Camera Calibration

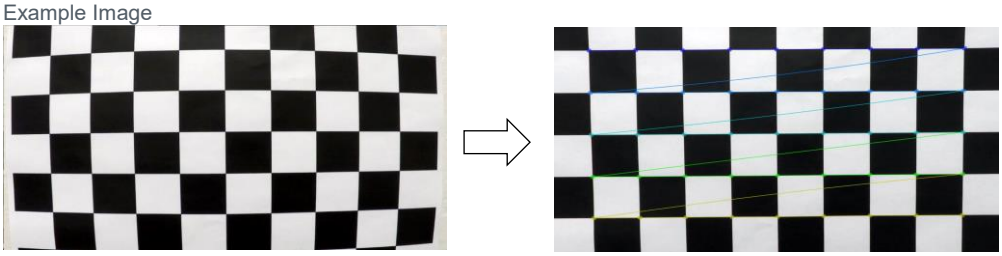
To remove distortions in an image greatest extend possible, multiple images from various angle and orientations are required. Thus all the provided calibration images in workspace are used to compute the camera matrix and distortion coefficients. General steps to compute the matrix and coefficients for a single image are mentioned below:

1. Manually count the number of visible corners in horizontal and vertical direction in the distorted images of the chess board
2. Convert the distorted color image to gray scale
3. The gray scale image is then used to find corners using command cv2.findChessboardCorners()
4. The command gives an array of corner indices as one of the output.
5. A mesh grid of dimension (nx , ny) is formed to represent real world chess board points on a plane. The mesh grid points are then reshaped into 3 dimensional array.
6. Calibration matrix and distortion matrix is then obtained using command cv2.calibrateCamera and Indices array and objective array as input.

The code presented below is taken from ***Cell : 1 and line no. = 9-25***

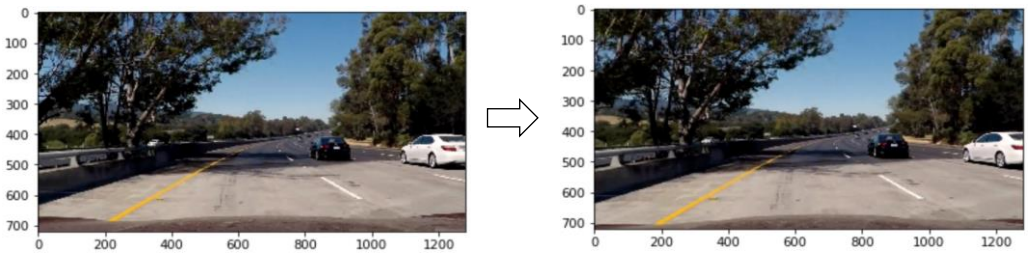
```
def cal_mat(images,nx,ny) :
    objpts = np.zeros((nx*ny,3),np.float32)
    objpts[:,:2]=np.mgrid[0:nx,0:ny].T.reshape(-1,2)
    o=[]
    i=[]
    for idx,fname in enumerate(images):
        image=cv2.imread(fname)
        gr_img=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
        ret,corners=cv2.findChessboardCorners(gr_img,(nx,ny),None)
        if ret==True:
            o.append(objpts)
            i.append(corners)

    ret,mtx,det,rvec,tvec=cv2.calibrateCamera(o,i,(720,1280),None,None)

    return ret,mtx,det
```

Example Image



However matrix and coefficients obtained by the calibration on a single image is not enough and it does not work. Therefore matrix and coefficients must be computed on the augmented arrays consisting of corner and objective points from all calibration images . For every calibration image provided in workspace, above steps are performed to find matrices. All the corner matrices are first joined together to form an augmented array and similarly an augmented Objective array is formed.

Examples of distortion-corrected image.

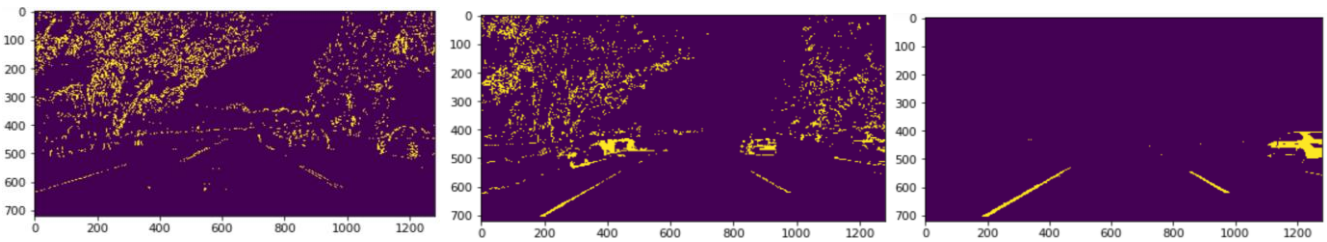


Binary images and combined binary images

I used binary images obtained from sobel operator in X direction along with S Channel and V channel. The thresholds were tuned to obtain maximum pixel points of curved striped lanes while discarding undesired pixels in the respective binary images. All three binary images where combined to obtained final binary 2D image to find lane. pixels.

