold condition is not met and that made the process more robust, but apparently not enough to beat the challenge videos (although it did not do bad).

The first image is treated separately, since there is no prior information as to where the lines actually are. Thus, the bottom histograms are estimated up-to (or down-to) the middle of the image (actually, roughly 3/4 of the total window-height) which implies that we need to have a good view at the beginning. Subsequent images are treated using the previous line estimates as priors. However, that doesn't prevent the search from reaching the other side (i.e., measuring good pixels in the region of the opposite line). Figure 10 illustrates lane detection (fit of quadratics) in the first frame and the fitted quadratics on the next frame, using the previously detected quadratics to initiate search for good pixels.

Figure 9: The warped version (bird's eye view) of the image in Figure 8. The road lines appear practically parallel, which suggests that the estimated transformation is reasonably accurate.

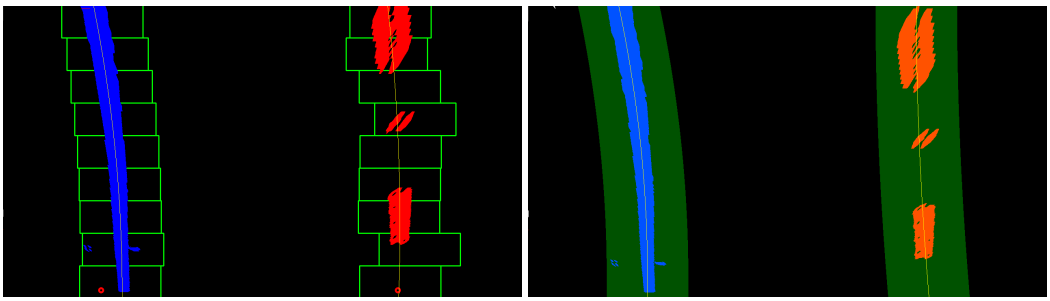
5.1 Radius of curvature and Offset from center of the lane

Having identified the left and right line "good" pixels to be used for fitting the quadratics, then provided a relationship between pixels in the warped image and actual distances in meters, then we can fit the polynomials in meters as follows:

$$x = \frac{1}{\alpha_x}\left(A_2\left(\alpha_y y\right)^2 + A_1\left(\alpha_y y\right) + A_0\right) \tag{4}$$

where (x, y) are pixel coordinates and $(\alpha_x x, \alpha_y y)$ are the corresponding coordinates in meters.

Radius of curvature and Offset from middle of the lane . Now it is very easy to obtain the radius of curvature at a specific height y in the image



(a) Quadratics in the first frame. (b) Quadratics in the next frame.

Figure 10: Fitting quadratics using windows with number of non-zero binary pixels beyond a certain threshold. The red circles indicate the most likely position of the lines across sequences (empirical estimate).

using the standard formula for quadratics:

$$R_{curve} = \frac{\sqrt{\left(1 + \left(2A_2 y + A_1\right)^2\right)^3}}{|2A_2|} \tag{5}$$

where A_2, A_1 are estimated from Euclidean coordinates. Using the fitted polynomials in pixel coordinates, we can find the horizontal position of the vehicle (in pixels) as the midpoint of the quadratic estimates at the bottom of the image (which is actually the largest y-coordinate):

$$offset_{pix} = cols/2 - \left(q_l\left(rows - 1\right) + q_r\left(rows - 1\right)\right)/2 \tag{6}$$

where q_l and q_r are the left and right fitted quadratics in pixel coordinates and $rows$, $cols$ are the rows and columns of the image; then, we may simply convert the offset to meters using ratio α_x:

$$offset_m = a_x offset_{pix} \tag{7}$$

Then putting it altogether in the original image simply requires applying the inverse homography H^{-1} to the points used to draw the quadratics in Figures 10a and 10b. The result is shown in Figure 11.

6 Conclusion

The polynomial fitting scheme appears to be very reliable, provided reasonable binary images in which the lines are prominent. On the other hand, the

Figure 11: Tracked lane in the original (undistorted) image. Radii of curvature from both sides and offset from the center of the lane displayed in the region above the horizon.

method suffers from lack of constraints. This means that when the binary image quality is not very good, the two curves can become intertwined or completely fall off, due to the absence of "good" pixels and the subsequent use of noisy ones to fit the quadratics. This weakness can be partially dealt with by using prior information on the locations of the lines. This is a good strategy, but it does not eliminate problems, since there other factors such as unusual vehicle heading, or very curvy turns (the challenge video for instance) in which the lanes are close to the expected locations, but they are somehow "skewing" away from the predicted/expected location.

Constraints and Particles My original thought on solving this problem was to use additional constraints in the context of a particle filter, in which each particle would be a pair of quadratics in which the lines could have slope difference and distance only within a specific range, thus ensuring the fitted lines will always be reasonably distant from each other and their slopes should be more-less the same across the vertical axis. Measurements now will be targeted by the particles and therefore we wont have to search in specific

regions, thus saving a lot of time for the measurement. Provided a careful implementation, the update step should not take up much time, even with a lot of particles.

Constraints only The same logic can apply without particles. We can simply fit both polynomials of the recovered "good" pixels and impose distance constraints which could be in the form of soft regularization on the constant terms (for distance) and the first derivatives (for slope). The least squares formulation could now be:

$$\begin{aligned} C = \sum_{y_{min}}^{y_max} \left(A_2 y^2 + A_1 y + A_0 - d_l\left(y\right)\right)^2 + \left(B_2 y^2 + B_1 y + B_0 - d_r\left(y\right)\right)^2 + \\ \lambda_1 \left(A_0 - B_0\right)^2 + \lambda_2 \left(2A_2 + A_1 - 2B_2 - B_1\right)^2 + \lambda_3 \left(A_0 - \mu_A\right)^2 + \\ \lambda 4 \left(B_0 - \mu_B\right)^2 \end{aligned} \tag{8}$$

where $d_r\left(y\right)$ and $d_r\left(y\right)$ are the "good" horizontal coordinates corresponding to row y; constants λ_1, λ_2, λ_3, λ_4 are user-defined regularization constants and $\left(2A_2 + A_1 - 2B_2 - B_1\right)^2$, $\left(A_0 - B_0\right)^2$, $\left(A_0 - \mu_A\right)^2$, $\left(B_0 - \mu_B\right)^2$ are soft constraints on slope, distance and position. I very confident that these constraints will make the detection very robust, even when the binary image is very noisy, or has not many good pixels on one side, which is the usual noisy setup. In the case of a totally disastrous image, for instance one that has no good pixels at all, then curvature constraints (e.g., on the second derivative and not necessarily on the curvature formula) will ensure that the detected lane will not bend abnormally.