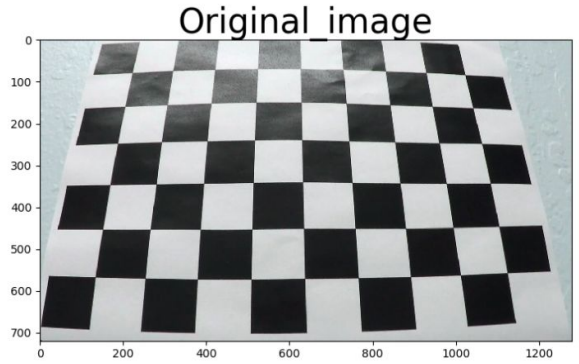
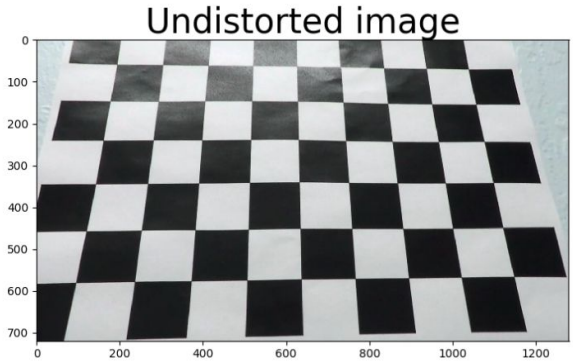
Advanced Lane Project:

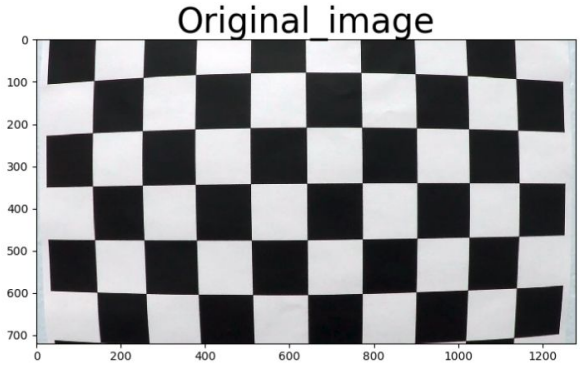
1 camera calibration

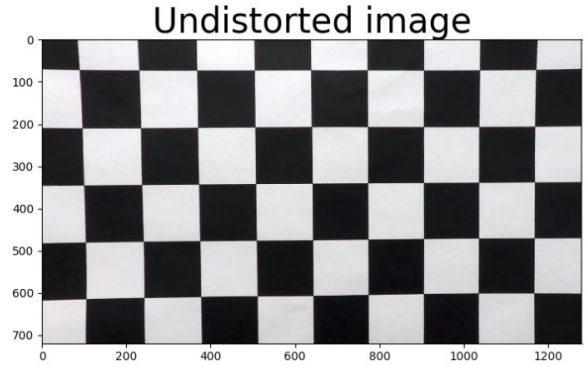
To begin with, we need to obtain distoration coefficients and camera transformation matrix which will be used to transform 3D images into 2D images.

The images for camera calibration have 9x6 corner grid, therefore we prepare an numpy array for storing each corner position in the given 9x6 grid as our object points, and then we provoke cv2.findChessboardCorners to find corresponding corner position in the test images, along with object points. Then, we are able to provoke cv2.calibrateCamera to get transformation matrix and distortion coefficients. The effectiveness of undistortion is as below:

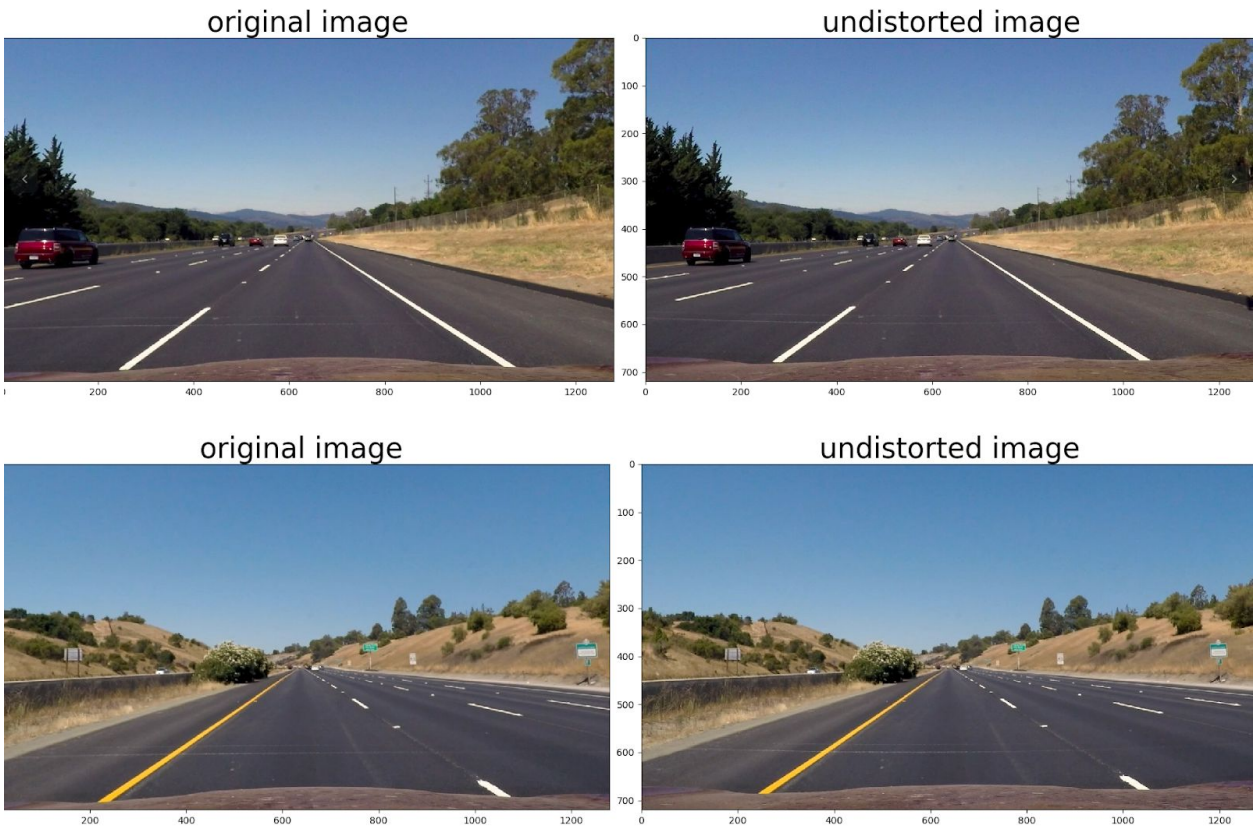








Finally, we apply to our test images:
Left side is distorted image, right side is undistorted image



After that, we store the matrix and coefficients into a pickle dictionary file on local file system so that parameters can be easily reused for our further work.

