Advanced Lane Finding

Write up

July 18th, 2021 Kenta Kumazaki

1. Purpose

The goals / steps of this project are the following:

- Make a pipeline that finds lane lines on the road and calculate curvature and position.
- Reflect on my work in a written report.

2. Submission

(1) GitHub

https://github.com/kkumazaki/Self-Drivig-Car_Project2_Advanced-Lane-Finding.git

(2) Directory

<folder: main>

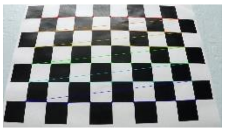
- **Writeup_of_Lesson8.pdf**: This file
- **P2.jpynb**: Pipeline
- Image files video files are saved as following:

<folder: output_images>

<folder: 1_Calibration >

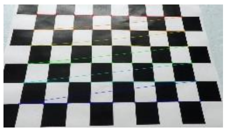
<folder: before_camera_cal>

Chessboard image before camera calibration.



<folder: after_camera_cal>

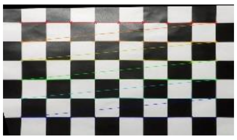
Chessboard image after camera calibration.



<folder: 2_Undistortion>

<folder: Chessboard>

Undistort and perspective transform of chessboard.



<folder: Camera>

Undistort camera images.

<folder: 3_Thresholded>

Color and Gradient threshold images.

<folder: 4_Transform>

<folder: Create>

Straight line images to get transform parameters

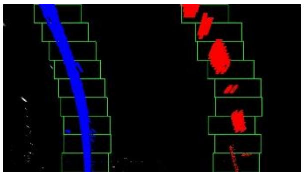
<folder: Adapt>

Adapt parameters and transform curve lines.

<folder: 5_Finding>

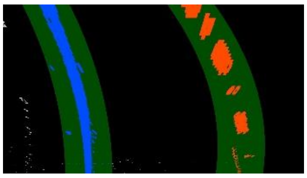
<Histogram>

Calculate by histogram.



<Prior>

Calculate by prior line info.



<folder: 6_Result>

Show the lane line area, curvature and position onto test images.

<folder: test_videos_output>

Output of my pipeline onto project_video.mp4.

3. Reflection

(1)Description of my pipeline

My pipeline consisted of 8 steps.

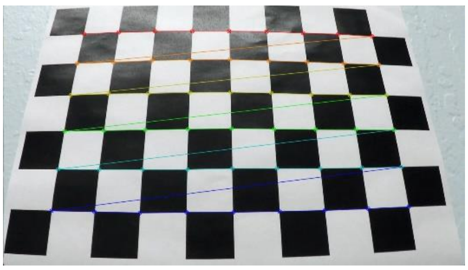
First, I calibrated the camera by using 20 chessboard images.

```python
#1. Camera Calibration

#(1)prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

#(2)Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.

#(3)Make a list of calibration images
images = glob.glob('camera_cal/calibration*.jpg')

#(4)Step through the list and search for chessboard corners
for fname in images:
    img = mpimg.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None)
```

```python

    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)

        # Draw the Chessboard Corners
        img = cv2.drawChessboardCorners(img, (9,6), corners, ret)
        #plt.imshow(img)

        # Get the parameters of Camera Calibration
        ret, mtx, dist,rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)

        # Undistorting test images
        undist = cv2.undistort(img, mtx, dist, None, mtx)

        logger1.debug(fname)
        logger1.debug(mtx)
        logger1.debug(dist)

```

```python
        # Plot original image and undistorted image
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax1.imshow(img)
        ax1.set_title('Original Image'+fname, fontsize=20)
        ax2.imshow(undist)
        ax2.set_title('Undistorted Image'+fname, fontsize=20)
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)

        cv2.imwrite('before_'+fname+'.jpg', img)
        cv2.imwrite('after_'+fname+'.jpg', undist)
```

I show one example of distortion correction below. Data folder is “output_images¥1_Calibration”.



Original image

before_camera_cal /calibration3.jpg

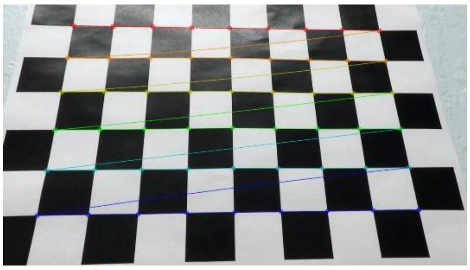
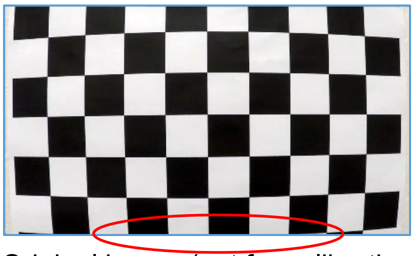


Image after calibration

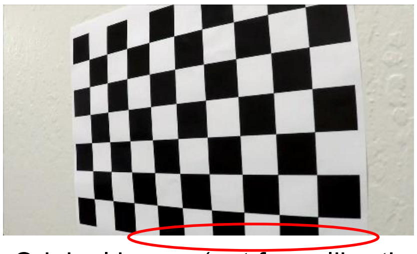
after_camera_cal/ calibration3.jpg

I couldn’t use some images for calibration because there are **not enough cross points** as following.



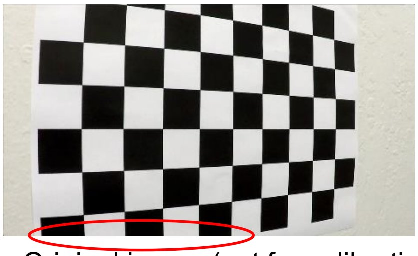
Original image (not for calibration)

camera_cal/ calibration1.jpg



Original image (not for calibration)

camera_cal/ calibration4.jpg



Original image (not for calibration)

camera_cal/ calibration5.jpg

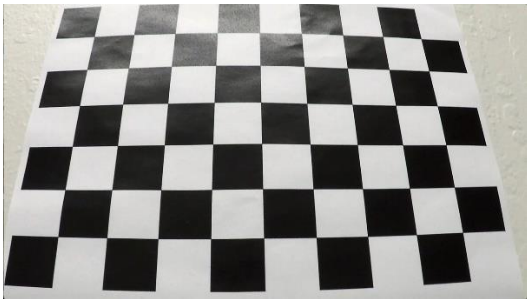
Second, I applied distortion correction and perspective transform of Chessboard.