The code presented below is taken from ***Cell : 1 and line no. = 33-54***

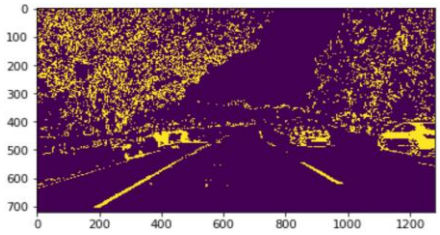
```
# Gradiant binary images
def grad_bin(gr_img,g_thresh,Ks,x,y):
    sobelx = cv2.Sobel(gr_img, cv2.CV_64F, x, y,Ks)
    scl=255*(np.absolute(sobelx)/np.max(sobelx));
    gbinary = np.zeros_like(gr_img)
    gbinary[(scl >= g_thresh[0]) & (scl <= g_thresh[1])] =
    return gbinary

# Extract S channel binary image
def S_channel(image,s_thresh):
    HLS=cv2.cvtColor(image,cv2.COLOR_RGB2HLS)
    S=HLS[:,:,2]
    s_binary = np.zeros_like(S)
    s_binary[(S >= s_thresh[0]) & (S <= s_thresh[1])] = 1
    return s_binary

# Extract V channel binary image
def V_channel(image,s_thresh):
    HLS=cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
    V=HLS[:,:,2]
    v_binary = np.zeros_like(V)
    v_binary[(V >= s_thresh[0]) & (V <= s_thresh[1])] = 1
    return v_binary
```



Sobel along X

S Channel of HLS

V channel of HSV



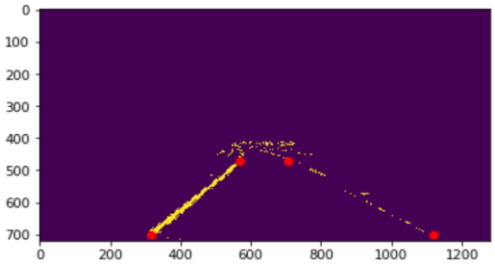
Combined binary image

Note : Combined binary images of the test images are stored in the folder output_images with name : cb_test.jpg*

Perspective transform

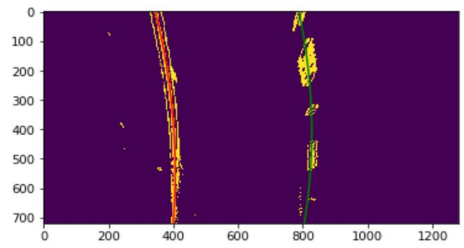
Firstly, The combined binary image is masked to remove undesired pixels from the image. This is one of the most crucial step where pixel points separated between left and right lanes. Using cv2.HoughLinesP() command, lines are obtained in the masked image and then pixels corresponding to left and right lanes are separated based on the slope of the respective line. A gradient range (0.4 to 1.7 or -1.7 to -0.4) is finalized to find pixel of respective lanes. Further threshold and other parameters are tuned to meet the project requirement. Lastly, four appropriate corner points are obtained by applying polyfit function on right lane and left lane points respectively. *The code presented below is taken from* ***Cell : 1 and line no. = 56 - 87***

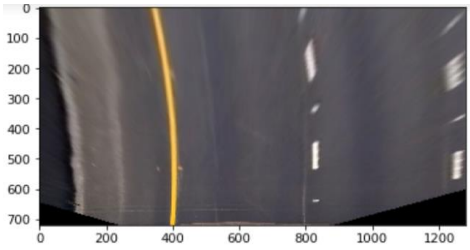
```
# Obtain 4 corners of rectangle for perspective tranformation
def drw_line(lines,image,low,up,Y_top,Y_bottom):
    left_line=[]
    right_line=[]

    for line in lines:
        for x1,y1,x2,y2 in line:
            slope= (y1-y2)/(x1-x2)
            if (((slope > low)&(slope <up))|((slope<-low)&(slope > -up))):
                if ((slope > low)&(slope < up)):
                    right_line.append((x1,y1))
                    right_line.append((x2,y2))
                elif ((slope<-low)&(slope > -up)) :
                    left_line.append((x1,y1))
                    left_line.append((x2,y2))

    temp1= np.array(left_line)
    YL=temp1[:,0]
    XL=temp1[:,1]

    temp2= np.array(right_line)
    YR=temp2[:,0]
    XR=temp2[:,1]

    left_fit = np.polyfit(XL,YL,1)
    right_fit= np.polyfit(XR,YR,1)

    X_top_R = int(right_fit[0]*Y_top + right_fit[1])
    X_bottom_R = int(right_fit[0]*Y_bottom + right_fit[1])
    X_top_L = int(left_fit[0]*Y_top + left_fit[1])
    X_bottom_L = int(left_fit[0]*Y_bottom + left_fit[1])
    return Y_top,Y_bottom, X_top_R,X_bottom_R,X_top_L,X_bottom_L
```



