Project 2: Advanced Lane Lines

The goal of this project is to write a software pipeline, that will identify positions of the left and right lane lines on the road, in the images and video stream.

This is achieved in following steps:

1. Camera Calibration – Calculate calibration matrix and distortion coefficient for the installed camera from a set of chessboard images
2. Undistortion – Apply this distortion correction to raw road images
3. Color and gradient thresholds – To obtain a binary image containing maximum pixels belonging to lane lines
4. Perspective Transform – obtain birds eye view of each road image
5. Fit polynomials to lane lines
6. Calculate radius of curvature and offset from center for each lane line

---

Camera Calibration

A camera mounted on the car (used to capture test images and videos) converts 3D image of the road into 2D. This conversion causes the generated images to become distorted – bent at the edges and lines may appear incorrectly skewed.

The distortion can be either radial or tangential and it needs to be corrected for an accurate lane lines detection in self driving cars

Radial distortion correction –

$$x_{distorted} = x_{ideal}\,(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{distorted} = y_{ideal}\,(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Tangential distortion correction –

$$x_{corrected} = x + (2p_1 xy + p_2(r^2 + 2x^2)$$

$$y_{corrected} = y + (2p_2 xy + p_1(r^2 + 2y^2)$$

Camera calibration for distortion correction can be performed using multiple images of a chessboard taken from different angles and applying functions from OpenCV library in python.

As seen in the attached snippet of the code (figure 3), I followed following steps for camera calibration –

1. Create empty arrays of object points (3D coordinates of undistorted chessboard image) and image points (2D coordinates of distorted chessboard image)
2. Object and image point here correspond to 9X6 corners visible in the chessboard images
3. A for loop is created to load the images of the chessboard one by one and convert them to grayscale
4. Cv2.findChessboardCorners() function is used to find chessboard corners in the grayscaled image
5. Once the 9X6 corner points are detected at the end of one for loop, their coordinates are added to the object and image points array.
6. Copy of the original image – img is used to overlay these corner points using cv2.drawChessboardCorners() function

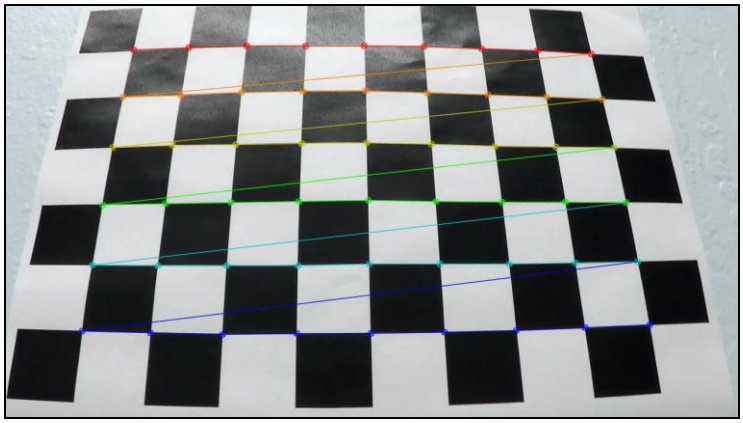


Figure 1:draw chessboard corners

7. Cv2.CalibrateCamera() function is used to compute the calibration coefficients. This function returns camera matrix (mtx), distortion coefficients (dist) along with rotation and translation vectors. Mtx and dist from this function are used later in the software pipeline to undistort test images using cv2.undistort()

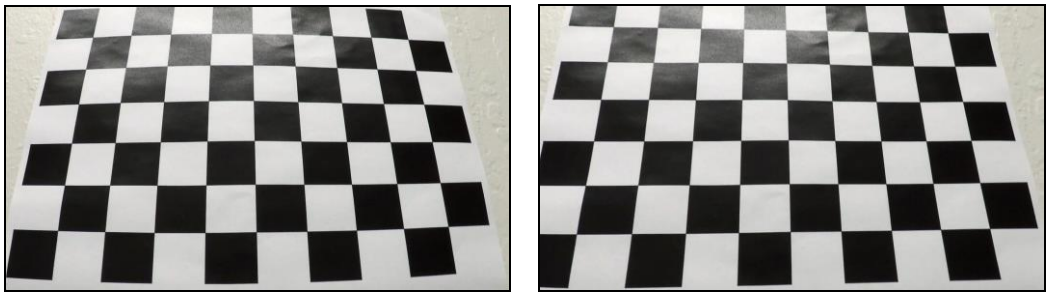


Figure 2: Image undistortion

```
import glob
#from slide_window import slide_window
%matplotlib qt

#Camera Calibration

#List of all images in the folder 'camera_cal' using glob API
imgs = glob.glob("C:/Users/t0794dm/Desktop/SDC_ND/project_2/CarND-Advanced-Lane-Lines/camera_cal/calibration*.jpg")

#object and image points arrays
objpoints = [] #image points - hold 2D coordinates of the distorted test image
imgpoints = [] #object points - hold 3D coordinates of real undistorted chessboard corner points

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

#image points
for index, name in enumerate(imgs):
    #read the images
    img = mpimg.imread(name)

    #convert the image to grayscale
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    #find chessboard corners in the grayscaled image
    ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
    #corners

    #if 9X6 corner points exist, add those coordinates to the imgpoints array
    if ret == True:
        imgpoints.append(corners)
        objpoints.append(objpts)

        #again, if corners exists, then draw them on the copy of the image - (corners found)
        img = cv2.drawChessboardCorners(img, (9,6), corners, ret)


#find dimentions of the images
img = mpimg.imread ('C:/Users/t0794dm/Desktop/SDC_ND/project_2/CarND-Advanced-Lane-Lines/camera_cal/calibration3.jpg')
img_dim = (img.shape[1], img.shape[0])

#calibrate camera
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_dim, None, None)
```

Figure 3: Code for Camera Calibration

Software Pipeline (for images)

1. **Distortion Correction** –
   Distortion correction coefficients and camera matrix calculated via cv2.calibrateCamera() function are applied to each image in the test image folder to undistort them, using cv2.undistort()

Figure 4:image undistortion

2. **Gradient Threshold (Sobel Operator) –**

After images are undistorted, we convert it into a binary image in order to perform further filtering to highlight lane lines in the image

First step towards achieving that is applying Sobel operator. I defined a function abs_sobel_thresh() to apply sobel operator to each test image and take derivative in the x and y direction.

Keeping the kernel size at 3, threshold windows are chosen such that maximum pixels from both lane lines are highlighted with minimum noise from other objects

X gradient threshold – [15,255]

X gradient threshold – [25,255]

```
def abs_sobel_thresh(img, orient='x', sobel_kernel = 3, thresh_min=0, thresh_max=255):
    #convert image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    #apply x and y  gradient and convert to absolute values
    if orient == 'x':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel))
    if orient == 'y':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel))
    #rescale to 8 bit integer
    scaled_sobel = np.uint8(abs_sobel/(np.max(abs_sobel)/255))
    # Create a copy and apply the threshold
    binary_output = np.zeros_like(scaled_sobel)
    binary_output[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    # Return the result
    return binary_output
```

3. **HLS and HSV Threshold –**

In addition to gradient threshold, I have applied saturation threshold (HLS) to capture lower saturation lane lines and value saturation to reduce the effect of right light and shadows from the image

S threshold – [110,255]

V threshold – [60,255]

In the final software pipeline, gradient and SV threshold is applied with OR condition so that lane line detection failure with gradient threshold because of lighting is captured by SV thresholds

Example of final binary image after color and gradient threshold is shown below

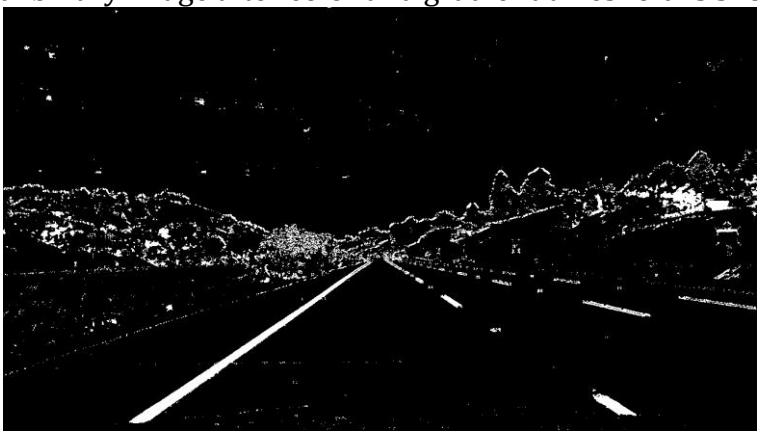


Figure 5 – Binary image after thresholds

```
def hls_threshold(img,thresh_min=0, thresh_max=255):
    #convert color image in hls channels
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    #extract s channel
    s_channel = hls[:,:,2]
    #create binary image copy and apply thresholds
    binary_output = np.zeros_like(s_channel)
    binary_output[(s_channel > thresh_min) & (s_channel <= thresh_max)] = 1

    #return result
    return binary_output

def rgb_threshold(img,thresh_min=0, thresh_max=255):
    #extract r channel
    r_channel = img[:,:,0]
    #create binary image copy and apply thresholds
    binary_output = np.zeros_like(r_channel)
    binary_output[(r_channel > thresh_min) & (r_channel <= thresh_max)] = 1

    #return result
    return binary_output

def hsv_threshold(img,thresh_min=50, thresh_max=255):
    #convert color image in hsv channels
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    #extract v channel
    v_channel = hsv[:,:,2]
    #create binary image copy and apply thresholds
    binary_output = np.zeros_like(v_channel)
    binary_output[(v_channel > thresh_min) & (v_channel <= thresh_max)] = 1

    #return result
    return binary_output
```

Figure 6: code for hls, hsv, rgb threshold function

4. **Perspective Transform** –

Once the binary image is created that highlights maximum pixels from lane lines, the next step is to perform perspective transform to generate a birds' eye view of the road image. This is done because ultimately we want to measure curvature of the lines. To do this we first define source and destination points. Source points are coordinates of 4 points on the test image that are approximately in the same plane and adequately cover both lane lines. I have selected source points such that perspective transform only captures lane lines while avoiding any noise from surrounding objects. I used an image Interactive window to detect x and y coordinates of source points. These source points are mapped to destination points that represent a rectangle of the same size as test image

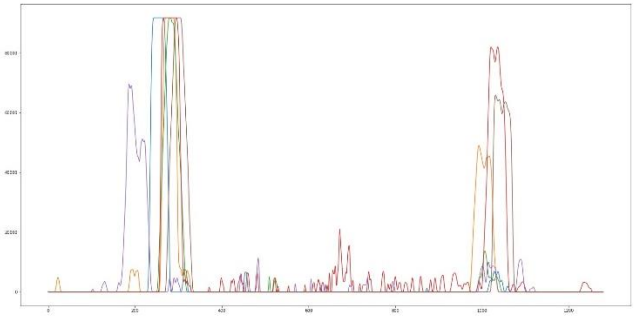
Cv2.getPerspectiveTransform() function is used to generate a transform, M and cv2.warpedPerspective() function is used to create a warped image with bird's eye view of the lanes.

Figure 7 shows an example of the road image after perspective transform. One of my criteria to define source points was that both lane lines should appear approximately parallel to each other after perspective transform

Figure 7: warped image with bird's eye view of the road

```
#perspective transform
# map 4 rectangular source points to 4 destination points
img_dim = (binary_image.shape[1],binary_image.shape[0]) #[1280,720]
src = np.float32(   #change these values to represent trapezoid
    [[550,455], #[551,486] #[555,455]
     [720,455], #[765,486] #[725,455]
     [1280,685], #[1093,682] #[1280,680]
     [0,685]]) #[290,690] #[0,680]
dst = np.float32(
    [[0,0],#[256,0]
     [1280,0],#[1024,0]
     [1280,720], #[1024,720]
     [0,720]])#[256,720]
#Calulate perspective transform, M
M = cv2.getPerspectiveTransform(src, dst)
# Inverse Perspective transform
Minv = cv2.getPerspectiveTransform(dst, src)
# warped image using linear interpolation
warped_image = cv2.warpPerspective(binary_image, M, img_dim, flags=cv2.INTER_LINEAR)
```

5. Finding Lane Lines (polynomial fit)
   Once the binary warped image of the lane lines is generated, the next step is to locate pixels belonging to left and right lane lines. For this I followed histogram technique as described in the course.
   - First, I took a histogram of the bottom half of the image (bottom half because we want to locate left and right lane pixels for the first frame starting at the bottom). Following plot shows histogram for all test images.



- Start point of the left and right lane is easily noticed in the above image and can be calculated from x and y coordinates of the image
- Using two highest peaks of the histogram as starting point, I used sliding window technique to determine location of frames moving further along the road
- For this project I selected 9 frames in one image, with a margin of 100 and minimum pixel concentration is kept at 50.
- Looping through each window, I found boundaries of current window and number of activated pixels in it. If the number of pixels in the subsequent window is more than 50, I reset the window location to the new mean position. This is performed in the function find_lane_pixels() in my code. A n example of test image 1 shown below.

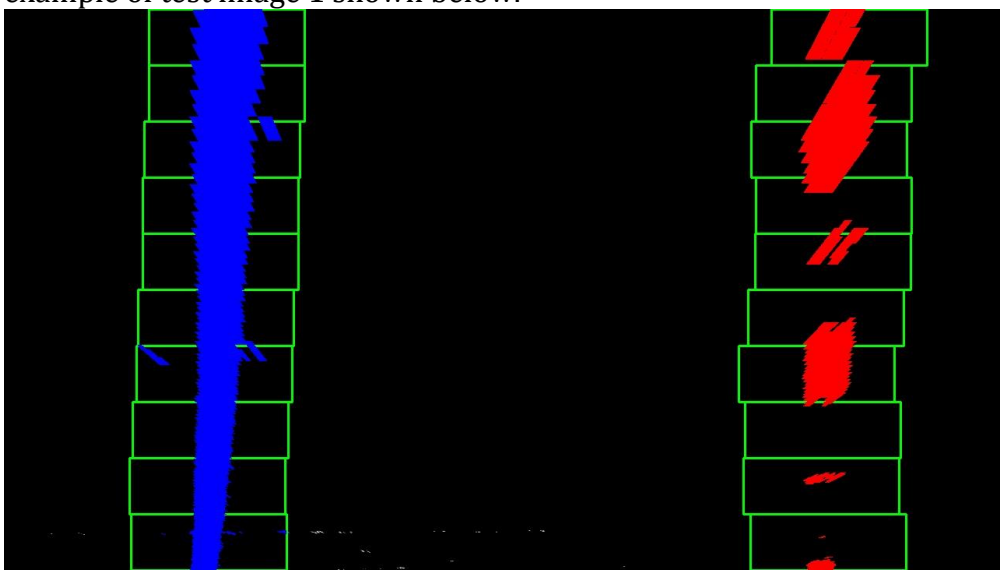


Figure 8: lane lines tracked

Once lane lines are tracked in the binary warped image, next step is to fit a polynomial through it to define left and right lanes.

- Using left and right pixel coordinates, I used np.polyfit to generate a polynomial for left and right lane
- To plot these curves on the top of the test image, I created a different array of x and y coordinates of lanes, using polynomial equation and an array of y coordinates in ploty (starting from the bottom of the image in an interval of 1)
- To visualized this on the binary image, left lane pixels are shown in blue and right lane in red. The polynomial fit is shown in green on the top of the lane lines

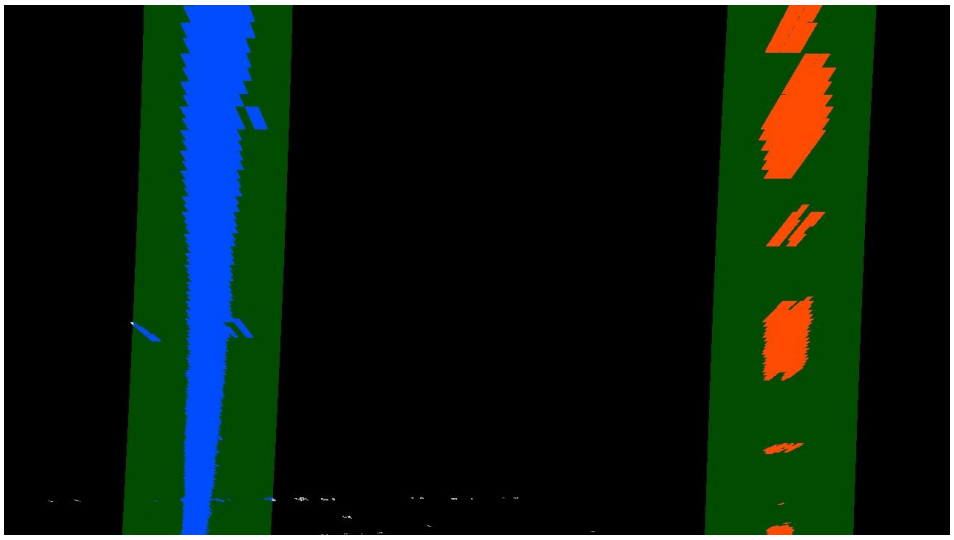


Figure 9: Lane lines visualization

For running this pipeline on videos, we don't need to do this process from the beginning for every frame, we can just search in the margin of previous lane lines position. For this I grabbed x and y pixel position from earlier frame along with the polynomial fit, and calculated location of pixels that fall into green shaded area shown above.

This way all the relevant lane pixels are found and we fit a second order polynomial, again for both lanes, repeating the step above, I generated left and right lanes [x,y] coordinates to plot on the image.

As a final step, lane lines are plotted on the original image using cv2.addWeighted().

Figure 10: lane lines on the original image

```python
def find_lane_pixels(binary_warped):
    # Take 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 starting point for the left and right lines - these will be left and right peaks of histogram
    midpoint = np.int(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # Define Hyperparameters
    # number of sliding windows
    nwindows = 9
    # width of the windows +/- margin
    margin = 100
    # minimum number of pixels found to recenter window
    minpix = 50

    # Calculate height of windows
    window_height = np.int(binary_warped.shape[0]//nwindows)
    # 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

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

    # For loop to 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
        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin
        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin

        # 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)

        # 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)

        # If > 50, recenter next window to new mean position
        if len(good_left_inds) > minpix:
            leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
        if len(good_right_inds) > minpix:
            rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

    # Concatenate the arrays of indices
    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

    # 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]

    return leftx, lefty, rightx, righty, out_img
```

Figure 11: function to find lane pixels in a binary image

```
def fit_polynomial(binary_warped):
    # Find lane pixels
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(binary_warped)

    # Fit a second order polynomial to each lane
    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, left_fit, right_fit, ploty, left_fitx, right_fitx
```

Figure 12: Function to fit polynomial both lanes

6. Radius of Curvature and Offset Calculation
    - For additional information, I calculated radius of curvature for both left and right lane lines and offset from center (assuming camera I mounted at the center of the image/car)
    - Lane is assumed to be 30 m long and 3.7 m wide. This information is used ot convert pixels into meters. using np. Polyfit(), I fit a polynomial for lanes (this time in units of meters)
    - Offset is distance by which center ofteh lane lines is off from center of the car/image. Center of the lane line is average of x coordinate of left and right lanes and center of the car is image width/2. Difference, offset is printed on the final output image, as shown below

Figure 13: final output image

Issues and possible improvements to the pipeline

Possible issues and improvements that can be implemented to my current pipeline

- Four coordinates of the rectangle selected for perspective transform works well with project video. I have eyeballed them for project video, but it captures a lot of noise other than lane line pixels in case of challenge videos. More concise region will increase chances of capturing only lane lines
- Additional thresholds need to be added to neutralize effect of bright light, tree shadow, cracks in the road and color change of the road.
- Crack in the road and dark marks beside right lane in the challenge video can be eliminated by taking only x gradient
- Kernel size can be increased to avoid spurring gradients from tree shadow road shade
- Perspective transform does not work well for challenge video where car is going down a curvy road. This effect can be minimized by reducing height of image during perspective transform, so as to only capture plane closer to the car