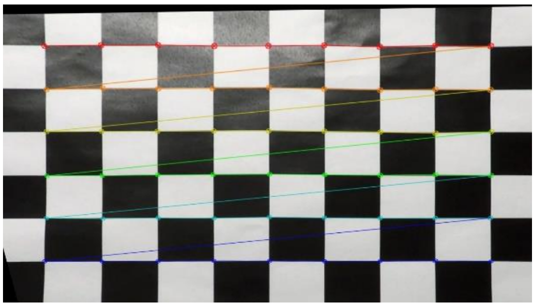
```
#2. Distortion Correction
#Apply a distortion correction to raw images.
#refer Lesson5, section 18

#(1)Distortion Correction and Perspective Transform of Chessboard
#Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
# Read Chessboard in an image
img = cv2.imread('camera_cal/calibration3.jpg')
nx = 9 # the number of inside corners in x
ny = 6 # the number of inside corners in y

# Calculate "M" for Perspective Transform
def corners_unwarp(img, nx, ny, mtx, dist):
    img_size = (img.shape[1], img.shape[0])

    # 1) Undistort using mtx and dist, which were calculated before.
    # Calibration is not needed, because mtx and dist are already exist.
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    img = undist

    # 2) Convert to grayscale
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # 3) Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

    # 4) If corners found:
    if ret == True:

            # a) draw corners
            img = cv2.drawChessboardCorners(img, (nx,ny), corners, ret)

            # b) define 4 source points src = np.float32([[,],[,],[,],[,]])
            src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]]) # Model Answer

            # c) define 4 destination points dst = np.float32([[,],[,],[,],[,]])
            offset = 100 #Model Answer
            dst = np.float32([[offset, offset], [img_size[0]-offset, offset],
                              [img_size[0]-offset, img_size[1]-offset],
                              [offset, img_size[1]-offset]])

            # d) use cv2.getPerspectiveTransform() to get M, the transform matrix
            M = cv2.getPerspectiveTransform(src, dst)

            # e) use cv2.warpPerspective() to warp my image to a top-down view
            warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

    return warped, M

# Show the result of Perspective Transform with Chessboard
top_down, perspective_M = corners_unwarp(img, nx, ny, mtx, dist)
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
f.tight_layout()
ax1.imshow(img)
ax1.set_title('Original Image of Chessboard', fontsize=30)
ax2.imshow(top_down)
ax2.set_title('Undistorted and Warped Image of Chessboard', fontsize=30)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```

The result of distortion correction and perspective transform of Chessboard is following.

It looks distortion correction and perspective transform works well.

Data folder is “output_images¥2_Undistortion”.





Original image

Chessboard/ Original_chess.jpg

Image after distortion correction and perspective transform

Chessboard/ Warped_chess.jpg

And also, I adapted distortion correction to the test images.

```
#(2)Apply a distortion correction to raw images
# Using List doesn't work well, so read each image each time.
straight_lines1 =  mpimg.imread('test_images/straight_lines1.jpg')
straight_lines2 =  mpimg.imread('test_images/straight_lines2.jpg')
test1 =  mpimg.imread('test_images/test1.jpg')
test2 =  mpimg.imread('test_images/test2.jpg')
test3 =  mpimg.imread('test_images/test3.jpg')
test4 =  mpimg.imread('test_images/test4.jpg')
test5 =  mpimg.imread('test_images/test5.jpg')
test6 =  mpimg.imread('test_images/test6.jpg')

# Undistort each image
undist_straight_lines1 = cv2.undistort(straight_lines1, mtx, dist, None, mtx)
undist_straight_lines2 = cv2.undistort(straight_lines2, mtx, dist, None, mtx)
undist_test1 = cv2.undistort(test1, mtx, dist, None, mtx)
undist_test2 = cv2.undistort(test2, mtx, dist, None, mtx)
undist_test3 = cv2.undistort(test3, mtx, dist, None, mtx)
undist_test4 = cv2.undistort(test4, mtx, dist, None, mtx)
undist_test5 = cv2.undistort(test5, mtx, dist, None, mtx)
undist_test6 = cv2.undistort(test6, mtx, dist, None, mtx)
```

I checked images after distortion correction and they looked as good as the examples of Lesson.

Data folder is “output_images¥2_Undistortion¥Camera”.

Distortion Correction image (straight)

undist_straight_lines1.jpg

Distortion Correction image (curve)

undist_test3.jpg

Third, I applied Color Threshold and Gradient Threshold to detect the edges of lane lines.

I used HLS color space and set threshold on S channel because it has better result than other channels in Lesson.

```
#3. Color and Gradient Threshold
#Use color transforms, gradients, etc., to create a thresholded binary image.

#Define a function that thresholds the S-channel of HLS and Sobel gradient
def pipeline(img, s_thresh=(170, 255), sx_thresh=(20, 100)):
    img = np.copy(img)
    # Convert to HLS color space and separate the V channel
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    l_channel = hls[:,:,1]
    s_channel = hls[:,:,2]
    # Sobel x
    sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 0) # Take the derivative in x
    abs_sobelx = np.absolute(sobelx) # Absolute x derivative to accentuate lines away from horizontal
    scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))

    # Threshold x gradient
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= sx_thresh[0]) & (scaled_sobel <= sx_thresh[1])] = 1

    # Threshold color channel
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh[1])] = 1
    # Stack each channel
    #color_binary = np.dstack(( np.zeros_like(sxbinary), sxbinary, s_binary)) * 255
    color_binary = np.dstack((sxbinary|s_binary, sxbinary|s_binary, sxbinary|s_binary)) * 255 #Try showing white color
    return color_binary

# Tune the threshold to try to match the images
hls_straight_lines1 = pipeline(undist_straight_lines1)
hls_straight_lines2 = pipeline(undist_straight_lines2)
hls_test1 = pipeline(undist_test1)
hls_test2 = pipeline(undist_test2)
hls_test3 = pipeline(undist_test3)
hls_test4 = pipeline(undist_test4)
hls_test5 = pipeline(undist_test5)
hls_test6 = pipeline(undist_test6)
```

I checked images after threshold and they looked as good as the examples of Lesson.

Data folder is “output_images¥ 3_Thresholded”.

Image after threshold

thresholded_test1.jpg

Image after threshold

thresholded_test2.jpg

Image after threshold

thresholded_test3.jpg

Image after threshold

thresholded_test4.jpg

Image after threshold

thresholded_test5.jpg

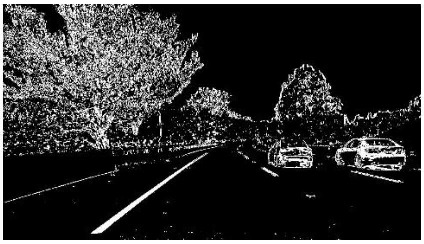


Image after threshold

thresholded_test6.jpg

Forth, I created bird view of the previous threshold mages.

```
#4. Perspective Transform to bird view
#Apply a perspective transform to rectify binary image ("birds-eye view").
#   Should try with 2 straight line images

#(1)Create warp function which fits well with straight lines images.
# Save RGB color images to get good src points (if we use cv2.imwrite(), should be converted)
save_img1 = cv2.cvtColor(undist_straight_lines1, cv2.COLOR_RGB2BGR)
save_img2 = cv2.cvtColor(undist_straight_lines2, cv2.COLOR_RGB2BGR)
cv2.imwrite("undist_straight_lines1.jpg",save_img1)
cv2.imwrite("undist_straight_lines2.jpg",save_img2)

#Warp function
def warp(img):

    # Define calibration box in source and destination
    img_size = (img.shape[1], img.shape[0])

    # Get src from undistorted images
    src1 = np.float32([[708,462],[1053,688],[246,688],[577,462]]) # undist_straight_lines1
    #src1 = np.float32([[662,436],[1053,688],[246,688],[617,436]]) # undist_straight_lines1
    src2 = np.float32([[678,441],[1040,675],[280,675],[607,441]]) # undist_straight_lines2
    src = (src1 + src2)/2 # Use average to get more robust src
    offsetx = 300
    offsety = 0
    dst = np.float32([[img_size[0]-offsetx,offsety],[img_size[0]-offsetx,img_size[1]-offsety],[offsetx,img_size[1]],[offsetx,offsety]])
```

```
    # Compute the perspective transform, M
    M = cv2.getPerspectiveTransform(src, dst)

    # Create warped image
    warped = cv2.warpPerspective(img, M, img_size, flags = cv2.INTER_LINEAR)

    return warped

# Generate warped images from straight lines images
warped_straight_lines1 = warp(undist_straight_lines1)
warped_straight_lines2 = warp(undist_straight_lines2)
```

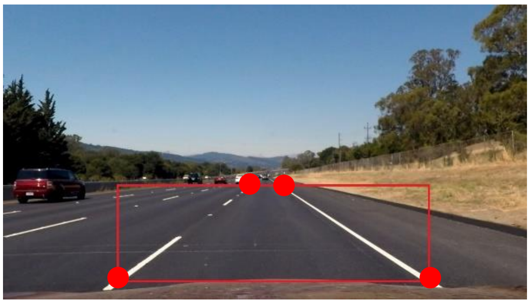
To determine src parameters of Perspective Transform, I used 2 straight line images as below.

I get 2 sets of src (src1, src2) and determine the final src by calculating average of src1 and sec2.

Data folder is “output_images¥4_Transform¥Create”.

src1 points

undist_straight_lines1_get_src.jpg

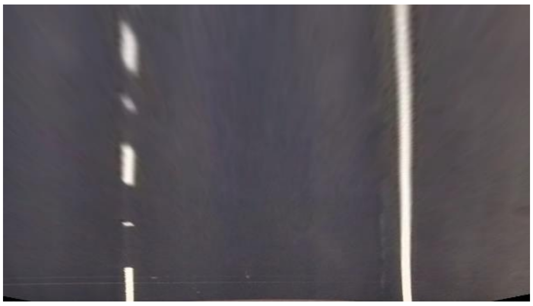


src2 points

undist_straight_lines2_get_src.jpg

Perspective Transform image

warped_straight_lines1.jpg



Perspective Transform image

warped_straight_lines2.jpg

By using those parameters, I calculated Perspective Transform of the Images after threshold and showed the bird view images as below.



<u>Image after Transform</u> warped_test1.jpg

<u>Image after Transform</u> warped_test2.jpg

<u>Image after Transform</u> warped_test3.jpg



<u>Image after Transform</u> warped_test4.jpg

<u>Image after Transform</u> warped_test5.jpg

<u>Image after Transform</u> warped_test6.jpg

Fifth, I found the lane lines in the bird view. There are 4 functions, so I explain step by step.

(1)find_lane_pixels(): Find pixels by using histogram.

```
#5. Finding the Lanes
#Detect lane pixels and fit to find the lane boundary.

binary_warped = warped_test6
file_name = "test6"
original_img = test6

Mode = 1 # Until Step 7: Plot Lane Lines area with green color
#Mode = 2 # Step 8: Plot Lane area with green color for movie file

