Self-driving Car Nanodegree Program

UDACITY

Project 2: Advanced Lane Finding

Due: 12/25/2018

Duc Tran
ducth1903@gmail.com

I. Objectives

The goals of this project are:

- 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 perspective transform to rectify binary image and obtain "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
- Document the process and findings along the project

II. Procedures

1. Camera Calibration

It is essential to calibrate the camera as the light rays falling into the curved lens of real cameras often bend too little or too much at the edges, resulting in the distortion of images. This phenomenon is also known as radial distortion. Another type of distortion is tangential distortion that happens when a camera's lens is not aligned perfectly parallel to the imaging plane, making the image look tilted so that some objects appear farther away or closer than they actually are. That's why we need to calibrate camera before using it to compensate for the distortion in the images.

Using 20 provided chessboard images taken by a camera, we can use those images for calibration. For each of those chessboard images, I first convert to gray-scaled, then utilize two functions to find chessboard corners and to calibrate camera from Open-CV library, as shown below:

```
# Image points: find corners of the chessboard
gray = cv2.cvtColor(cam_img, cv2.COLOR_RGB2GRAY)   # convert to gray scale
ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
# if detected corners...
if ret:
        img_pts.append(corners)
        obj_pts.append(objp)

# Calibrate camera
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_pts, img_pts, gray.shape[::-1], None,
None)
```

Duc Tran
ducth1903@gmail.com

The last function calibrateCamera() returns the camera matrix, distortion coefficients, rotation and translation vectors. We can then use the resulting camera matrix and distortion coefficients to undistort raw images. The example result is shown in Figure 1.

```
undistorted = cv2.undistort(img, mtx, dist, None, mtx)
```

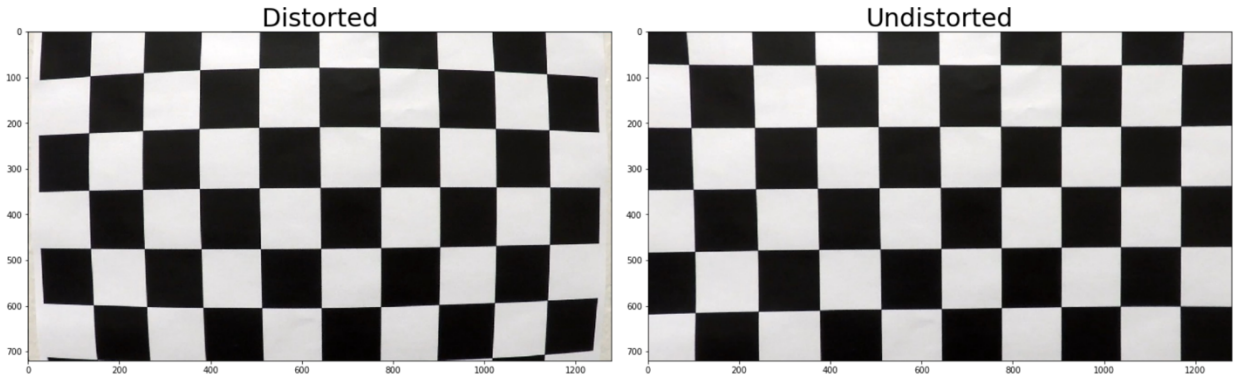


Figure 1: Result of Camera Calibration

2. Pipeline (test images)

The main pipeline is described in the flow chart below.

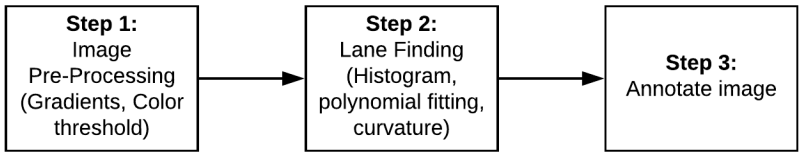


Figure 2: Project 2 Pipeline

We will discuss each of these steps in details in the following subsections.

2.1. Image Pre-processing

In this step, we will pre-process images by applying various techniques such as gradients, color thresholds, and perspective transform. The flow chart for Step 1 is shown below.

Figure 3: Step 1 flow chart

Here, the first step is to undistort the images using the camera matrix and distortion coefficients as discussed in previous section. After trying several combinations of gradients as

well as color thresholds such as RGB, HSV and HLS color space, each has its advantages and disadvantages over others:

- Gradient (Sobel): detect edges well but may detect edges of unwanted shadow as well
- RGB: detect White color well
- HSV: detect other color such as Yellow well but not White
- HLS: can detect both Yellow and White

In case of HSV and HLS, Hue (H) and Saturation (S) stay fairly consistent under lighting conditions. The chosen thresholds are:

- Gradient (Sobel X): (15, 100)
- RGB: (200, 200, 200) -> (255, 255, 255)
- HSV: (13, 100, 100) -> (30, 255, 255)
- HLS: (15, 170, 100) -> (50, 255, 255)

After that, we need to apply perspective transform to obtain bird-eye view of the images. To do this, we need to specify the source and destination points:

```
src = np.float32(
    [[(img_size[1] / 2) - 55, img_size[0] / 2 + 100],
    [((img_size[1] / 6) - 10), img_size[0]],
    [(img_size[1] * 5 / 6) + 60, img_size[0]],
    [(img_size[1] / 2 + 55), img_size[0] / 2 + 100]])

dst = np.float32(
    [[(img_size[1] / 4), 0],
    [(img_size[1] / 4), img_size[0]],
    [(img_size[1] * 3 / 4), img_size[0]],
    [(img_size[1] * 3 / 4), 0]])
```

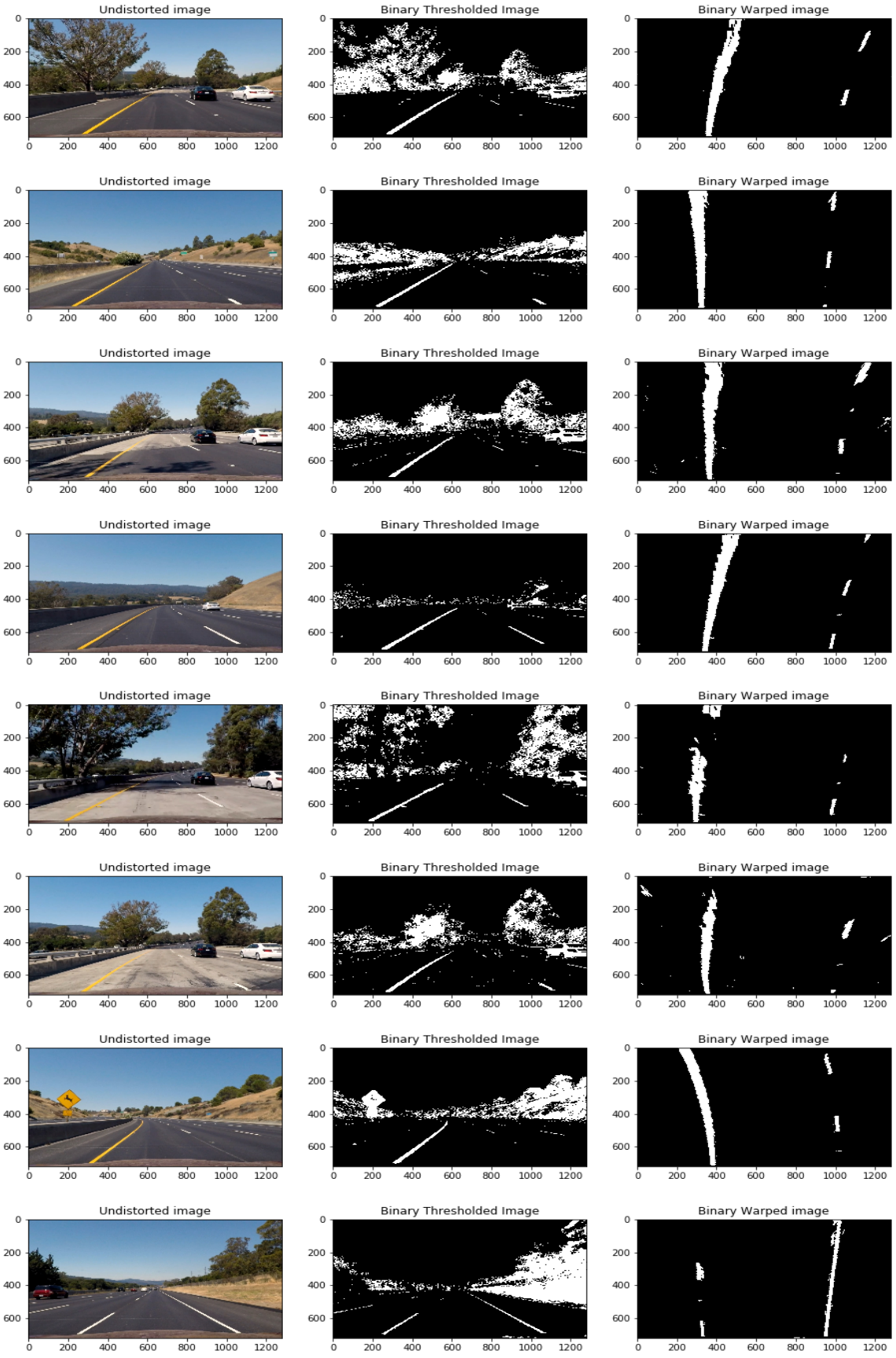
That will be:

src = [(585, 460), (203, 720), (1127, 720), (695, 460)]

dst = [(320, 0), (320, 720), (960, 720), (960, 0)]

The results on test images can be seen below. As we can see, in the warped images, the lane lines are quite parallel, which matches the road in reality.

Duc Tran
ducth1903@gmail.com



Duc Tran
ducth1903@gmail.com

2.2. Lane Finding

The process for lane finding can be found below:

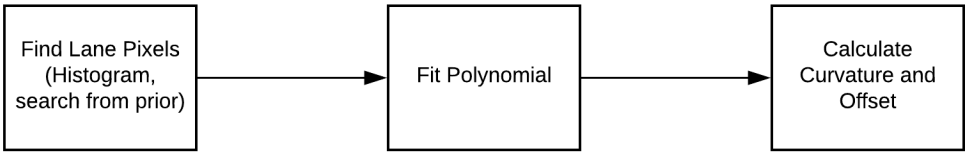


Figure 4: Lane Finding process

First, we need to find the lane pixels using Histogram method for the lower half of the warped image as lanes are most likely straight if looking at the front of the car. Thus, the two highest accumulated pixels in each column will show the lane line positions. The parameters currently used for the sliding window method are:

- Nwindows = 9
- Margin = 100
- Minpix = 30

Next, we can fit polynomial to the detected pixels of the two lanes by using Numpy function, *polyfit()*. Then we also need to calculate the curvature and offset of the car with respect to the lane center. The result for polynomial fitting is shown below. Also, the code showing how to calculate the radius of curvature and offset is in function *measure_curvature_real()* and is shown here.

```
def measure_curvature_real(ploty, xm_per_pix, ym_per_pix):
    left_fit, right_fit = line_class.left_best_fit, line_class.right_best_fit
    # Define y-value where we want radius of curvature
    # We'll choose the maximum y-value, corresponding to the bottom of the image
    y_eval = np.max(ploty)

    ##### TO-DO: Implement the calculation of R_curve (radius of curvature) #####
    temp = xm_per_pix/ym_per_pix
    temp_square = xm_per_pix/(ym_per_pix**2)
    left_curverad = (1+(2*temp_square*left_fit[0]*y_eval+temp*left_fit[1])**2)**(3/2) /
np.absolute(2*left_fit[0]*temp_square)
    right_curverad = (1+(2*temp_square*right_fit[0]*y_eval+temp*right_fit[1])**2)**(3/2) /
np.absolute(2*right_fit[0]*temp_square)

    # Calculating vehicle's position w.r.t the lane center
    left_f = np.poly1d(left_fit)
    right_f = np.poly1d(right_fit)
    bottom_left_lane = left_f(y_eval)
    bottom_right_lane = right_f(y_eval)
    lane_center = (bottom_left_lane+bottom_right_lane)/2    # in pixels!

    return left_curverad, right_curverad, lane_center
```

Duc Tran
ducth1903@gmail.com

Besides the way shown in the lecture video to calculate the curvature, after getting the coefficients for the polynomial fitting curve (parabola in this case), I convert it from pixels to meters:

$$x = \frac{mx}{my^2}ay^2 + \frac{mx}{my}by + c$$

Thus, first and second derivative of x with respect to y are:

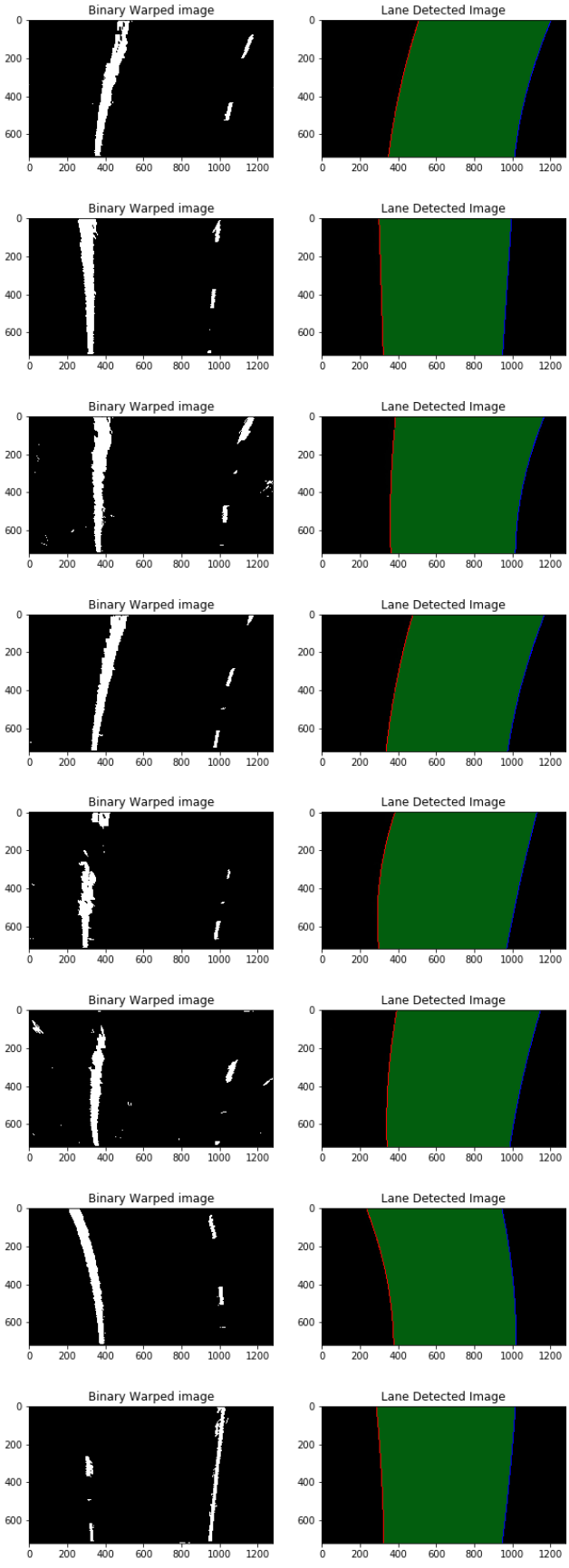
$$\frac{dx}{dy} = 2\frac{mx}{my^2}ay + \frac{mx}{my}b$$

$$\frac{d^2x}{dy^2} = 2\frac{mx}{my^2}a$$

Thus, radius of curvature is calculated as:

$$R = \frac{[1 + \left(2\frac{mx}{my^2}ay + \frac{mx}{my}b\right)^2]^{3/2}}{|2a\frac{mx}{my^2}|}$$

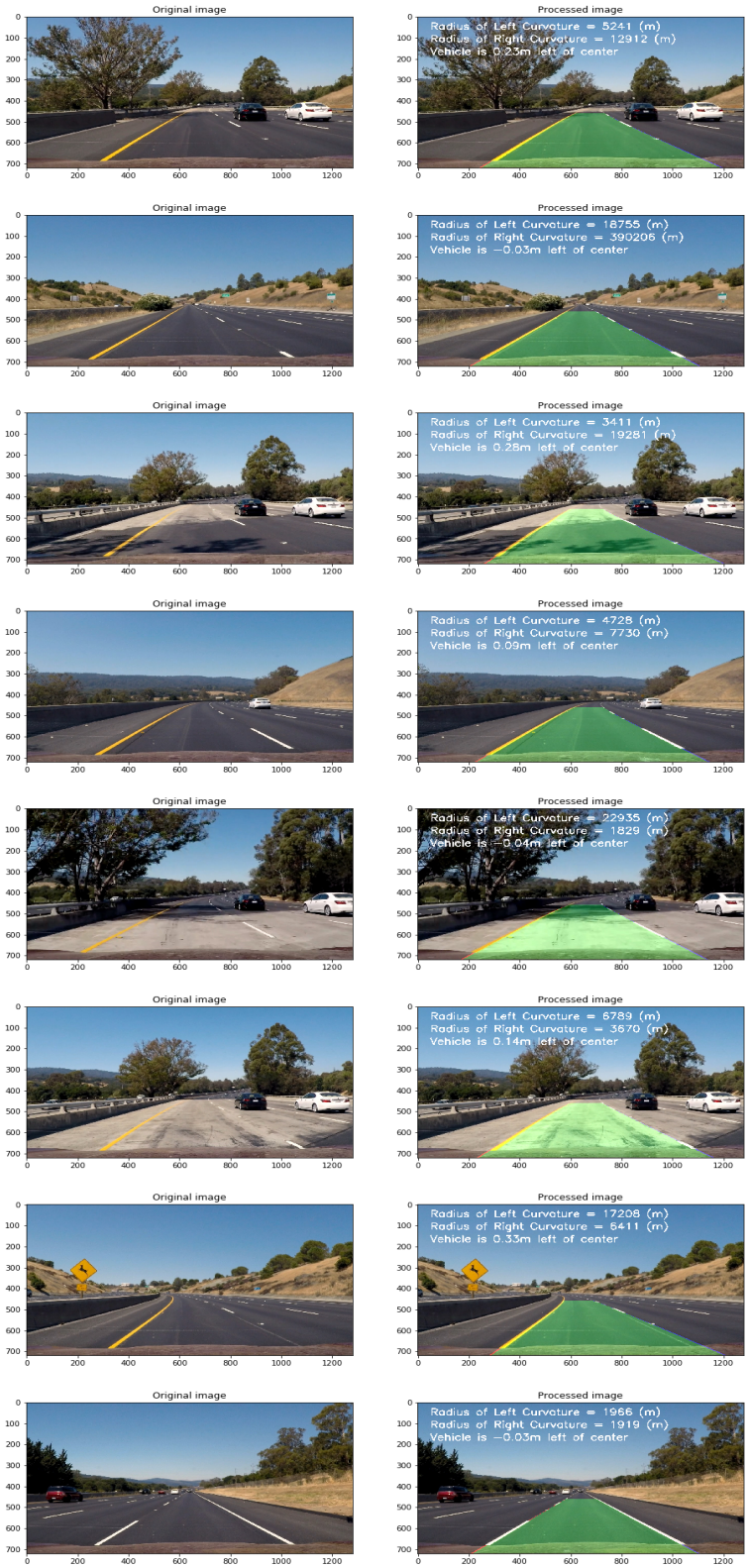
Duc Tran
ducth1903@gmail.com



Duc Tran
ducth1903@gmail.com

2.3. Annotate images

In this final step of the pipeline, we take the lane detected images, along with the radius of curvature and offset to put into the unwarped images. The results of test images are as following.



3. Pipeline (video)

After testing on the test images, we apply the pipeline on video (series of images). The link to the project video can be found here.

III. Discussions (Shortcomings and possible improvements)

3.1. Bad Frames

It is inefficient to do the sliding window to find the histogram every new frame since the lane line is not expected to change much compare to the previous frame. Thus, we can do a more highly targeted ROI search. In particular, instead of sliding window and applying histogram again, we will do a focus search around the polynomial from previous frame. Doing this will improve the speed significantly. This is done in the *search_around_poly()* in the code. In here, I also do sanity check for good/bad frames. The conditions used for sanity check are:

- Useful pixels used for detecting lines should be more than 50% of the total pixels in the image
- The new polynomial should be roughly similar with the previous polynomial as the lines should not change much (poly coefficients should not change much)
- The two detected lines are close to parallel and separate by the right distance horizontally (in this case is approximately 3.7 meters as the standard road is 3.7-meter wide)
- Similar curvature

To keep track of number of bad frames and other information such as the current polynomial coefficients, the best coefficients, etc., I created a class named Line() as suggested by the project instruction. Both lanes with share this class and class's attributes are updated every frame to keep track of the number of bad frames so far and whether there is a need to reset the sliding window after several bad frames. Currently, I set the number of bad frames until reset to **5 bad frames**. When it reaches this number, instead of using previous polynomial, it will run sliding window and histogram method again to re-find the lanes.

3.2. Smoothing

The line detections will jump around from frame to frame a bit and it is recommended to smooth over the last n frames of video to obtain a cleaner and more stable result. Again, I declare an attribute to the Line class above as counter to count number of frames until smoothing out. Currently, I set the smoothing for **every 3 frames**.

3.3. Further improvements

There are still a lot of rooms for improvements, especially for the challenge and harder challenge videos, in which the curve is more rigorous and various lighting conditions as well as shadows. An additional improvement could be implementing outlier rejection and use a low-pass filter to smooth the lane detection over frames (i.e. adding each new detection to a weighted mean of the position of the lines to avoid jitter)