Advanced Lane Finding

A more in-depth revisit of the lane finding problem.

This project finds lane lines using a more distinctive color space, HLS color space, as well as image gradient features, such as the Sobel gradient features, to filter lane line pixels. We also explore using perspective transformation to convert the image from driver's view to a top-down "bird's-eye" view. We use a 2nd degree polynomial to parameterize the lane line curves and find the curvature of the road. To account for the distortion caused by the camera lenses, we also use chessboard images to calibrate our camera parameters and correct the video frames.

Camera Calibration

The camera calibration consists of two steps, one radial correction

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

And another, the tangential correction:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

So in the end, we need to calibrate for these 5 parameters:

$$Distortion\ coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

These images are courtesy of OpenCV tutorials.

Conveniently, OpenCV has built-in library functions that help us find those parameters.

We use cv2.findChessboardCorners() to find the chessboard corner points in image space, and correspond them to 3D points in a coordinate system whose z axis is always 0 to find the distortion coefficients with the function cv2.calibrateCamera(). We gather these 2D-3D correspondences over multiple images to gain more reliable camera calibration parameters.

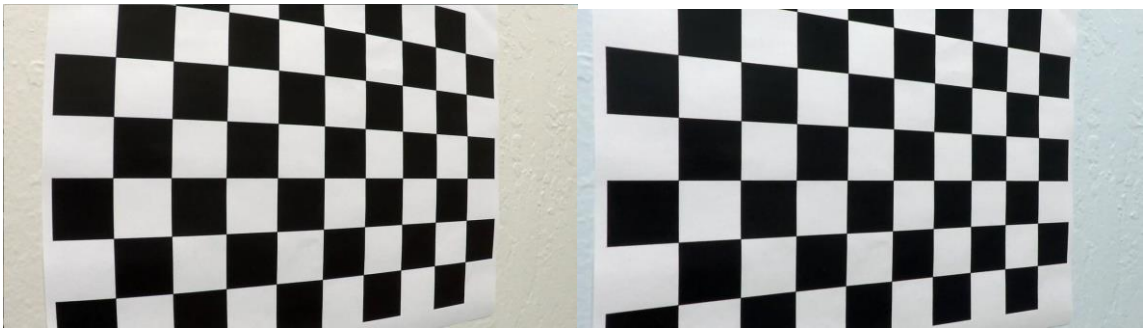

Figure 1. Before and after camera calibration

Image Thresholding

To find the lane lines, we need to first roughly filter out pixels using color value thresholding, and in this example, pixel neighborhood gradient thresholding. For the color thresholding, It turns out that HLS is better at detecting the white and yellow lines on the road better than RGB color space, so that's what being employed here. We primarily use the "saturation" S channel of the HLS color space with some minmax values calculated beforehand using the pick_color.py to threshold the color. This does not seem too robust and particularly filters out too many pixels for stable extraction of lane lines. Therefore we also add the gradient features, specially Sobel operator features, to threshold pixels. This gives us more pixels, and of course more noise to our algorithm. Most of this is from the lesson 30.

This step is done in ImageThresholder class in detect.py (lines 33-69)

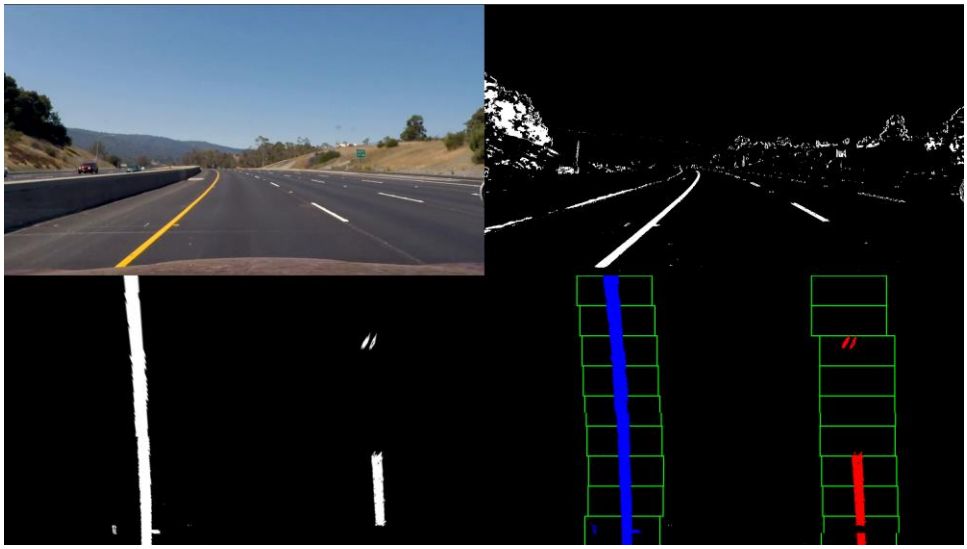

Figure 2: Top-left: original image. Top-right: binary image after sobel_operator. Bottom-left: binary after warping by perspective transform. Bottom-right: left and right lane lines identified using sliding windows.

Top-Down View Perspective Transform

To calculate the lane line curvature it's hard to do it with the driver's side view, which is taken from an angel to the ground. With some geometry we find that warping the image so that the lane lines are parallel to each other, also makes the warped plane of the lane lines parallel to the ground. So I wrote a tool that allows the user to pick out four points that can represent corners of a rectangle on the lane lines, and compute the perspective transform from them. The code for finding rectangle corners is in lane_finding/warp.py (line 34 - 131), and the code for finding transform matrix from two sets of matching points in different coordinates are in lane_finding/warp.py (line 1- 31)

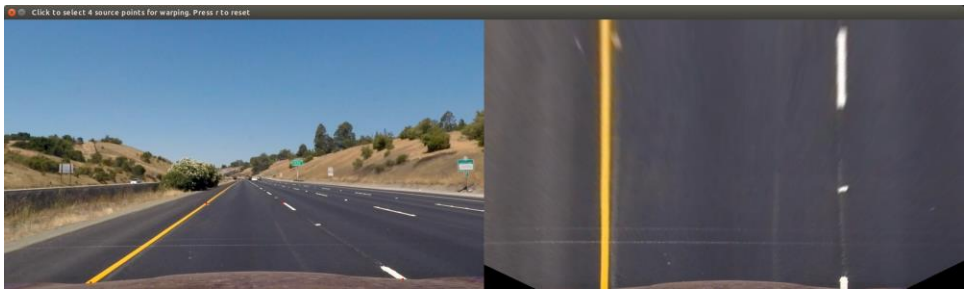

Figure 3: Left is driver's view, right is transformed top-down view. The red dots on the left are corners of an imaginary rectangle that rests on the lane lines.

Lane Line Identification and Parameterization

Here I'm using an approach that is identical to the one taught in class. It's called sliding window approach but I believe it should be called window tracking. There are two steps: initialization and tracking. We split the window into top half and bottom half, and we sum up active pixels in each column of the lower half of the binary image and find the column that has the highest number of hot pixels, which we regard as the x value of a lane line. We do this for both left and right lane, and we use this as a starting point for our tracking. For tracking, we divide the window into N horizontal slices, and we use the x values from the last iteration(of immediate window below) as the center axis for our new window, find all hot pixels in this window, and find the mean x value to be the new x position. We do this for all slices until we reach the top. We add up all pixels we find inside our windows to fit a polynomial curve (lane_finding/detect.py line 72 - 167).

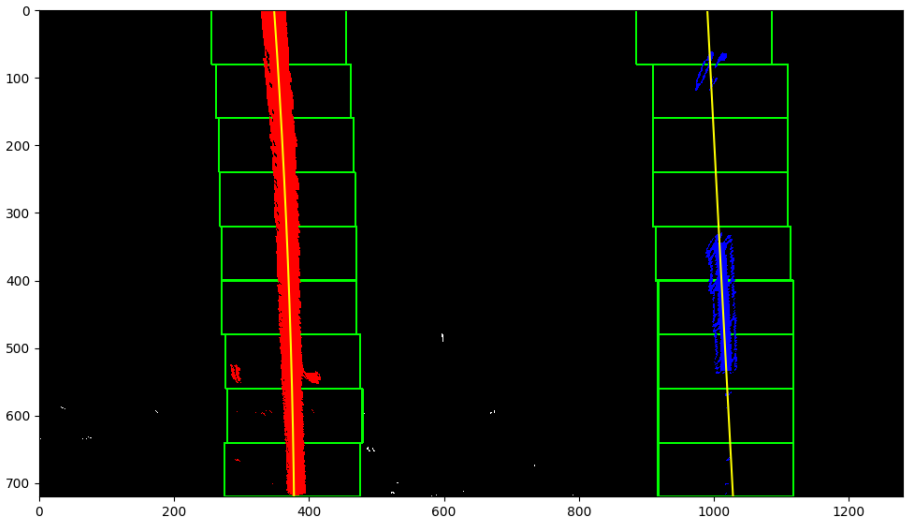

Figure 4. Parameterized lane lines.

Curvature Calculation

We use x=f(y) where f is a 2nd degree polynomial to parameterize the lane lines:

$$f(y) = Ay^2 + By + C$$

and the curvature radius for any function with the form x=f(y) is:

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

Plugging in the derivatives, we get:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

Another thing that we need to take note while the top-down view ensures the lane lines are parallel to each other, it does not ensure the preservation of the aspect ratio. Therefore before we calculate the lane lines, we also scale the x and y pixels to meters to calculate the curvature in real world unit.

We get the A, B and C of the quadratic equation for figure 4. to be:

A, B, C = -1.39337053e-03 3.01816922e-02 2.02852922e+00

And the curvature is: 358.84m.

The code is from lesson 35 (lane_finding/detect.py line 189 – 206)

Lane Area Visualization

Figure 5 – Lane area outlined by lane points.

The visualization code (lane_finding.py line 169 - 187) is taken from lesson 36.

Conclusion

Thresholding the image is definitely the hardest problem as the noise can influence our detection dramatically. By using the color_picker for HLS space to find the threshold, we are able to find the color and sobel threshold more accurately for the entire video. Sliding window lane finding is still too noisy and convolution in this case may be more robust. This algorithm could also use a more robust outlier detection algorithm in order to pass the challenge videos.

Files

output.mp4: Final output video.

lane_finding/: Source code for lane finding module.

output_images/: Images included in this document.