Lastly four appropriate destination points are chosen and cv2.warpPerspective() function is used to get warped image

```
    src=np.zeros((4,2),np.float32)
    src[0]=[X_top_L,Y_top]
    src[1]=[X_bottom_L,Y_bottom]
    src[2]=[X_top_R,Y_top]
    src[3]=[X_bottom_R,Y_bottom]

    dx1= 400
    dx2= 800
    dy1= 250
    dy2= 720
    dst=np.float32([[dx1,dy1],[dx1,dy2],[dx2,dy1],[dx2,dy2]])

    M = (cv2.getPerspectiveTransform(src, dst))
    warped = cv2.warpPerspective(mskd_edges,M,(S.shape[1],S.shape[0]), flags=cv2.INTER_LINEAR)
```



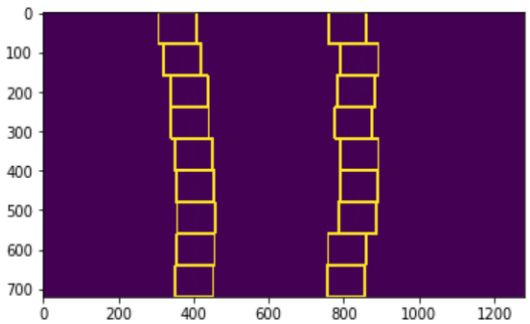


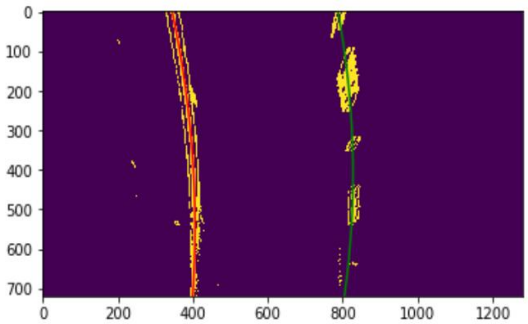
Note : Warped binary images of the test images are stored in the folder output_images with name : warped_test.jpg*

Lane-line pixels and fit their positions with a polynomial

After perspective transformation step, lane pixels are obtained by implementing sliding window method. In sliding window method, lane pixels are searched inside a smaller window of dimensions 80 x 100. The search begins from the bottom of the image and initially the x coordinate of the center of first image is chosen to be the value obtained from histogram. Further the next window is slides horizontally based on the location of the center of the pixels inside the search window. This goes on in a loop until all the rows of the image is searched and desired pixels of both left and right lanes are obtained .

The location of code for this part is ***Cell : 1 line no. 164 -222***





As the lane curvatures do not change much in subsequent frames, Localized search method is implemented by searching for the lane pixels in the vicinity of lane curve obtained in previous frame. To accomplish this task, global variables are used and lane curvature coefficients are stored. *The location of code for this part is* ***Cell : 1 line no. 224 -264***
In addition to that lane pixels from last frames are added/combined with lane pixels of current frame to obtained robust lane curvature. This ensures an efficient pixel search and lane curvature computation. In the project video test, the lane pixels are detected by localized search method in almost all of the frames except in very few detectable frames.
The location of code for this part is ***Cell : 1 line no. 160-161***

```
160     # Combine lane pixels from last frame
161     img=np.bitwise_or(warped,lf)
```

The radius of curvature of the lane and the position of the vehicle with respect to center.

Once the lane curvature coefficients are obtained using polyfit function, Lane radius and position of the vehicle in real world is obtained by axis transformation. The transformation coefficients are referred from the lessons. The radius of curvature of each lane is computed for the bottom pixels of the image representing the position of car. The transformation steps are shown in code below :

```
293     # Axis transformation and real world positon and lane radius computation
294     ym_per_pix = 30/720
295     xm_per_pix = 3.7/700
296
297     left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
298     right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2)
299
300     # Radius calculation at y=720
301     y_eval=720
302     left_curverad =  round(((1+(2*left_fit[0]*y_eval*ym_per_pix+left_fit[1])**2)**1.5)/(2*abs(left_fit[0])),1)
303     right_curverad = round(((1+(2*right_fit[0]*y_eval*ym_per_pix+right_fit[1])**2)**1.5)/(2*abs(right_fit[0])),1)
304
305     # Vehicle position calculation with respect to the center
306     pos=round((((X_bottom_R+X_bottom_L)*0.5)-640)*xm_per_pix,2)
```

Example of final images from the out put video

Out put Video link

https://view5639f7e7.udacity-student-workspaces.com/view/CarND-Advanced-Lane-Lines/output_images/project_video_sol.mp4

Discussion : 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 pipeline fails to find lane pixels for when the vehicle encounters bump or a jerk. Also if pipeline is vulnerable to changes in lane surfaces, shadows, tire marks, black patches, or vehicles in front or adjacent lane . It required lots of tuning of Hough lines parameters to get suitable pixels. Some times pixels are not easily obtained and pipeline fails to find the lanes. As solution to these problems, I have to abstract thoughts. First, a filter can be designed which can identify anomalous and undesired pixel by comparing the last frame. Second, a dynamic mask can be designed which can change its shape according to the information received on lanes from last frame.
Examples of pipe line failure: