Advanced Lane Finding Project

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients on a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("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.
- Other points in Rubric

Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is in **CameraCalibExtra.py** file **(line 10-73)**.

Note that I was trying to use as much as image possible for a better calibration (not much but was good to try). In this case calibration1.jpg (9x5) and calibration5.jpg (7x6) are not 9x6 full view but other grid size is workable.

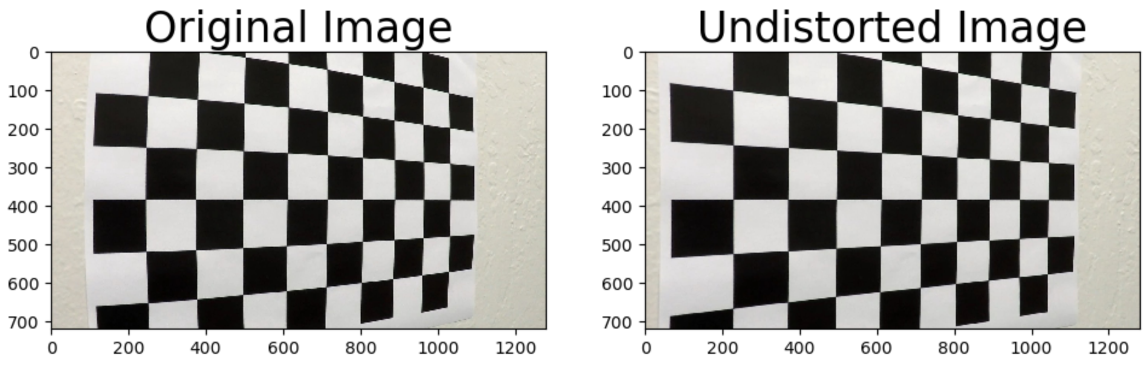
To extract the 'corner' on the image coordinate using Opencv, I used
ret, corners = cv2.findChessboardCorners(gray, (nx,ny), None)

The wold coordinate or "object points" corresponding to these found corners will be the (x, y, z) coordinates of the chessboard corners. I am assuming the chessboard is fixed on the (x, y) plane at z=0 such that the object points are on the same global frame for each calibration image.

The successful detected corners, and corresponding world coordinate are appended to 'imgpoints' and 'objpoints' respectively for a calibration.

I then used the output 'objpoints' and 'imgpoints' to compute the camera calibration and distortion coefficients using the '**cv2.calibrateCamera()**' function.

I applied this distortion correction to the test image using the '**cv2.undistort()**' function and obtained this result.



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

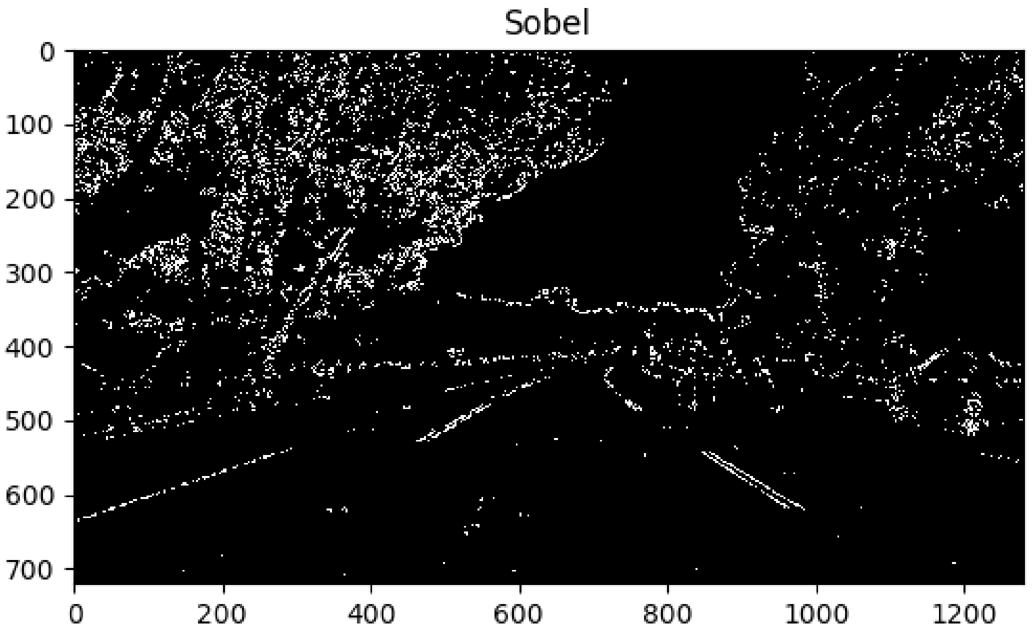
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I used a combination of color (hls, rgb) and gradient thresholds to generate a binary image.

```
 # [kernel, min thresh, max thresh]
gradx  = [3,20,150]
grady  = [3,20,150]
mag    = [5,40,150]
direct = [15,0.8,1.2]
```

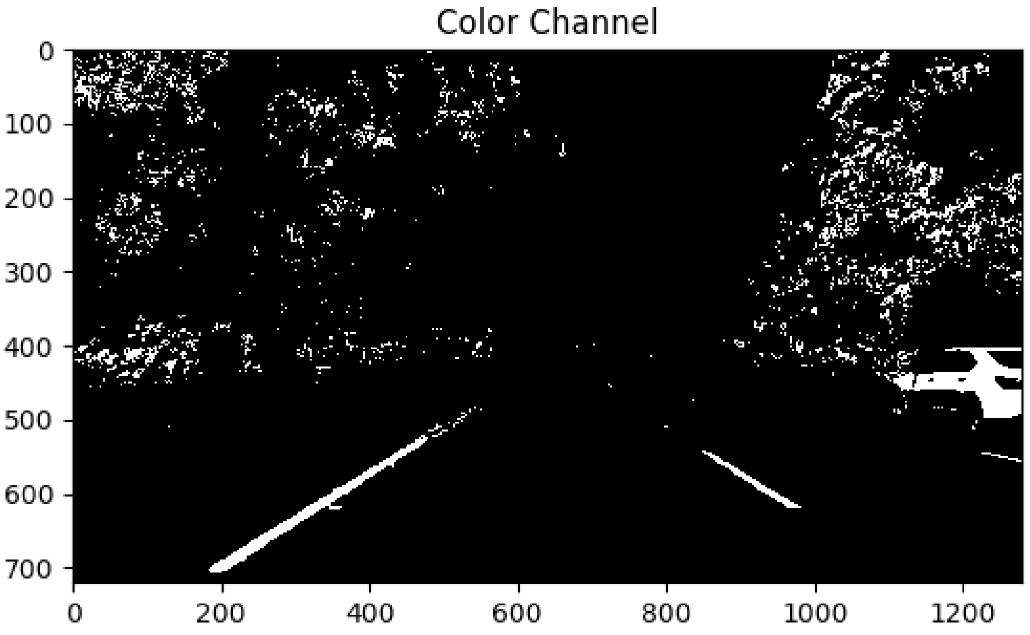
```
# Color space thresholding
hls_thresh = [(15,80),(120,230),(120,230)]
rgb_thresh = [(220,255),(None,None),(None,None)]
```



Function **ApplySobelThreshold()** in **AdvanceLane_PiplineVideo.py** (line 147-159)

Thresholding with the following binary mask:

```
 bin_output[((gradx == 1) & (grady == 1)) | ((mag_binary == 1) & (dir_binary == 1))] = 1
```

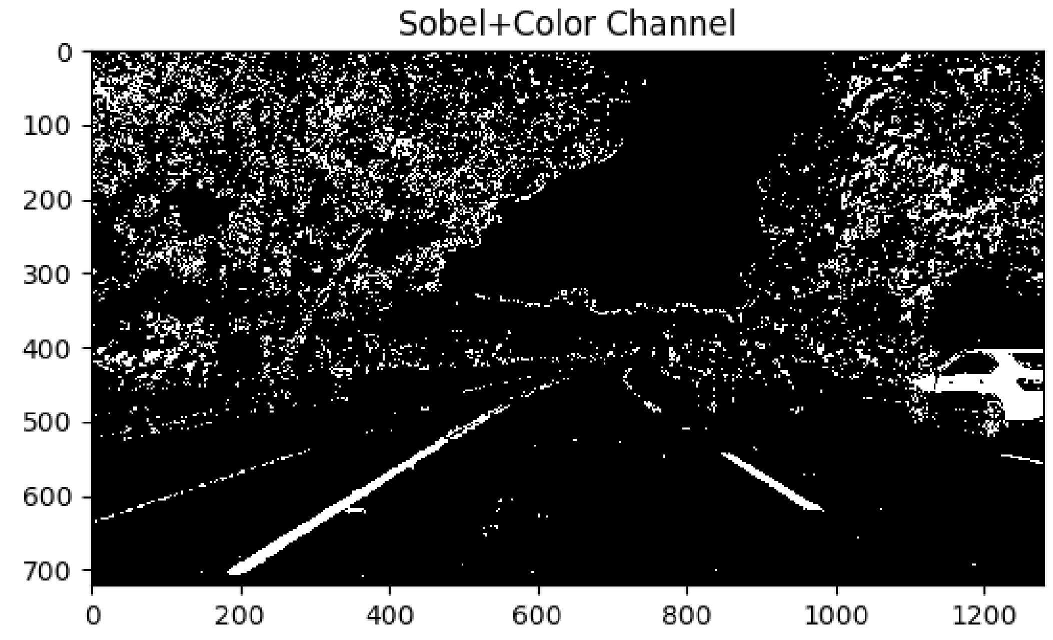


Function **ColorChannelThreshold()** in **AdvanceLane_PiplineVideo.py** (line 162-191)

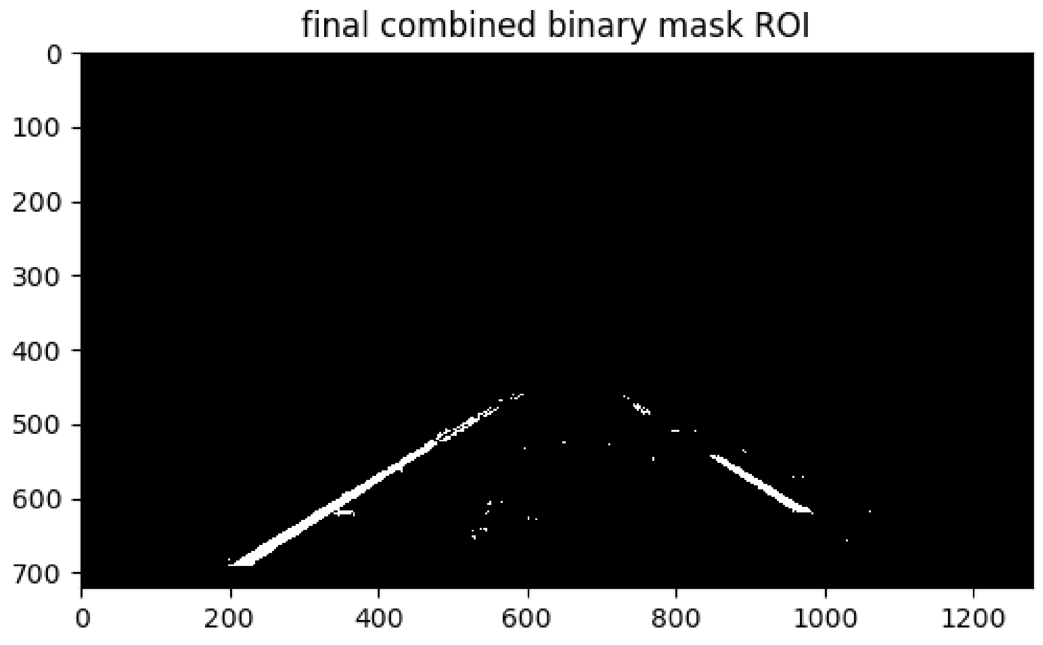
Thresholding with the following binary mask:

```
bin_output[ ((bin_h == 1) & (bin_s == 1))  | (bin_r ==1) ] = 1
```

Here's an example of my output for this step. (note: this is not actually from one of the test images)

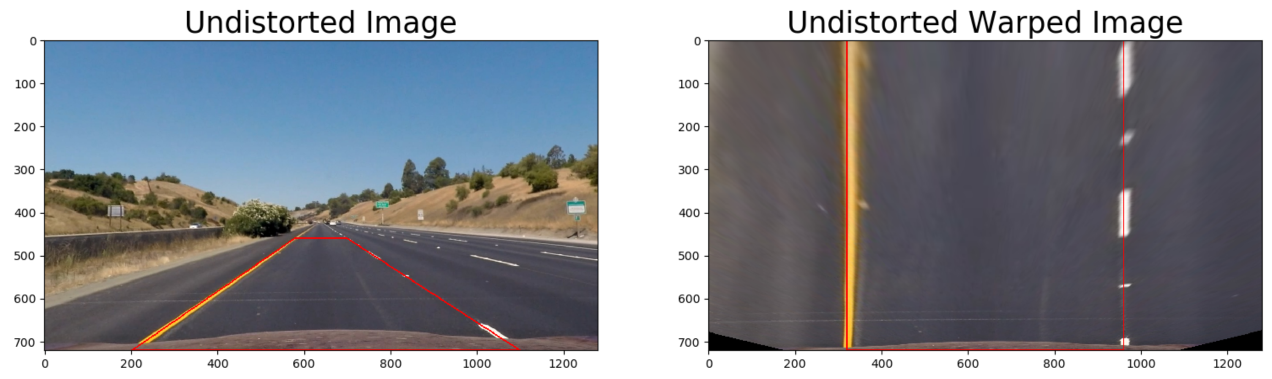


Combine Sobel with Color channel (line 657 in **AdvanceLane_PiplineVideo.py**):
bin_output[((bin_sobel == 1) | (bin_color == 1))] = 1



Select ROI: function **region_of_interest()** in **AdvanceLane_PiplineVideo.py** (line 193)

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.



The code for my perspective transform includes a function called ‘ComputeTransform()’, which appears in lines 78 through 91 in the file ‘**CameraCalibExtra.py**’.

The **‘ComputeTransform()’** function takes as inputs an image (‘img’), as well as source (‘src’) and destination (‘dst’) points, camera matrix (‘mtx’) and distortion parameters (‘dst’).

I chose to hard code the source and destination points by mapping the points from the source image to the desire destination points in the following manner:

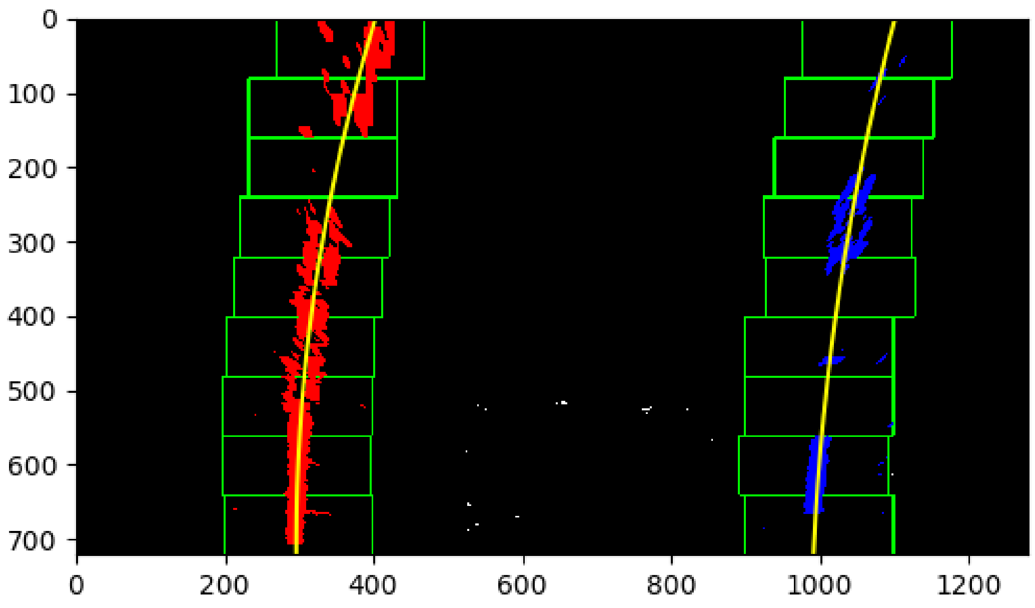
Source points (col, row pixel)	**Destination points (col, row pixel)**
203, 720	320, 720
580, 460	320, 0
700, 460	960, 0
1100, 720	960, 720

I verified that my perspective transform was working as expected by drawing the ‘src’ and ‘dst’ points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

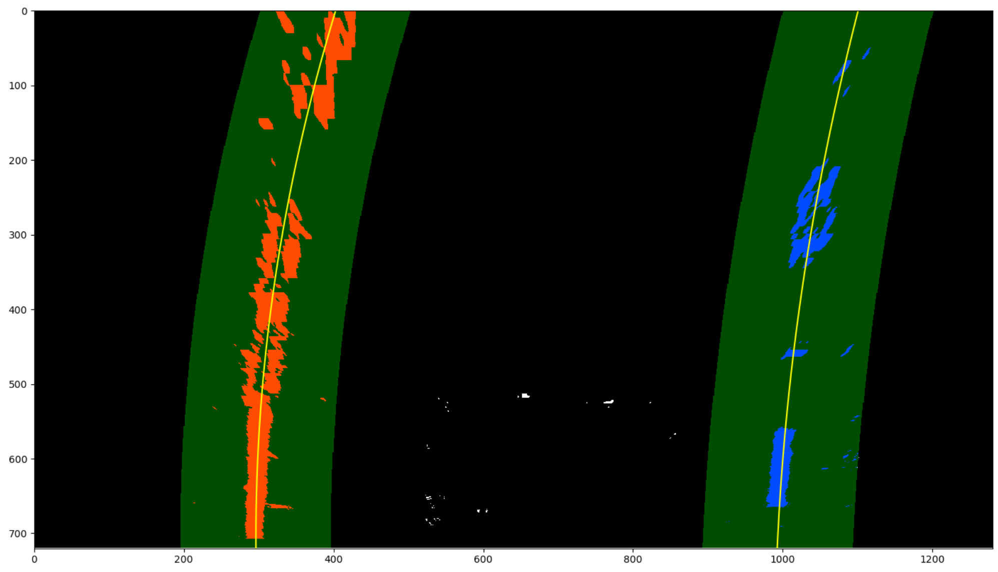
Pixel to Meter conversion: Work out from the warped straight line image
y = line length 487 to 553 = 66 pixels for 3.0 m   => 0.0455 m/pixel
x = line gap 327 to 970   = 643 pixels for 3.7 m => 0.0060 m/pixel

4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

On start up, I use the histogram peak approach to search the line pixels and fit the 2nd order polynomial line like the one below.
ExtractLinesFromInitial() in **AdvanceLane_PiplineVideo.py** (line 218-334)



On the next image frame, I don't start to search blindly. I use the previous fit line to pick the neighbor pixels around these line for the fit.
ExtractLinesReiterate() in **AdvanceLane_PiplineVideo.py** (line 337-405)



2nd Polynomial fit: let x := columns, y:= rows, $x = Ay^2 + By + C$
FitLinePolynomial() in **AdvanceLane_PiplineVideo.py** (line 408-419)

When a bad condition detected such as the lines are not parallel, the gap is not in the range it should be, the left and right curvature is very different and also the is a big jump in the average curvature from the previous loop for e.g. 2 consecutive order, we need to reset the iteration and start the search from the beginning.

There a function call **SantityCheck()** in **AdvanceLane_PiplineVideo.py** (line 549-596) to check all of these condition on every iteration. I used a simple check on the gap, if the line parallel enough, the difference in left and right curvature, and the curvature comparison on the previous loop. These values are the average over the 10- sampling intervals.

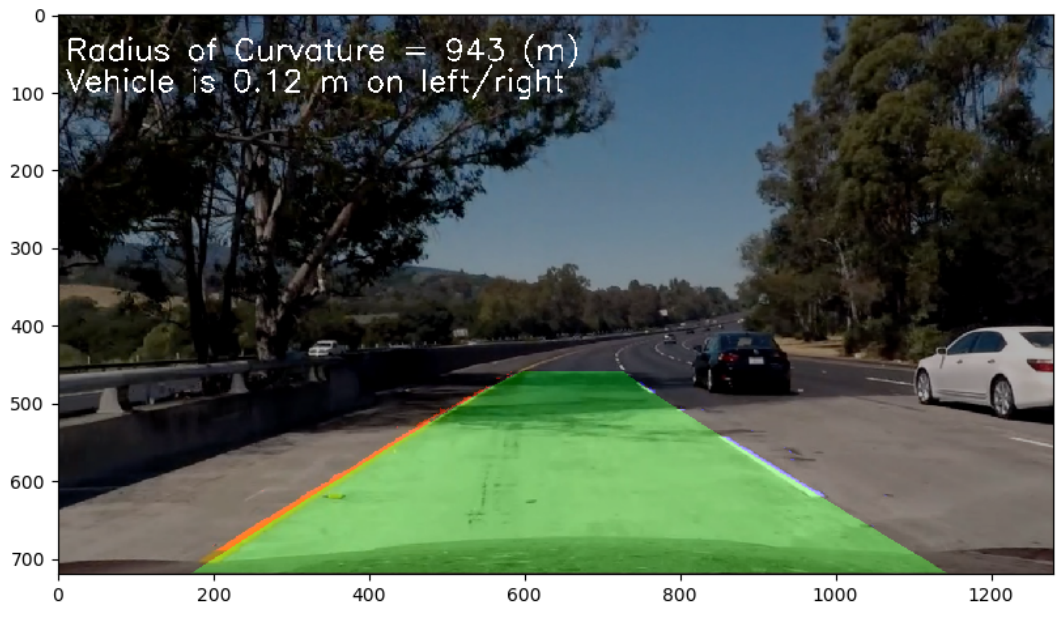
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Curvature calculation function: **ComputeLineCurvature()** in **AdvanceLane_PiplineVideo.py** (line 433-469). In brief, I used the line fit (e.g. best fit from n-iterations) to compute the radius of curvature in a meter unit (line 464).

The position of the vehicle from the center was computed using the x- base point (e.g. best x after n-iterations) of the left/right line (code line 802-811).

Position = (image width – left base point – right base point)/2
Direction: On the right if Position +, On the left if Position –

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.



Radius of Curvature = 943 (m)
Vehicle is 0.12 m on left/right
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

InverseProjectionImage() function in **AdvanceLane_PiplineVideo.py** (line 471-506).

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

See project_video_out.mp4
See challenge_video_out.mp4

Note: challenge_video_out.mp4 is a work in progress and is used as a reference for this project on the normal video. I fine tuned it to make it work as a benchmark, just to see how different it would be from the normal video (project_video.mp4)
in term of a parameter setting.

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

The challenges I face were

- Finding the good set of thresholding ranges that work on normal, bright, dark conditions.
- Finding the bit mask logic for combining binary extraction in sobel edge detection and color channel HLS or RGB in order to get the best possible result and work on these various conditions.
- Trying to find the solution to detect bad conditions in order to stabilize the curve radius and vehicle position calculation.

When this solution might fail

- When it snows, or rains. So lane lines are not very visible or the condition is changed very much (very bright/ dark so the lane lines are very not clear to see) that these fixed thresholding parameters are no longer work.
- Other lane or tire marks as well as substantial overcasting shadow as appeared on the challenge videos would be appeared as the lane lines. The might lead to the fault classification.

Further improvement

- To get it works on wider conditions, my implementation of this lane detections needs to be generalized more to avoid over fitting into one condition. For this work, I might need to reduce the number of bit mask logics for the binary image thresholding in order to make it more flexible.
- One could use adaptive or machine learning approaches to adjust the thresholding parameters and other image processing parameters according to the lighting conditions. For example, we could select the parameters base on the current image histogram. However, I am not sure how reliable it would be.
- In addition to the land detection from the binary image as used in this project, other different techniques that less subjective or sensitive to image lighting and contrast could be explored such as registration/pattern matching techniques.