2 image process

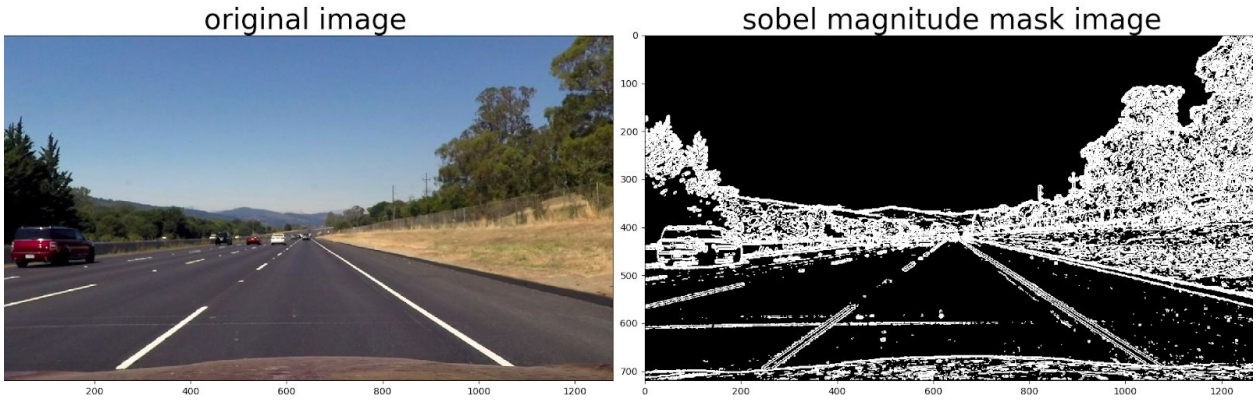
The purpose of this phase is to help us to be able to easily identify road lane visually by various ways of recognizing the area of interest.

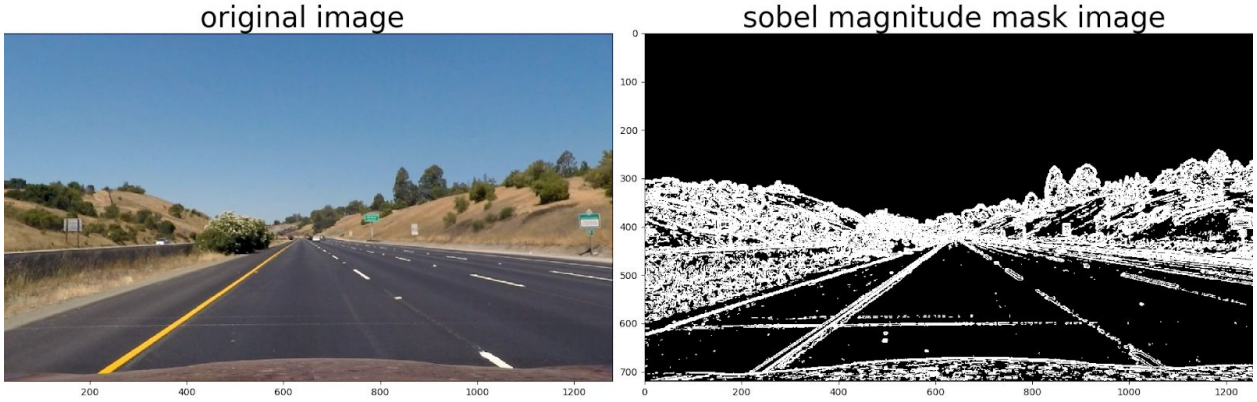
1) Sobel magnitude threshold mask

Road lane normally has a strong lane marking on left side and right side of the road, such feature can be captured by sobel kernel multiplication. Regardless of either x-axis direction or y-axis direction, we simply apply a sobel magnitude threshold to mask our input image.

We can compute magnitude of sobel by following piece of code:
sobel_magnitude = np.sqrt(np.square(sobel_x) + np.square(sobel_y))
Where sobel_x, sobel_y are sobel multiplication value on x-axis and y-axis, respectively.

As an example for demonstration, we use a min threshold 10 for masking, which means all pixel whose sobel magnitude value less than 10 will be 0, otherwise be 1.



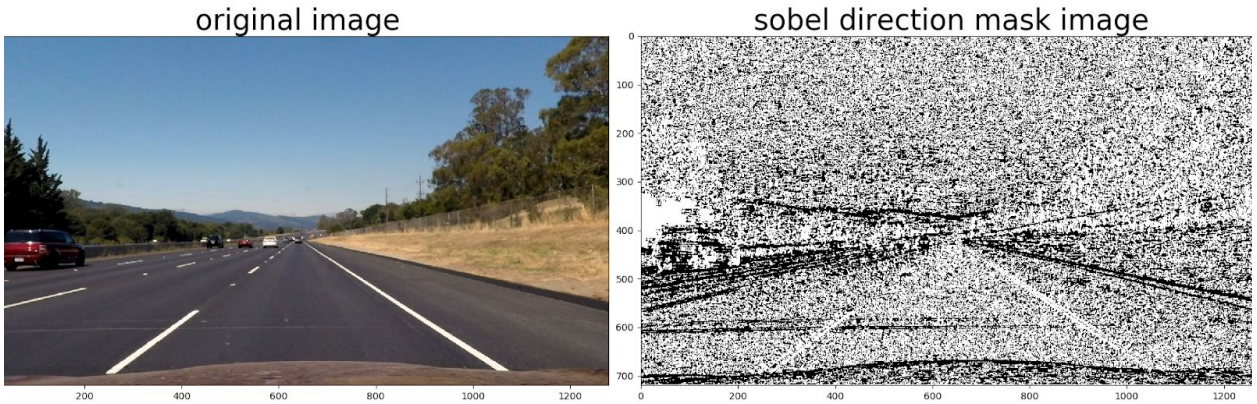


2) Sobel direction threshold mask

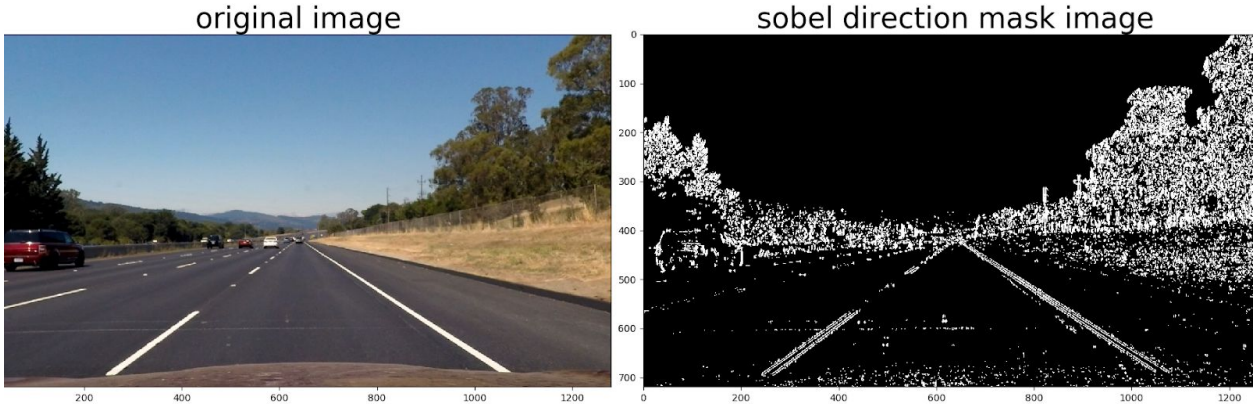
However, as you can see from above output, there are still a lot of noise on both sides of the image. We further optimize our results by masking sobel direction value which is a arctan value of sobel_y/sobel_x. To be more convenience, we convert radian into angle by following piece of code:

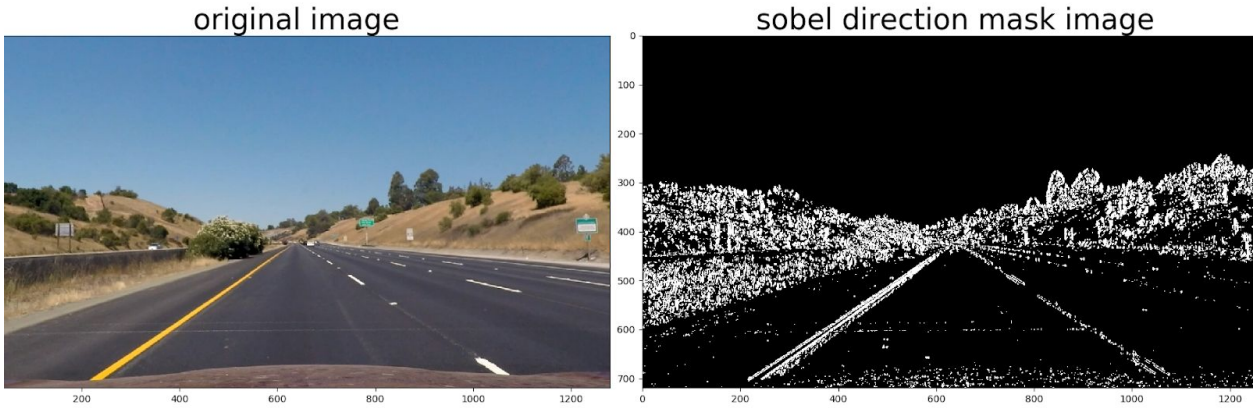
sobel_direction = np.arctan2(sobel_y, sobel_x) * 180 / np.pi

We set the range of our threshold mask be [-60, 60], and we have following outputs:



Combined with the output of sobel magnitude, we have:



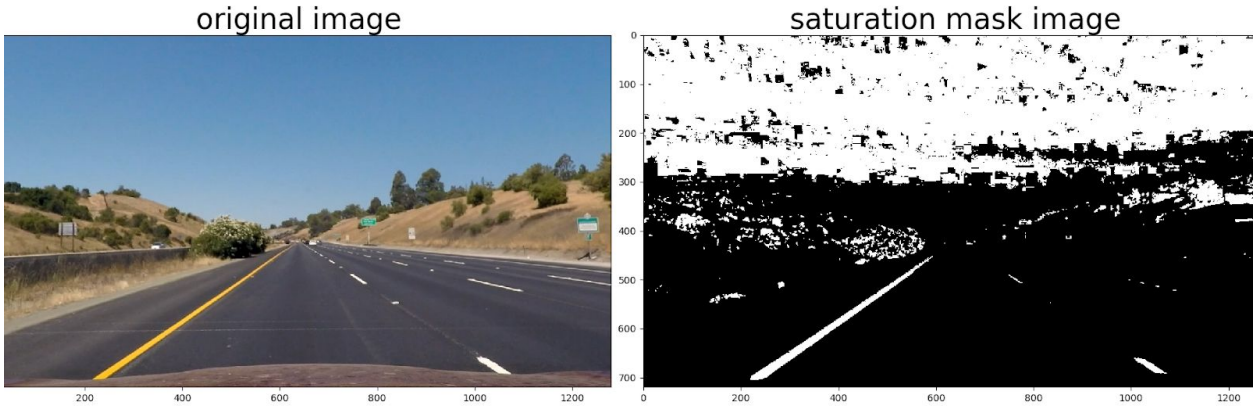


3) Color space threshold mask

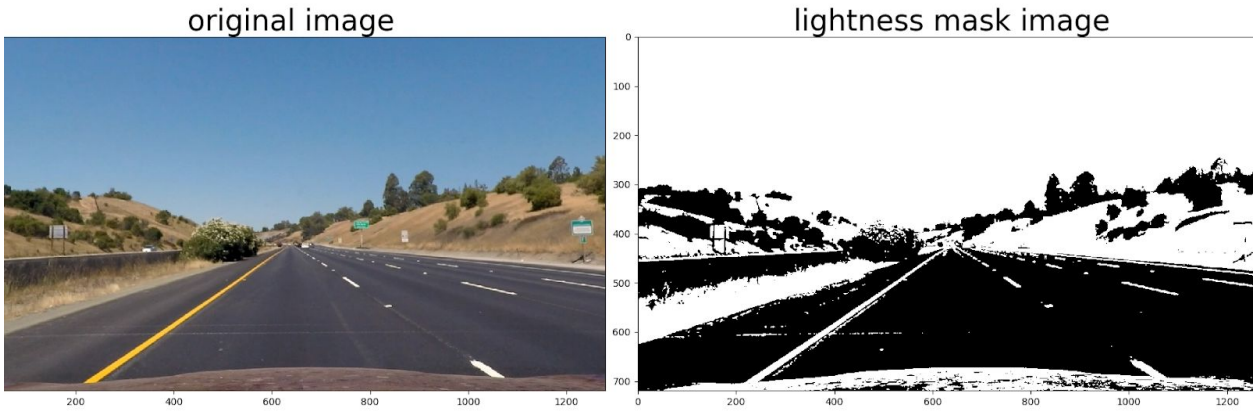
Most noises on the road plane has been eliminated and to remove noises on surrounding area, our next step would be to mask color space.

First of all, we consider the HSV color space, because HSV space is more robust than RGB space given the various quality of test images.

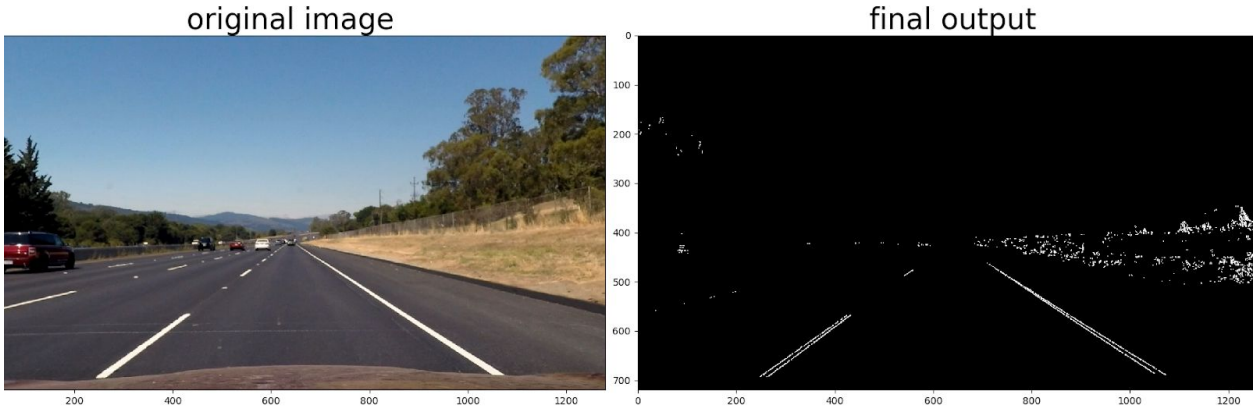
Below is the output via masking saturation value with a range of (100, 255):

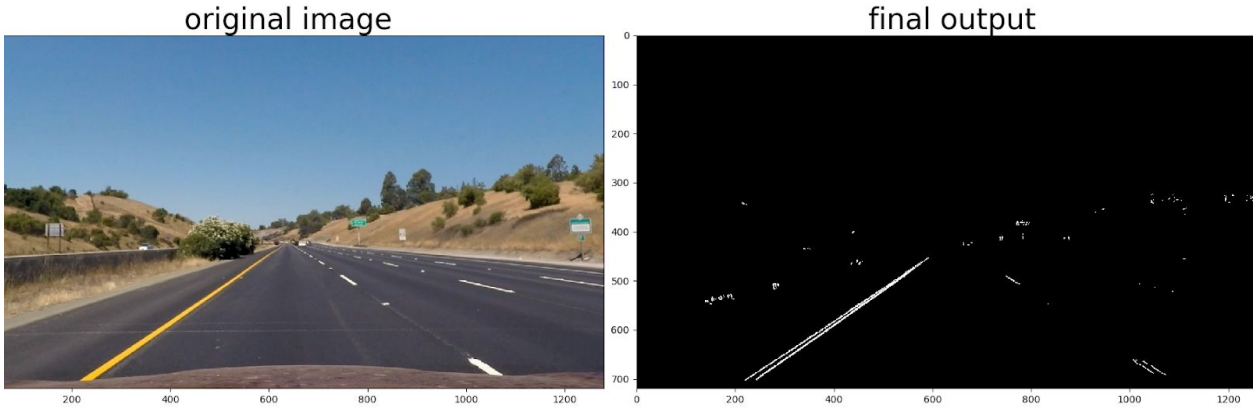


Then, we can see the output by masking the lightness value with a range of (100,255):



Both of L and S values have significant impact on changing the distribution of noisy, thus we combine them together with all previously defined threshold, we have folloowing outputs:





Finally, our final output will makes relatively easy to identify the position of lanes visually.

4) Perspective transform

So far at this phase, we are able to apply a similar technique as been used in the first project to solve most straight lines situation. However, in reality, different perspective makes the shape of lanes more like a curve instead of a straight line. Thus, to able to deal with different perspective or shapes of the lane, we need to apply a perspective transform to our image. The key of this step is to identify the ROI from our input image and project that ROI into our output image.

The trick of identifying ROI is to roughly define the four points of a Trapezoid shape from input image. Specifically, our ROI is defined as below:

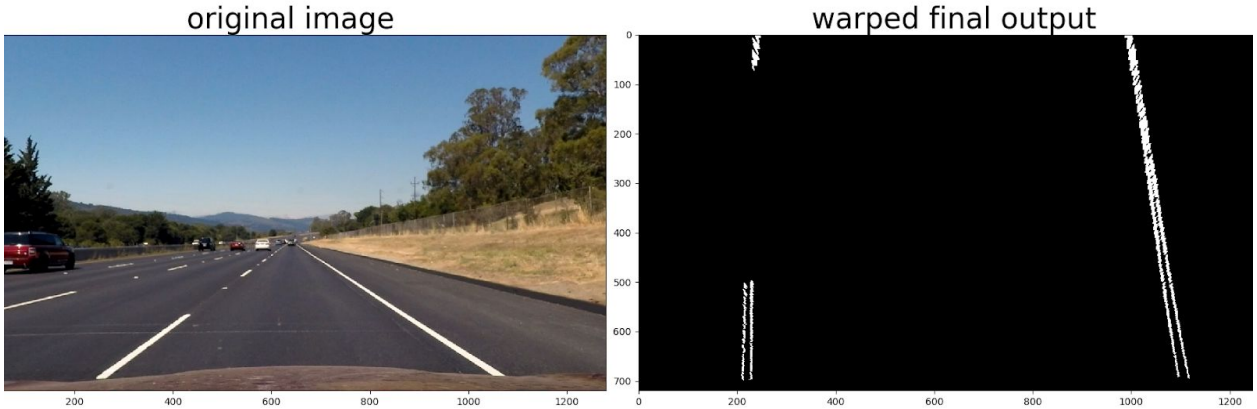
After definition of ROI, we use following code to compute transformation matrix , inverse transformation matrix and apply that matrix to get our final warped image:

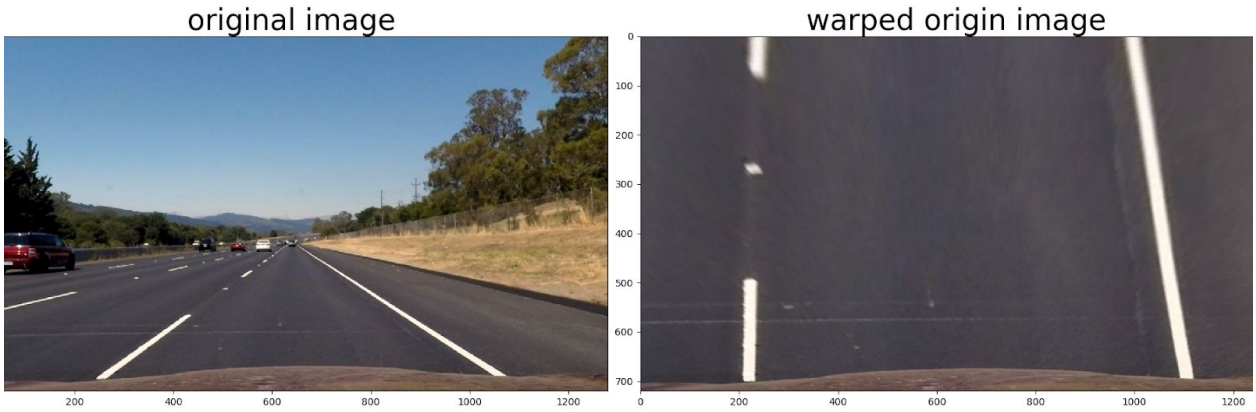
self.M = cv2.getPerspectiveTransform(rect, dst)

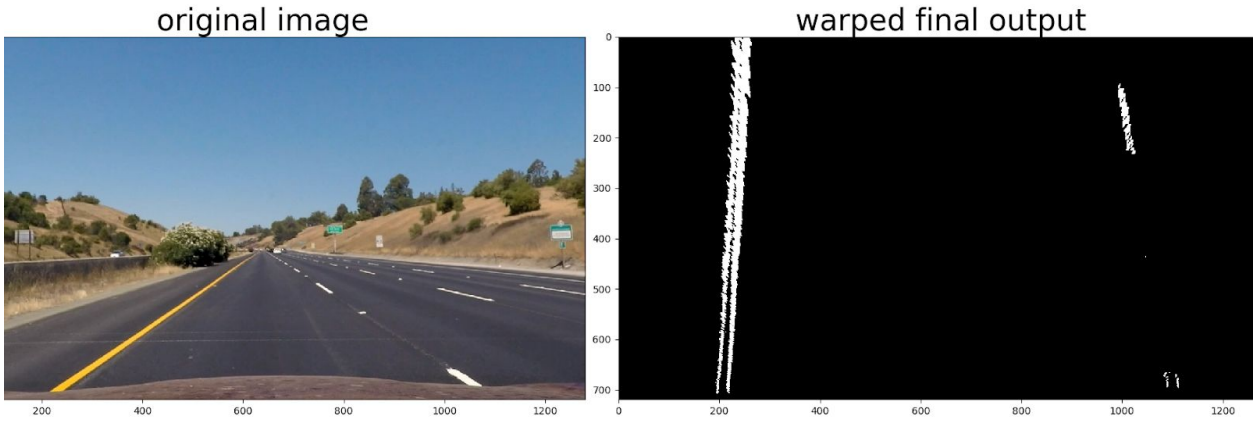
self.MINV = cv2.getPerspectiveTransform(dst, rect)

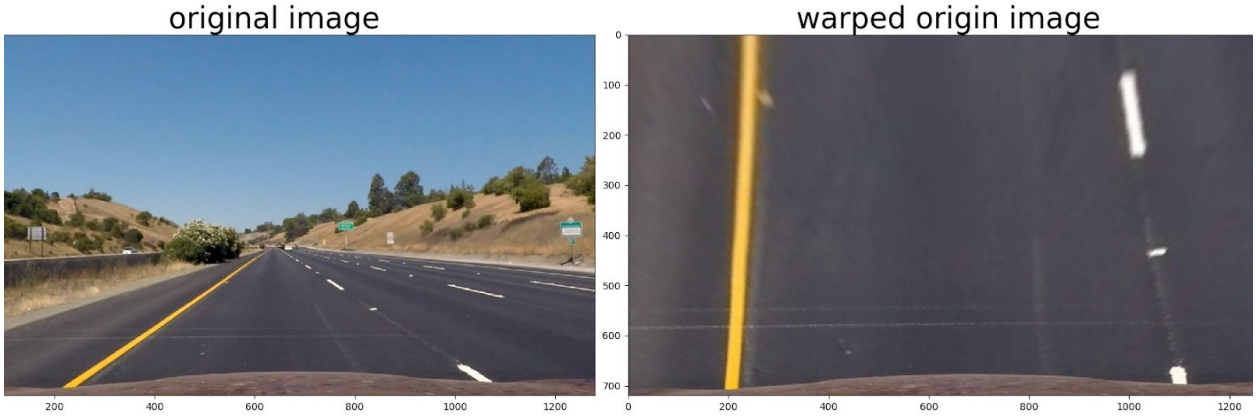
self.warped_image = cv2.warpPerspective(input_img, self.M, img_size, flags=cv2.INTER_LINEAR)

Our final output is like:







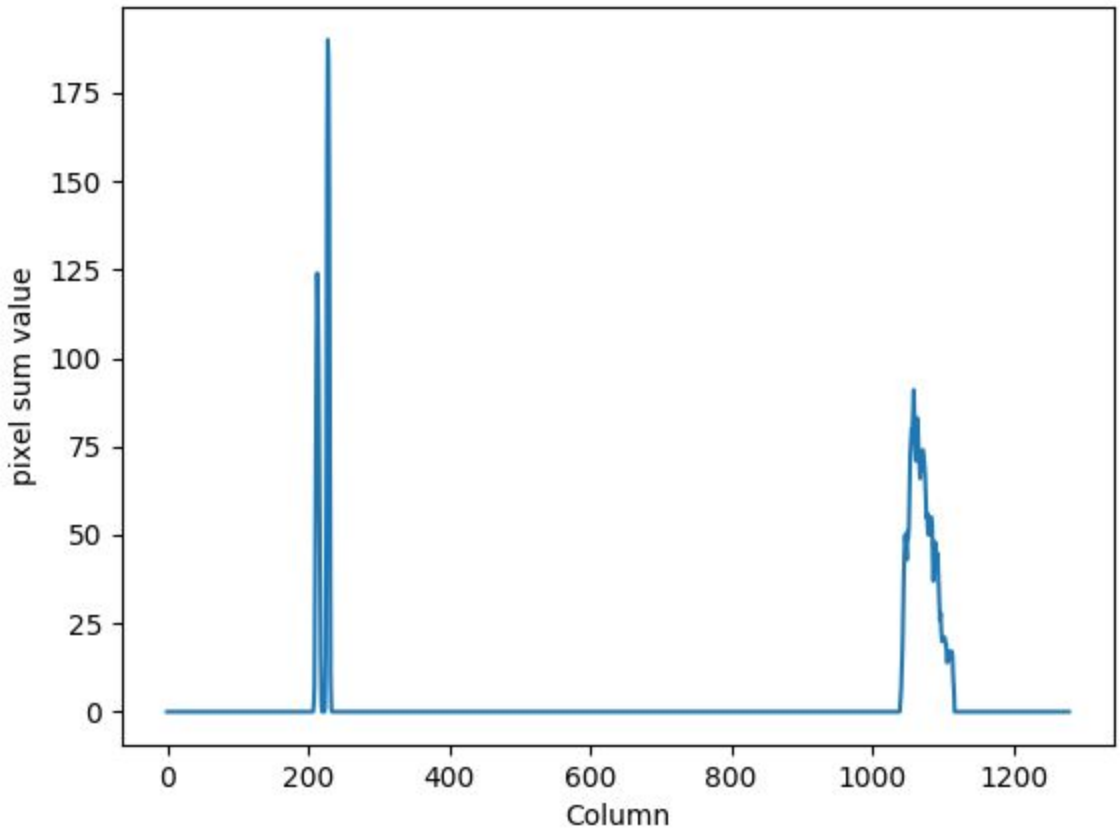


3 lane searching design

1) algorithms design

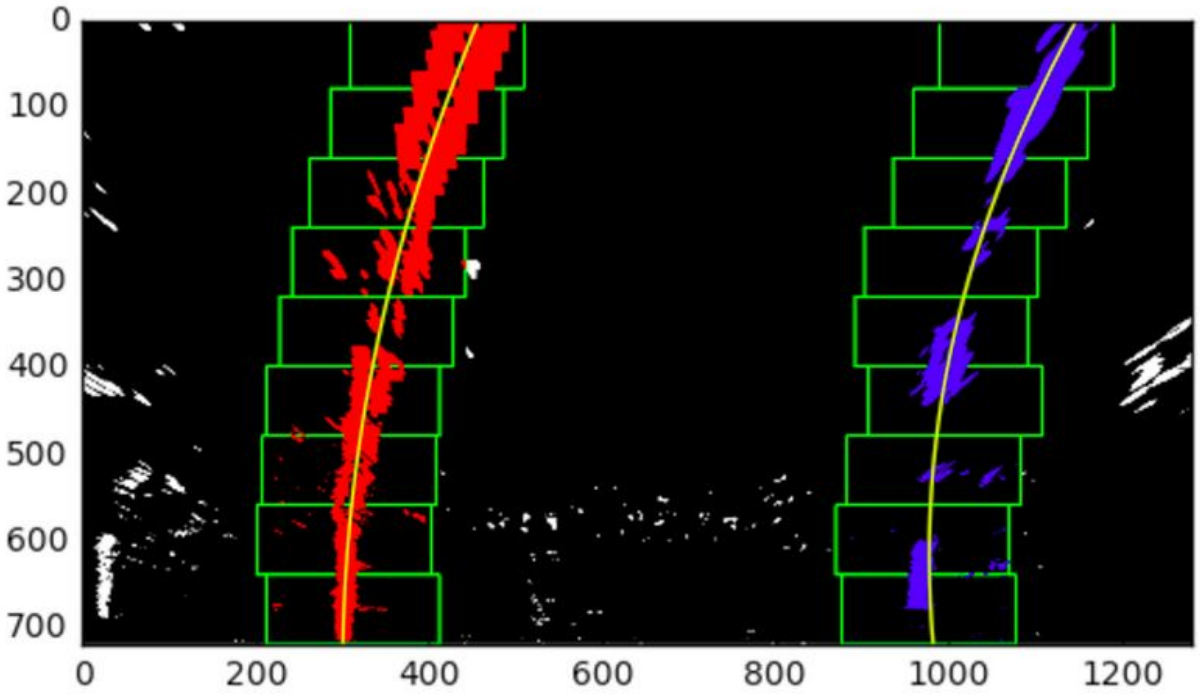
We add up the sum of pixels in each column from the middle row to the bottom row and we plot the distribution of the sum of each column to find the two peak column, one peak column should locate in the first half of the width and the other peak column should locate in the second half of the width.

As you can see from below that the first peak is around 200th column and the second column is around 1100th column which corresponding to the left lane and right lane, respectively.



Next, we define a search window which will search all pixels whose x-axis position are within a predefined ‘margin’ width and y-axis position are within the height of defined search window, with positive pixel value. Once we locate such pixels in each search windows range, we append those pixel coordinates into corresponding lane_x array for x-axis position and lane_y array for y-axis position.

The search windows is like:

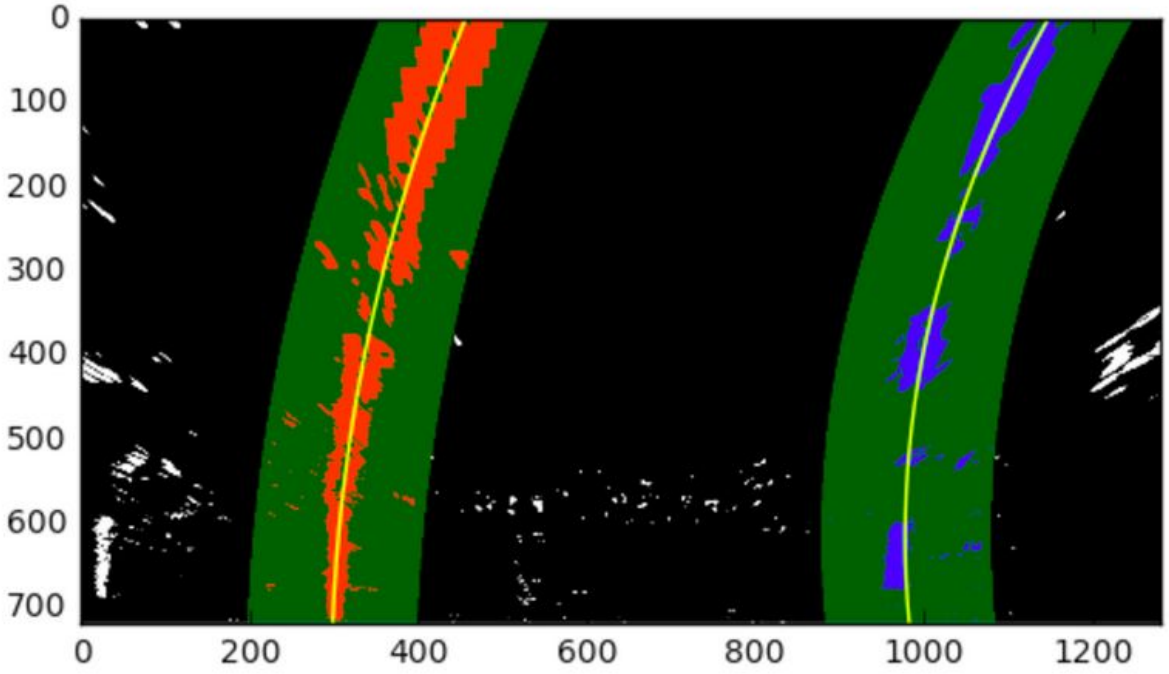


2) algorithms discussion

There are another sliding window approach, instead of sum up pixel value, performing convolution withn search window width to find the maximum column be the centroid pixel of each searching window. However, the drawback of this approach is that the centroids we have found are tend to be noise and incorrect because the centroid we found is just the column with the peak convolution results which is not necessarily relates to the correct pixel which should resides on the lane area. Besides, it would be difficult to apply the fitting technique to plot a interpolate a curve which represents the direction of the lane.

3) Lane fitting

After we have found all related pixels which approximate to the lane area, we can provoke polyfit from numpy to perform polynomial fitting to get corresponding coefficients of the polynomial to get an estimated lane curve.



4 curvature & offset computation

To compute the curvature in meters, in addition to compute left-lane polynomial fitting coeffcients and right-lane polynomial fitting coefficients, we need to convert pixel coordinates information into meters and then compute new lane polynomial fitting coefficient. We compute curvature in following code:

```
y_eval = (self.image.shape[0] - 1) * self.ym_per_pixel
        left_curv = ((1 + (2 * leftfit[0] * y_eval + leftfit[1]) ** 2) ** 1.5) / np.absolute(2 * leftfit[0])
        right_curv = ((1 + (2 * rightfit[0] * y_eval + rightfit[1]) ** 2) ** 1.5) / np.absolute(2 * rightfit[0])
        self.curvature = int((left_curv + right_curv) / 2)
```

We take the average of left curvature and right curvature as our result.

The offset meters from the center on the road is just the distance between the middle of the image and the middle of the lane(between left lane and right lane) assuming that the camera is mounted at the middle of the car and camera perspective do not change.
We can obtain offset by following code:

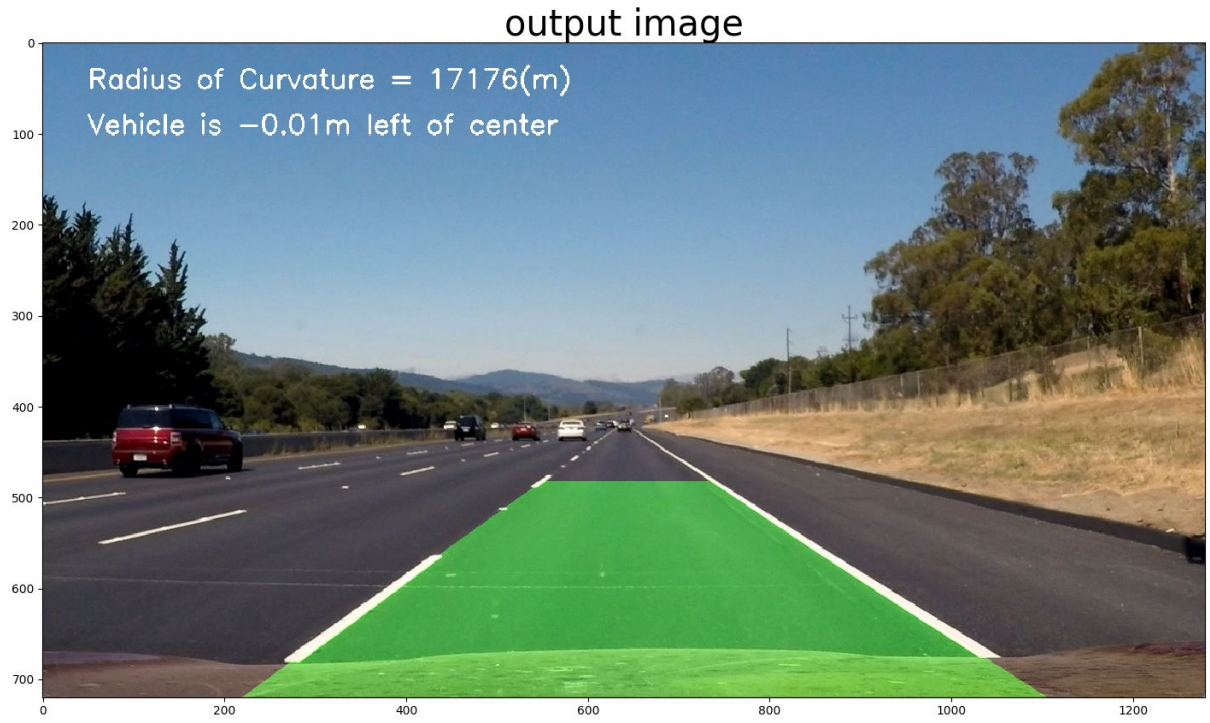
```
ploty = self.ploty * self.ym_per_pixel
left_offs = np.zeros(self.image.shape[0], np.float32)
```

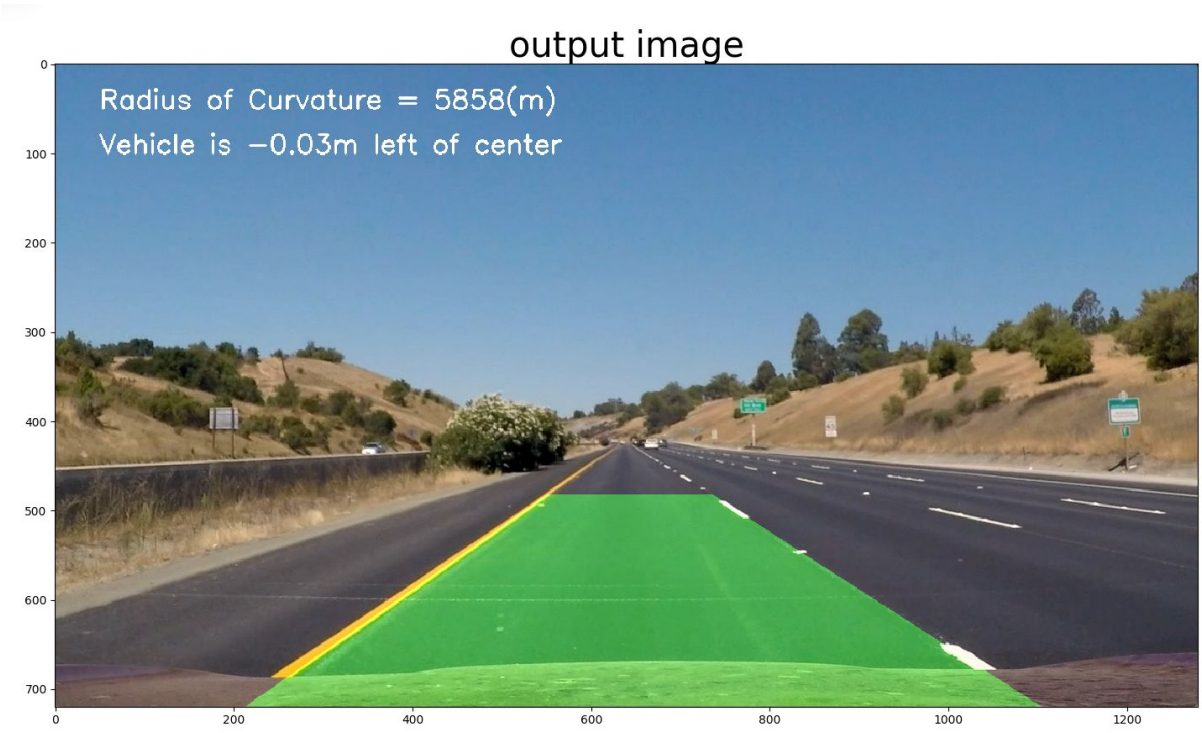
```
 for idx in range(self.image.shape[0]):
            left_offs[idx] = ploty[idx] * leftfit[0] ** 2 + ploty[idx] * leftfit[1] + leftfit[2]
 self.offset_center = round(np.median(left_offs) + self.h_dist/2 -
(self.image.shape[1] * self.xm_per_pixel) / 2, 2)
```

Self.h_dist is the horizontal distance or the width of two lanes, we add up width of lanes and the median value of left_lane fitting x-axis value, then minus the center x-axis of the image.

If offset is bigger than zero, the vehicle is on the left side of the center;
If offset is less than zero, the vehicle is on the right side of the center.

Finally, we have following outputs:





5 reinforcement strategies

We use a Line class to archive all past information, like left lane fitting coefficients, right lane fitting coefficients, horizontal distance of lanes to let us strengthen the reliability of our lane detection design and smooth the output.
The structure of Line class is like:

def __init__(self, size): #size indicates how many of the past iterations will we stores
self.detected = False #was line been detected successfully based on previous fit
self.past_left_fits = [] #all successful past n iterations fitting coefficients
self.best_left_fit = None #averaged fitting overall last n iterations
self.past_right_fits = []
self.best_right_fit = None
self.past_dists = []
self.best_dist = None
self.cache_size = size #define the length of our archive information

1) Smoothness

Once we have secured the fitting coefficients of lanes, we can search within a margin area around previously fitting result to make our outputs more smooth.

Furthermore, to stablize the green polygon area, we take the average of all past successful polynomial fitting coefficients as the best fit to plot current iteration output. We take this strategy in following logic:

result_img = lane_obj.draw_fitting_lane(input_image, inv_M, lines.best_left_fit, lines.best_right_fit)

Result_img is our final output, inv_M is the inverse transformation matrix to project polygon from the perspective transformation back to original image, lines.best_left_fit and lines.best_right_fit are the averaged fitting coefficients of all past successful lane fitting coefficients that Line class archived.

2) Robustness

Sometimes, the search algorithm will fail to detect valid lane pixel information, such scenarios are mainly two kinds:

1) We have found zero pixels in sliding window search based on previous fitting range;
2) We have found pixels but new coefficients are either too far away from previous results or the horziontal distance of lanes is too much different from previous results.

In case 1), the reason might have sth to do with last fitting, we remove last fitting from our archive and take the best fit from previous results to plot this iteration;
In case 2), the reason might be various, thus we take the best fit to plot this iteration and continue to next iteration.
Both failure cases will be marked as “failure” which means we will make a whole new sliding search in the next iteration instead of searching around previous fitting.

6 challenges

In this project, we take a fixed ROI for perspective transformation, which might be problematic if the vehicle make a sharp turn that could twist the perspective of camera a bit. In such cases, which occurs in the harder_challenge video, we might either change a new ROI for perspective transformation or decide which pixels are resides on the ROI, instead of our current naive approach, simply cutting the image into two halfs for each lane searching.