#(1) Peaks in Histogram by Sliding Window
#Findind pixels by using hystogram
def find_lane_pixels(binary_warped):
    # Take a histogram of the bottom half of the image
    histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)
    # Create an output image to draw on and visualize the result
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))
    # Find the peak of the left and right halves of the histogram
    # These will be the starting point for the left and right lines
    #midpoint = np.int(histogram.shape[0]//2) #There's warning if I use np.int...
    midpoint = int(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

```
    # HYPERPARAMETERS
    # Choose the number of sliding windows
    nwindows = 9
    # Set the width of the windows +/- margin
    margin = 100
    # Set minimum number of pixels found to recenter window
    #minpix = 50 # If it's 50, it's not good at test4.jpg...
    minpix = 200

    # Set height of windows - based on nwindows above and image shape
    #window_height = np.int(binary_warped.shape[0]//nwindows) #There's warning if I use np.int...
    window_height = int(binary_warped.shape[0]//nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    # Current positions to be updated later for each window in nwindows
    leftx_current = leftx_base
    rightx_current = rightx_base
```

```
    # Create empty lists to receive left and right lane pixel indices
    left_lane_inds = []
    right_lane_inds = []

    # Step through the windows one by one
    for window in range(nwindows):
        # Identify window boundaries in x and y (and right and left)
        win_y_low = binary_warped.shape[0] - (window+1)*window_height
        win_y_high = binary_warped.shape[0] - window*window_height
        ### TO-DO: Find the four below boundaries of the window ###
        win_xleft_low = leftx_current-margin  # Update this
        win_xleft_high = leftx_current+margin  # Update this
        win_xright_low = rightx_current-margin  # Update this
        win_xright_high = rightx_current+margin  # Update this

        # Draw the windows on the visualization image
        cv2.rectangle(out_img,(win_xleft_low,win_y_low),
        (win_xleft_high,win_y_high),(0,255,0), 2)
        cv2.rectangle(out_img,(win_xright_low,win_y_low),
        (win_xright_high,win_y_high),(0,255,0), 2)

        ### TO-DO: Identify the nonzero pixels in x and y within the window ###
        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
        (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonzero()[0]
        good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
        (nonzerox >= win_xright_low) &  (nonzerox < win_xright_high)).nonzero()[0]
```

```
        # Append these indices to the lists
        left_lane_inds.append(good_left_inds)
        right_lane_inds.append(good_right_inds)

        ### TO-DO: If you found > minpix pixels, recenter next window ###
        ### (`right` or `leftx_current`) on their mean position ###
        #pass # Remove this when you add your function
        if len(good_left_inds) > minpix:
            #leftx_current = np.int(np.mean(nonzerox[good_left_inds]))#There's warning if I use np.int...
            leftx_current = int(np.mean(nonzerox[good_left_inds]))
        if len(good_right_inds) > minpix:
            rightx_current = int(np.mean(nonzerox[good_right_inds]))#There's warning if I use np.int...
            #rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

    # Concatenate the arrays of indices (previously was a list of lists of pixels)
    try:
        left_lane_inds = np.concatenate(left_lane_inds)
        right_lane_inds = np.concatenate(right_lane_inds)
    except ValueError:
        # Avoids an error if the above is not implemented fully
        pass
```

(2) fit_polynomial(): Calculate fitting polynomial by using find_lane_pixels().

```
#Calculate fitting polynomial
def fit_polynomial(binary_warped):
    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(binary_warped)

    ### TO-DO: Fit a second order polynomial to each using `np.polyfit` ###
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)

    # Generate x and y values for plotting
    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
    try:
        left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
        right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
    except TypeError:
        # Avoids an error if `left` and `right_fit` are still none or incorrect
        print('The function failed to fit a line!')
        left_fitx = 1*ploty**2 + 1*ploty
        right_fitx = 1*ploty**2 + 1*ploty

    ## Visualization ##
    # Colors in the left and right lane regions
    out_img[lefty, leftx] = [255, 0, 0]
    out_img[righty, rightx] = [0, 0, 255]
```

```
    # Plots the left and right polynomials on the lane lines
    #plt.plot(left_fitx, ploty, color='yellow')
    #plt.plot(right_fitx, ploty, color='yellow')

    #return out_img
    return out_img, left_fit, right_fit # Add left_fit,right_fit for the next step

#Get result of fitting polynomial
out_img, left_fit, right_fit = fit_polynomial(binary_warped)
```

The result of histogram is shown below.

Data folder is "¥ output_images¥5_Finding¥Histogram".

I used the same parameters as Lesson at first, but the parameter: **min_pix = 50** didn't work with "test4" image. It calculated the average of the bottom 4th area **(A),** then bottom 5th area moved to left and the right lane was not detected **(B)**.

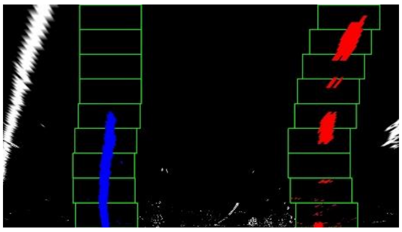


Image after Histogram

Find_from_Histogram_test1_
minpix_50.jpg

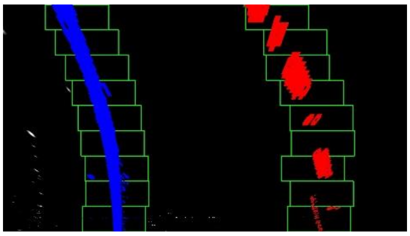


Image after Histogram

Find_from_Histogram_test2_
minpix_50.jpg

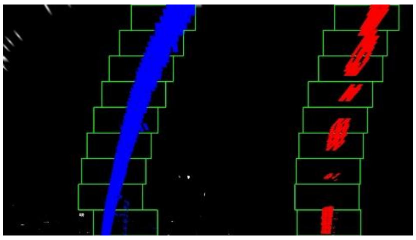


Image after Histogram

Find_from_Histogram_test3_
minpix_50.jpg

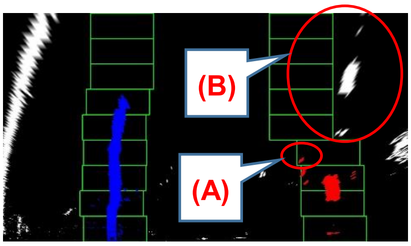


Image after Histogram

Find_from_Histogram_test4_
minpix_50_bad.jpg

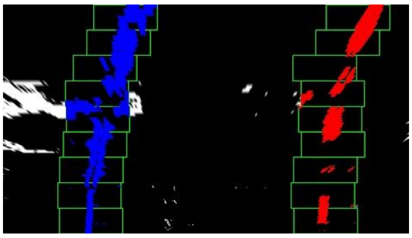


Image after Histogram

Find_from_Histogram_test5_
minpix_50.jpg

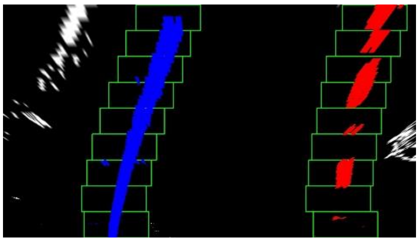


Image after Histogram

Find_from_Histogram_test6_
minpix_50.jpg

To avoid it, I changed parameter: **minpix = 200**, then the problem was solved **(C)**.

I checked other images and there are no degradation.

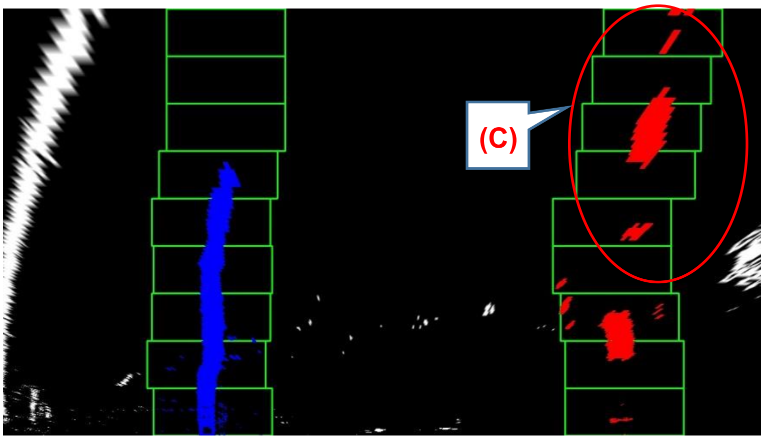


Image after Histogram with parameter tuning

Find_from_Histogram_test4_minpix_200.jpg

(3) fit_poly(): Calculate polynomial.

```
#(2)Search lines from Polynomial fit values from the previous frame
#def fit_poly(img_shape, leftx, lefty, rightx, righty):
def fit_poly(img_shape, leftx, lefty, rightx, righty):
    ### TO-DO: Fit a second order polynomial to each with np.polyfit() ###
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty = np.linspace(0, img_shape[0]-1, img_shape[0])
    ### TO-DO: Calc both polynomials using ploty, left_fit and right_fit ###
    left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
    right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

    return left_fitx, right_fitx, ploty
```

(4) search_around_poly (): Search lines from Polynomial fit values from the previous frame.

```
def search_around_poly(binary_warped, xm_per_pix, ym_per_pix):
    # HYPERPARAMETER
    # Choose the width of the margin around the previous polynomial to search
    margin = 100

    # Grab activated pixels
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
```

```
    ### TO-DO: Set the area of search based on activated x-values ###
    ### within the +/- margin of our polynomial function ###
    ### Hint: consider the window areas for the similarly named variables ###
    left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy +
                    left_fit[2] - margin)) & (nonzerox < (left_fit[0]*(nonzeroy**2) +
                    left_fit[1]*nonzeroy + left_fit[2] + margin)))
    right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzeroy +
                    right_fit[2] - margin)) & (nonzerox < (right_fit[0]*(nonzeroy**2) +
                    right_fit[1]*nonzeroy + right_fit[2] + margin)))

    # Again, extract left and right line pixel positions
    leftx = nonzerox[left_lane_inds]
    lefty = nonzeroy[left_lane_inds]
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]

    # Fit new polynomials
    left_fitx, right_fitx, ploty = fit_poly(binary_warped.shape, leftx, lefty, rightx, righty)

    #Calculate left_fit, right_fit for Curvature measuring
    left_fit_pix = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
    right_fit_pix = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)

    left_fitx_pix = left_fitx*xm_per_pix
    right_fitx_pix = right_fitx*xm_per_pix
```

```
    ## Visualization ##
    # Create an image to draw on and an image to show the selection window
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    window_img = np.zeros_like(out_img)
    # Color in left and right line pixels
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

    # Generate a polygon to illustrate the search window area
    # And recast the x and y points into usable format for cv2.fillPoly()
    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))])
    left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                              ploty])))])
    left_line_pts = np.hstack((left_line_window1, left_line_window2))
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty]))])
    right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margin,
                              ploty])))])
    right_line_pts = np.hstack((right_line_window1, right_line_window2))
```

```
    if Mode == 1:
        # Draw the lane onto the warped blank image
        cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
        cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
        result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

        # Plot the polynomial lines onto the image
        #plt.plot(left_fitx, ploty, color='yellow')
        #plt.plot(right_fitx, ploty, color='yellow')
        ## End visualization steps ##

    elif Mode == 2:
        # Draw the lane onto the warped blank image
        final_line_window1 = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
        final_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
        final_line_pts = np.hstack((final_line_window1, final_line_window2))
        cv2.fillPoly(window_img, np.int_([final_line_pts]), (0,255, 0))
        result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

    #return result
    return result, left_fit_pix, right_fit_pix, left_fitx_pix, right_fitx_pix # Add values to calculate curvature
```

In fifth step, finally get the result of lane finding in bird view.

```
# Define conversions in x and y from pixels space to meters
ym_per_pix = 30/720 # meters per pixel in y dimension #Define outside
xm_per_pix = 3.7/700 # meters per pixel in x dimension #Define outside

# Run image through the pipeline
# Note that in your project, you'll also want to feed in the previous fits
#result = search_around_poly(binary_warped)
result, left_fit_pix, right_fit_pix, left_fitx_pix, right_fitx_pix = search_around_poly(binary_warped, xm_per_pix, ym_per_pix)
cv2.imwrite('output_images/Search_from_Prior_'+file_name+'.jpg',result)
```

Data folder is “¥output_images¥5_Finding¥Prior”.

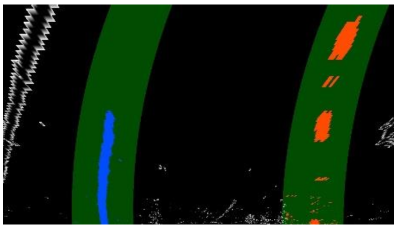


Image after Polynomial fit

Search_from_Prior_test1_minpix_200.jpg

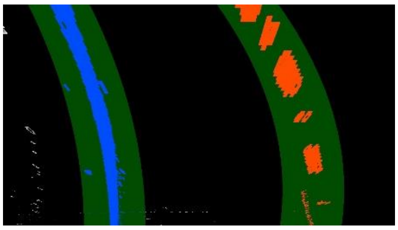


Image after Polynomial fit

Search_from_Prior_test2_minpix_200.jpg

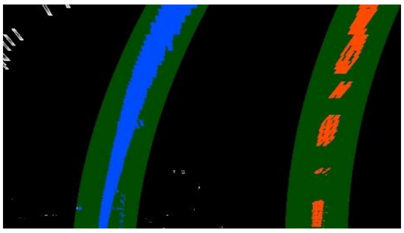


Image after Polynomial fit

Search_from_Prior_test3_minpix_200.jpg

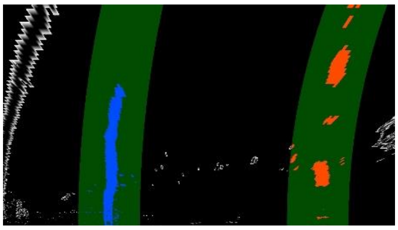


Image after Polynomial fit

Search_from_Prior_test4_minpix_200.jpg

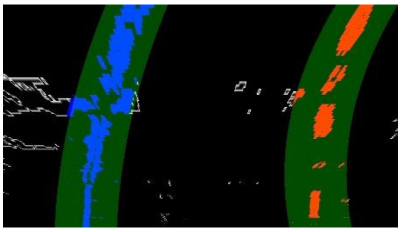


Image after Polynomial fit

Search_from_Prior_test5_minpix_200.jpg

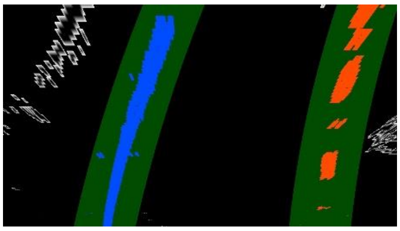


Image after Polynomial fit

Search_from_Prior_test6_minpix_200.jpg

Sixth, I measured curvature and vehicle position.

```
#6. Measuring curvature and vehicle position
#Determine the curvature of the lane and vehicle position with respect to center.

#(1)Measuring curvature

def measure_curvature_real():
    '''
    Calculates the curvature of polynomial functions in meters.
    '''
    # Define conversions in x and y from pixels space to meters
    #ym_per_pix = 30/720 # meters per pixel in y dimension # Define outside
    #xm_per_pix = 3.7/700 # meters per pixel in x dimension # Define outside

    # Use Meter-based left_fit, right_fit
    left_fit_cr = left_fit_pix
    right_fit_cr = right_fit_pix

    # 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)
    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
    y_eval = np.max(ploty)* ym_per_pix
```

```
    # Calculation of R_curve (radius of curvature)
    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])

    #print(y_eval)
    #print(left_fit_cr[0],left_fit_cr[1],left_fit_cr[2])
    #print(right_fit_cr[0],right_fit_cr[1],right_fit_cr[2])

    return left_curverad, right_curverad


# Calculate the radius of curvature in meters for both lane lines
left_curverad, right_curverad = measure_curvature_real()
curverad = (left_curverad + right_curverad)/2.0 # Resulting curvature is average of left & right

#print(left_curverad, right_curverad, curverad)


#(2)Measuring vehicle position
center = result.shape[1]/2.0*xm_per_pix
position = (left_fitx_pix[-1] + right_fitx_pix[-1])/2.0 - center
```

Seventh, I warped the found lane lines (green line area) from bird view to the original images.

I also put texts of curvature and vehicle position.

```
#7. Warp back onto original
# Warp the detected lane boundaries back onto the original image.

#(1)Unwarp function (refered warp function)
def unwarp(img):
    # Define calibration box in source and destination
    img_size = (img.shape[1], img.shape[0])

    # Get src from undistorted images
    src1 = np.float32([[708,462],[1053,688],[246,688],[577,462]]) # undist_straight_lines1
    src2 = np.float32([[678,441],[1040,675],[280,675],[607,441]]) # undist_straight_lines2
    src = (src1 + src2)/2 # Use average to get more robust src
    offsetx = 300
    offsety = 0
    dst = np.float32([[img_size[0]-offsetx,offsety],[img_size[0]-offsetx,img_size[1]-offsety],[offsetx,img_size[1]],[offsetx,offsety]])

    # Compute the perspective transform, M
    M = cv2.getPerspectiveTransform(dst, src)# Switch src and dst to calculate unwarp

    # Create warped image
    unwarped = cv2.warpPerspective(img, M, img_size, flags = cv2.INTER_LINEAR)

    return unwarped
```

```python
#(2)Unwarp lane finding image and plot onto the original image
unwarped = unwarp(result)
#final_result = unwarped
final_result = cv2.addWeighted(original_img, 1.0, unwarped, 0.3, 0)

#(3)Show the result
text1 = "Curvature[m]: " + str(round(curverad, 1))
if position > 0:
    text2 = "Position to Left[m]: " + str(round(abs(position), 3))
else:
    text2 = "Position to Right[m]: " + str(round(abs(position), 3))
cv2.putText(final_result, text1, (350, 100), cv2.FONT_HERSHEY_SIMPLEX, 2, (255, 255, 255), 5, cv2.LINE_AA)
cv2.putText(final_result, text2, (350, 200), cv2.FONT_HERSHEY_SIMPLEX, 2, (255, 255, 255), 5, cv2.LINE_AA)

plt.imshow(final_result)
plt.title('Final result')
cv2.imwrite('output_images/Final_result_'+file_name+'.jpg',cv2.cvtColor(final_result,cv2.COLOR_BGR2RGB))
```

Data folder is “¥ output_images¥6_Result”.

Image after unwarp

Final_result_test1.jpg

Image after unwarp

Final_result_test2.jpg

Image after unwarp

Final_result_test3.jpg

Image after unwarp

Final_result_test4.jpg

Image after unwarp

Final_result_test5.jpg

Image after unwarp

Final_result_test6.jpg

Finally eighth, I ran the video file “project_video.mp4”.

For the video, I changed from “Mode=1” to “Mode=2”, so only the following part in function “fit_poly()” was changed.

This means the green line area was changed from “on the lanes” to “between the lanes”.

```python
    if Mode == 1:
        # Draw the lane onto the warped blank image
        cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
        cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
        result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

        # Plot the polynomial lines onto the image
        #plt.plot(left_fitx, ploty, color='yellow')
        #plt.plot(right_fitx, ploty, color='yellow')
        ## End visualization steps ##

    elif Mode == 2:
        # Draw the lane onto the warped blank image
        final_line_window1 = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
        final_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
        final_line_pts = np.hstack((final_line_window1, final_line_window2))
        cv2.fillPoly(window_img, np.int_([final_line_pts]), (0,255, 0))
        result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)
```

(2) Result of my pipeline

My pipeline mostly worked well with the vide file“project_video.mp4”.

The folder is “test_videos_output” and the video file name is “**project_video_output.mp4**”.

(3) Identification of potential shortcomings with my current pipeline, and suggestion of possible improvements

In the output video, there was just one moment when the left lane detected the edge of the wall as below **(A)**.

There may be the following countermeasures:

(i) Right lane looks match well **(B)**, and the width of lane can be assumed by High Definition Map Information **(C)**. In this case, we may be able to eliminate the left line candidate points that are far from the right line.

(ii) The left line mismatch occurred for just one moment, so we may be able to eliminate the sudden change of the lane line detection by smoothing with the former detected line data. The former data doesn’t have to be only one step before, but can include some steps before to make it more robust.

project_video_output.mp4

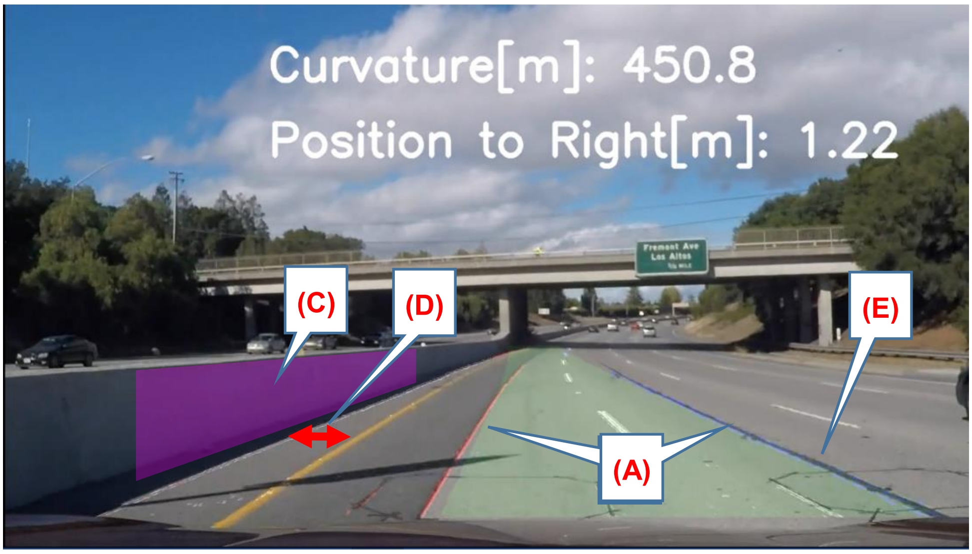
I also ran my pipeline with “challenge_video.mp4” and “harder_challenge_video.mp4”.
I found other 3 potential shortcomings with my current pipeline as bellow:

(i) Robustness against bad road condition

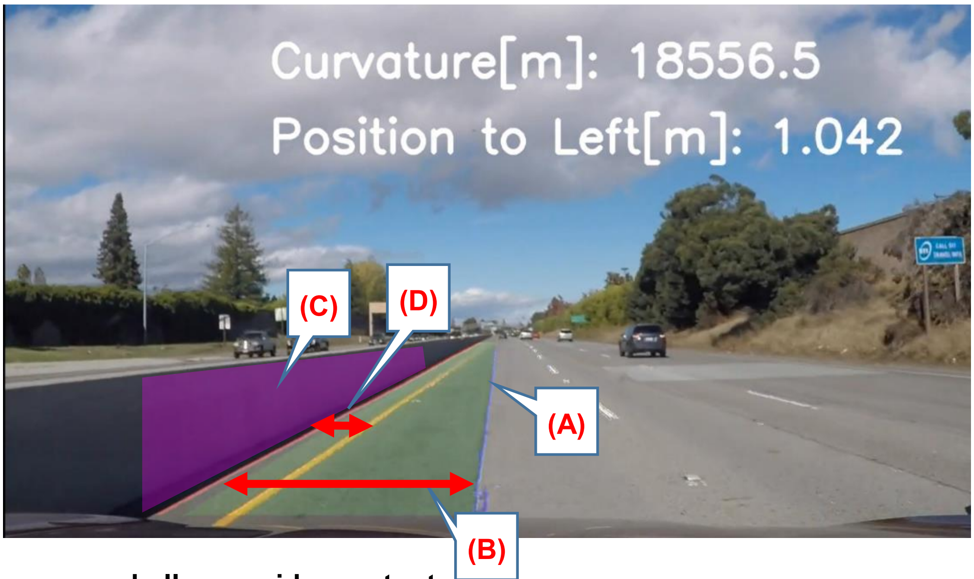
If there are clack lines on the road, my pipeline mismatched them **(A)**.
One of countermeasures is the same as previous page, which is assuming the width of lane by map information if the lane width is different from it **(B)**.

However, if the clack lines are deep there’s no idea which lane is real lane.
We may be able to assume the left lane by Sensor Fusion. We can detect the position of wall by Radar or LiDAR (C), and we know how much distance is there between wall and left lane by High Definition Map Information **(D)**, so we can set the higher weight around the real left lane during lane detection algorithm.

We also may be able to assume the right lane by detecting another vehicle on the adjacent lane.
If there’s another vehicle in the right side, we can set lower priority on the right side clack line area **(E)** to detect the real lane.



challenge_video_output.mp4



challenge_video_output.mp4

(ii) Robustness against backlight

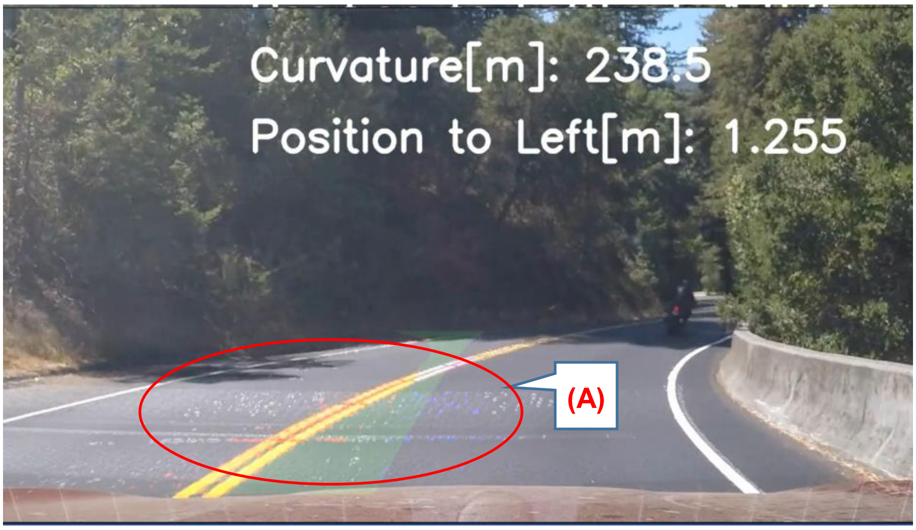
Red dots are left side and **blue dots are right side**.

When there's backlight and camera captured dots of light, the lane detection became unstable **(A)**.

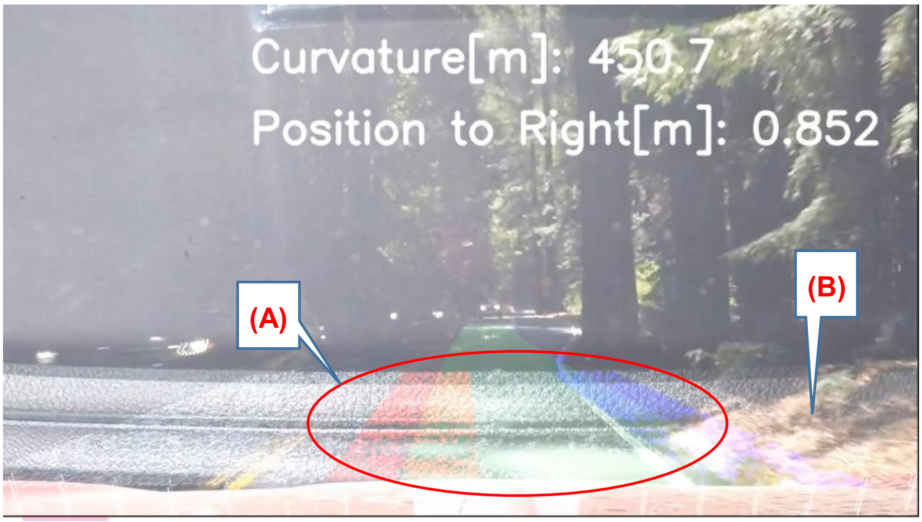
There are some area without wall **(B)** and this is one side lane, so the countermeasures in the previous page don't work.

If the time of backlight is short, we may be able to decrease the sudden change of the lane line detection by smoothing with the former detected line data, which was written in 2 pages before.

If the time of backlight is long, it's better to hand over the vehicle control to human driver safely, then stop the autonomous driving system.



harder_challenge_video_output.mp4



harder_challenge_video_output.mp4

(iii) Robustness against steep curve

In case of steep curve, the lane detection doesn't match the real lane at all.

My pipeline covers only straight roads and gradual curves **(A)**.

However, on steep curves the coverage area should be changed like **(B)** and I should not detect dots in other area.

We can detect whether it's steep curve or not by the following information:

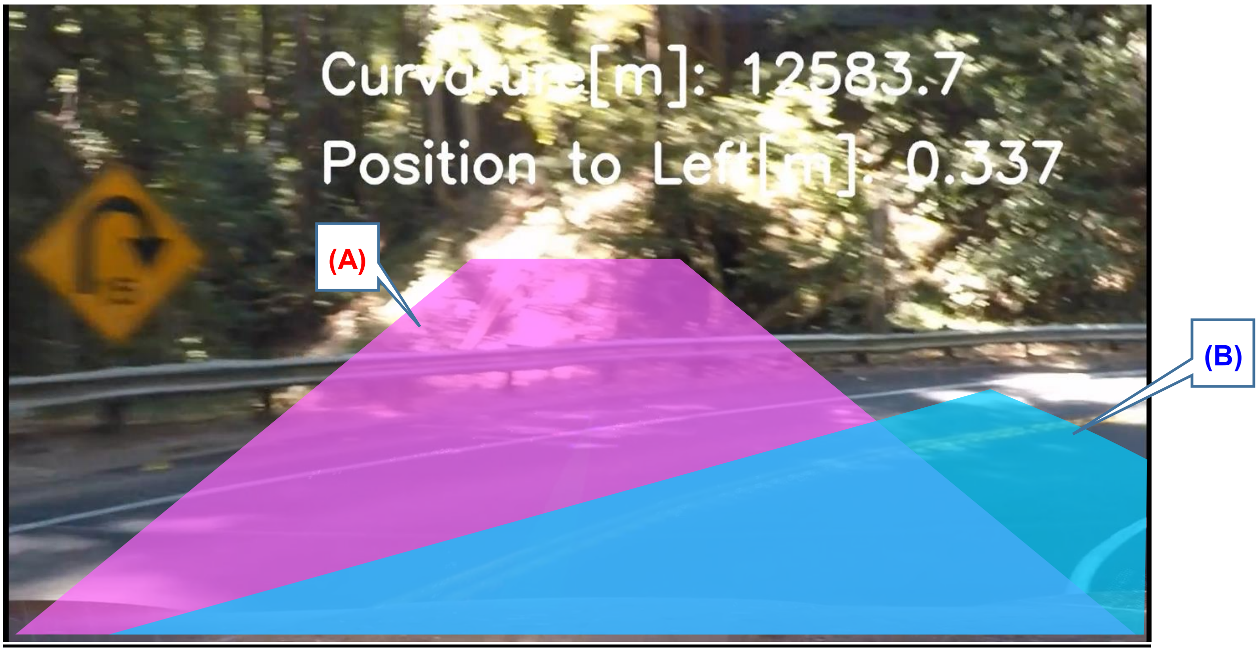
(i) High Definition Map Information

(ii) Current angle of steering wheel

(iii) Current lateral acceleration

And also, decreasing vehicle speed helps self-driving in this scene by the following viewpoints:

- The curvature changes slowly, so lane detection by camera image will be more robust.
- Even if lane detection mismatches sometimes, the vehicle behavior will be slowly so it's easier to recover to the stable status.



harder_challenge_video_output.mp4