Advanced Lane Lines Finding

Author: Guang Yang
The goals of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given 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.

Camera Calibration

Each lens has its unique lens distortion based on its parameters. The most common distortions include radial distortion and tangential distortion. Since we will be using camera to track lane lines, as well as determine curvature of the road, it becomes critical for us to remove these optical distortions. Based on the documentation from OpenCV, the radial distortion can be modeled with the following function:

$$x_{raddistort} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{raddistort} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6$$

and the tangential distortion is modeled as the following:

$$x_{tandistort} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{tandistort} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

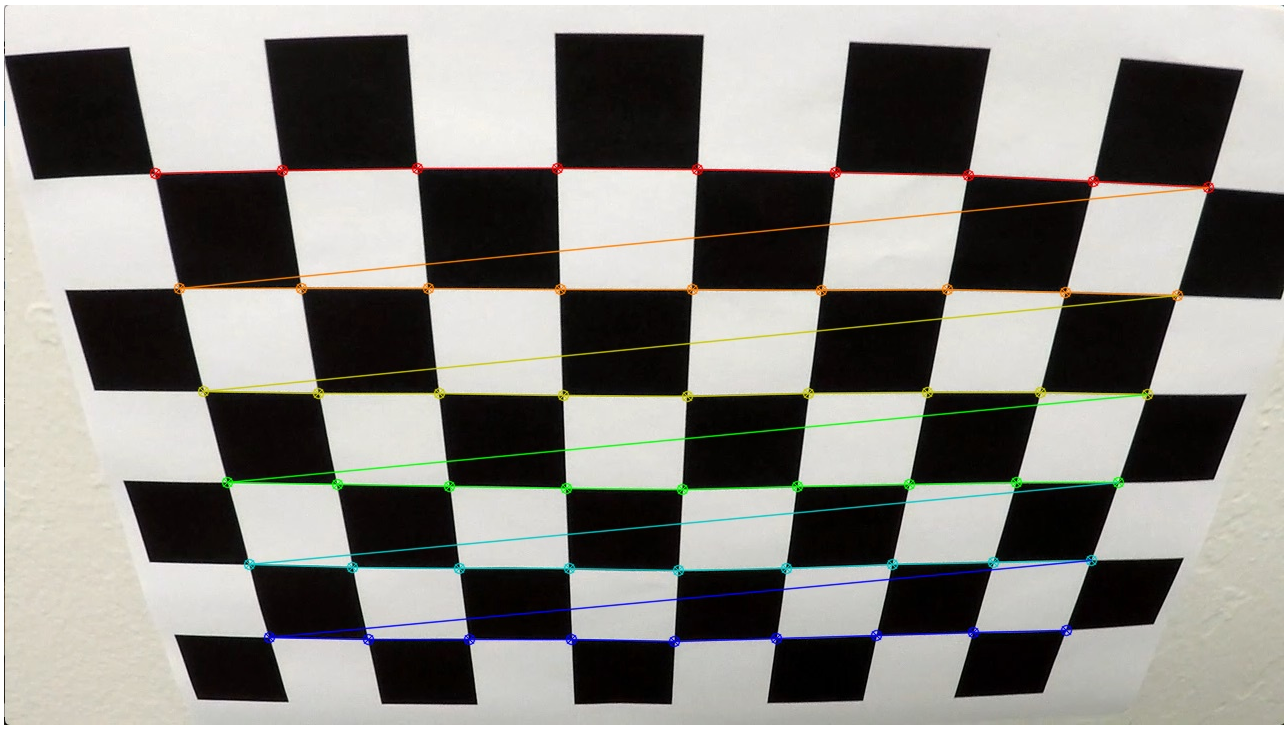
Therefore, we can define the distortion coefficients d as:

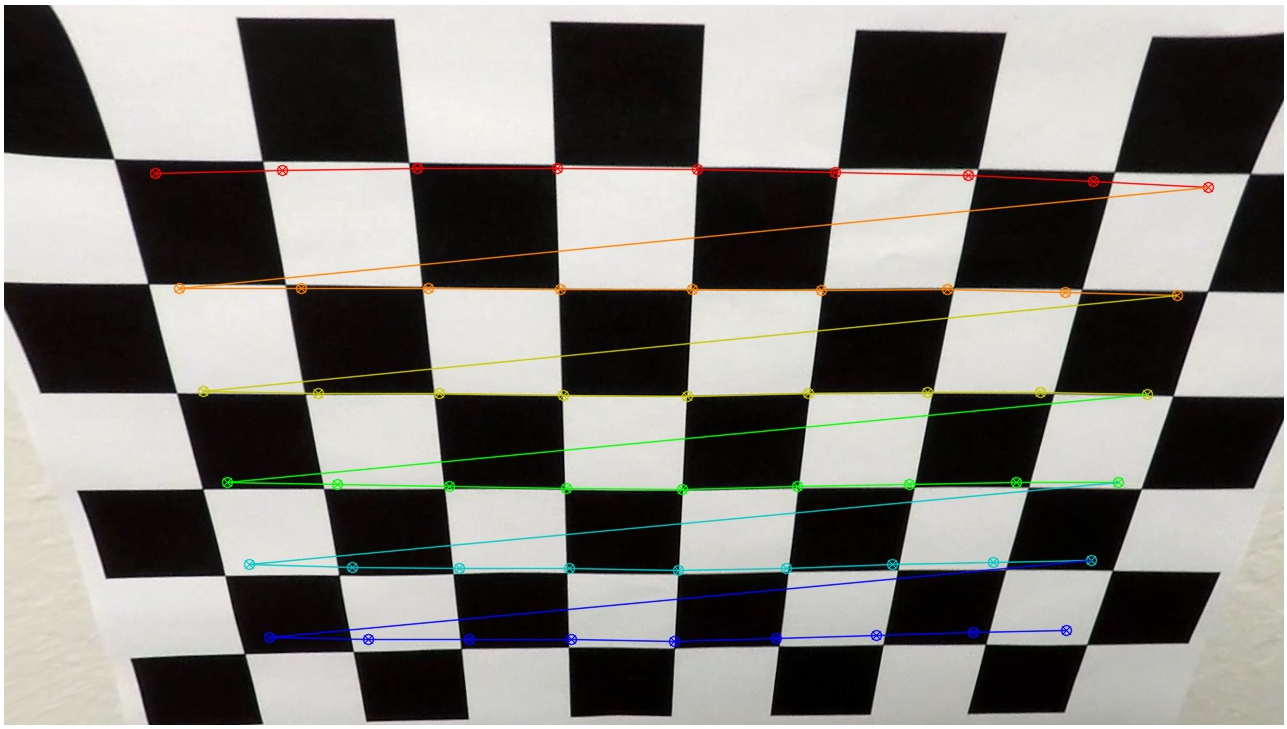
$$d = [k_1, k_2, p_1, p_2, k_3]$$

The camera matrix is defined as:

$$\mathbf{C} = \begin{bmatrix} f_x & 2 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where fx, fy is the focal length of the camera lens and cx, cy is the optical center. To optain camera matrix $\mathbf{C}$ and distortion coefficient d, we can utilize cv2.calibrateCamera( ) function from OpenCV. We can now transform the original chessboard image to the undistorted image:





The code is shown as the following:

```
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import json
%matplotlib qt

# 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)

# 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.
d = 0
# Make a list of calibration images
images = glob.glob('../camera_cal/calibration*.jpg')

def camera_calibration_matrix(objpoints, imgpoints, img):
    #This function takes chessboard images with known dimensions and obtain camera calibration matrix
    localImg = np.copy(img)
    grayImg = cv2.cvtColor(localImg,cv2.COLOR_BGR2GRAY)
    ret, mtx, dist, rvecs, tves = cv2.calibrateCamera(objpoints, imgpoints, grayImg.shape[::-1],None,None)
    return mtx,dist
```

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

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(grayImg, (9,6),No
ne)



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

        mtx,dist = camera_calibration_matrix(objpoints, imgpoi
nts, img)

        # Undistort images
        undistortedImg = cv2.undistort(img,mtx,dist, None, mt
x)
        # Draw and display the corners
        originalImg_with_chessboard = cv2.drawChessboardCorner
s(img, (9,6), corners, ret)
        undistortedImg_with_chessboard = cv2.drawChessboardCor
ners(undistortedImg, (9,6), corners, ret)

        outputImgPath = "./calibrated_images/"
        cv2.imwrite(outputImgPath+"original_%d.jpg"%d,original
Img_with_chessboard)
```

```
        cv2.imwrite(outputImgPath+"undistorted_%d.jpg"%d,undistortedImg_with_chessboard)
        d+=1


np.savetxt("camera_calibration_matrix.csv",mtx, delimiter=",")
np.savetxt("camera_distortion_coefficients.csv",dist, delimiter=",")
```

Through this process, we can also obtain both camera calibration matrix mtx and distortion coefficient $dist$, which will be used later.

Image Distortion Correction

Once we obtain the camera matrix and distortion coefficients, we will use it for distortion correction for lane line images. To demonstrate this process, I will describe how I apply the distortion correction to one of the test images like this one:

Sobel Opeartor for gradient measurements

The Sobel Opeartor are used to perform convolution on the original image to determine how gradient changes. Intuitvely, we want to measure the change of gradient with respect to both x axis and y axis, as well as direction of gradient.
In this project, I use 3×3 Scharr filters (set ksize = -1) kernels, namely $\mathbf{S_x}$ and $\mathbf{S_y}$ for the two axises.

Gradient along x axis:

$$\mathbf{S_x} = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ 3 & 0 & 3 \end{bmatrix},$$

Gradient along y axis:

$$\mathbf{S_y} = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

The magnitude $\mathbf{S}$ and direction θ of the gradient can be easily obtained through trigonometry:

$$\mathbf{S} = \sqrt{\mathbf{S_x}^2 + \mathbf{S_y}^2}$$

$$\theta = atan(\frac{\mathbf{S_y}}{\mathbf{S_x}})$$

The Sobel operator code is the following:

```
def abs_sobel_thresh(img, orient='x', sobel_kernel=-1, thresh=
(60, 150)):
    # Calculate directional gradient
    # Apply threshold
    thresh_min = thresh[0]
    thresh_max = thresh[1]
    grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    if orient =='x':
        sobelDir = cv2.Sobel(grayImg, cv2.CV_64F, 1,0, ksize=s
obel_kernel) #x oritentaion
    elif orient =='y':
        sobelDir = cv2.Sobel(grayImg, cv2.CV_64F, 0,1, ksize=s
obel_kernel) #y orientation

    absSobelDir = np.absolute(sobelDir)
    scaled_sobel = np.uint8(255*absSobelDir/np.max(sobelDir))
    grad_binary = np.zeros_like(scaled_sobel)
    grad_binary[(scaled_sobel >= thresh_min) & (scaled_sobel <
= thresh_max)] = 1
    return grad_binary

def mag_thresh(image, sobel_kernel=-1, mag_thresh=(50, 150)):
    # Calculate gradient magnitude
    # Apply threshold
    thresh_min = mag_thresh[0]
    thresh_max = mag_thresh[1]
    grayImg = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sobelx = cv2.Sobel(grayImg, cv2.CV_64F, 1,0,ksize=sobel_ke
rnel) #x oritentaion
```

```
    sobely = cv2.Sobel(grayImg, cv2.CV_64F, 0,1,ksize=sobel_ke
rnel) #y orientation
    absSobelxy = np.sqrt(sobelx**2+sobely**2)
    scaled_sobel = np.uint8(255*absSobelxy/np.max(absSobelxy))
    mag_binary = np.zeros_like(scaled_sobel)
    mag_binary[(scaled_sobel >= thresh_min) & (scaled_sobel <=
thresh_max)] = 1
    return mag_binary


def dir_threshold(image, sobel_kernel=-1, thresh=(0.5, np.pi/
2)):
    # Calculate gradient direction
    # Apply threshold
    thresh_min = thresh[0]
    thresh_max = thresh[1]
    grayImg = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sobelx = cv2.Sobel(grayImg, cv2.CV_64F, 1,0,ksize=sobel_ke
rnel) #x oritentaion
    sobely = cv2.Sobel(grayImg, cv2.CV_64F, 0,1,ksize=sobel_ke
rnel) #y orientation
    gradientDirectionImg = np.arctan2(np.absolute(sobely), np.
absolute(sobelx));
    dir_binary = np.zeros_like(gradientDirectionImg)
    dir_binary[(gradientDirectionImg >= thresh_min) & (gradien
tDirectionImg <= thresh_max)] = 1
return dir_binary
```

Here is a comparison of original image and the image after applying Sobel thresholding:

Color Thresholding

Another thresoholding techqniue is used in color space. Intuitively, we want to extract the color of interest from the image that resembles the lane line (In this case, yellow and white colors) The standard RGB color space is a three dimensional vector space with Red, Green and Blue for each axis. In theory, we can directly perform color thresholding in RGB color space. However, the surrounding light can change

dramatically in real-life situation, which can lead to poor performance and various of other issues. Alternatively, we can represent an image in Hue, Saturation and Value (HSV) color space or Hue, Lightness and Saturation (HLS) color space. Why do we want to perform color thresholding in those color spaces? Well, the use of Hue and Saturation are critical because they are indepentdent of brightness.
For the project, I decide to use HLS color space for color thresholding. To convert the image from RGB to HLS, I use the OpenCV function cv2.cvtColor(im, cv2.COLOR_RGB2HLS). After some testings, the saturation channel (S channel) performs the best in terms of extracting lane line, but I will combine it with Hue thresholding to get more degree of freedom. The following code demonstrate how a binary image is generated through S and H channel thresholding.
The code is the following:

```
def color_threshold(imgage, S_thresh=(0, 255),H_thresh=(0, 25
5)):
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    H = hls[:,:,0]
    S = hls[:,:,2]
    binary_output = np.zeros_like(S)
    binary_output[(S > S_thresh[0]) & (S <= S_thresh[1])& (H >
H_thresh[0])& (H <= H_thresh[1])] = 1
return binary_output
```

Here is an example of applying color thresholding:

By combining both gradient and color threshold together, we can achieve more degree of freedom to filter out unwanted background:

```
def threshold(image,ksize, abs_sobel_thresh_param_x, abs_sobel
_thresh_param_y ,mag_thresh_param, dir_thresh_param,saturation
_thresh_param, hue_thresh_param):
    # Convert original RGB image to Gray Image
```

```
    grayImg = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)


    # Gradient Threshold
    gradx = abs_sobel_thresh(grayImg, orient='x', sobel_kernel
=ksize, thresh=abs_sobel_thresh_param_x)
    grady = abs_sobel_thresh(grayImg, orient='y', sobel_kernel
=ksize, thresh=abs_sobel_thresh_param_y)
    mag_binary = mag_thresh(grayImg, sobel_kernel=ksize, mag_t
hresh=mag_thresh_param)
    dir_binary = dir_threshold(grayImg, sobel_kernel=ksize, th
resh=dir_thresh_param)
    combined_gradient_binary = np.zeros_like(dir_binary)
    combined_gradient_binary[((gradx == 1) & (grady == 1)) |
((mag_binary == 1) & (dir_binary == 1))] = 1


    # Color Threshold
    color_binary = color_threshold(image, S_thresh=saturation_
thresh_param,H_thresh=hue_thresh_param)


    # Combine Gradient Threshold and Color Threshold
    image_after_threshold_binary = np.zeros_like(color_binary)
    image_after_threshold_binary[(combined_gradient_binary ==
1)|(color_binary == 1)]=1
return image_after_threshold_binary
```

To further improve the result, I performed a masking on thresholded image by using the following code:

```
def mask_image(img, vertices):
    mask = np.zeros_like(img)
    if len(img.shape) > 2:
        channel_count = img.shape[2]  # i.e. 3 or 4 depending
on your image
        ignore_mask_color = (255,) * channel_count
```

```
    else:
        ignore_mask_color = 255
    #filling pixels inside the polygon defined by "vertices" w
ith the fill color
    cv2.fillPoly(mask, vertices, ignore_mask_color)
    #returning the image only where mask pixels are nonzero
    masked_image = cv2.bitwise_and(img, mask)
return masked_image
```

Here is a example of processed image using combined thresholding technique and image masking:

Perspective Transform

The code for my perspective transform includes a function called `perspective_transform(image)`, as shown below:

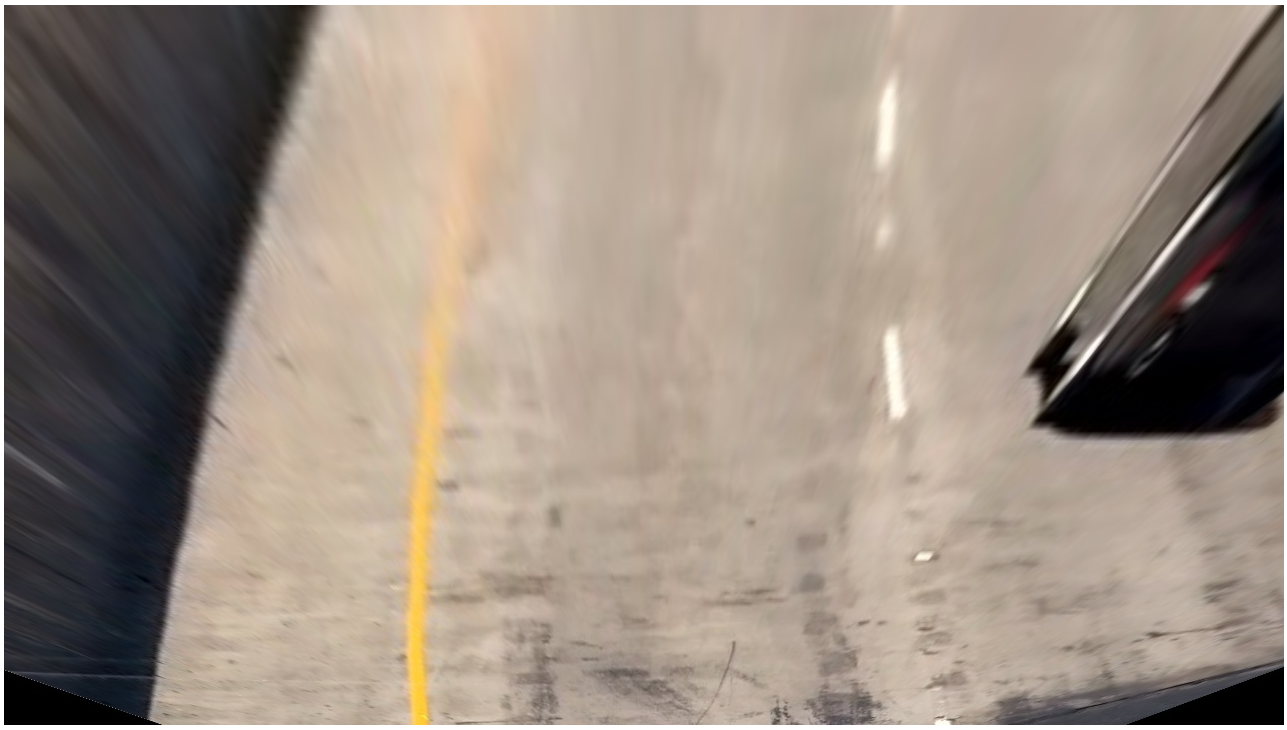
```
def perspective_transform(image):
    if len(image.shape) > 2:
        height, width, chanels = image.shape
    else:
        height, width = image.shape

    src = np.array([[(width*0.1,height*0.9),
                     (width/2.3,height/1.6),
                     (width/1.7,height/1.6),
                     (width*0.9,height*0.9)]],dtype=np.float32)

    dst = np.array([[(0.2*width,height),
                     (0.2*width,0),
                     (0.8*width,0),
                     (0.8*width,height)]],dtype=np.float32)
```

```
    transform_matrix = cv2.getPerspectiveTransform(src,dst)
    top_down_image = cv2.warpPerspective(image, transform_matrix, (width,height), flags = cv2.INTER_LINEAR)
return top_down_image, transform_matrix
```

To make the code more general to cameras with different resolution, here is the source and destination chart with respect to image size (In this project, width = 1280, height = 720):

Source	Destination
width × 0.1, height × 0.9	0.2 × width, height
width/2.3,height/1.6	0.2 ×width, 0
width/1.7,height/1.6	0.8 ×width, 0
width × 0.9,height ×0.9	0.8 ×width, height

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.



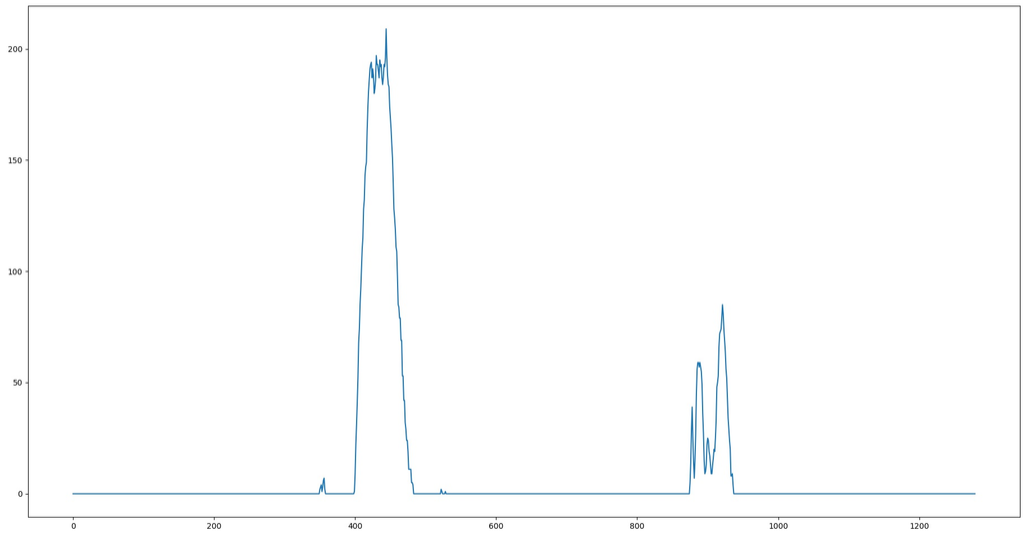
Lane Line Detection

The next step is to fit lane line using a polynomial as the following:

$$y = ax^2 + bx + c$$

The fitting process is nothing more than finding the correct coefficients for the polynomial function. Before fitting the lane line, we need to process raw images with functions that we have defined above and output thresholded, masked, transformed binary images. We name these "warped_images". Here is an example of a warped image:

One approach to detect lane lines is to count the number of pixels along the x-axis and then get a histogram from it.



The location where the highest peak is where the lane line is approximately located. From here, we can further improve the result using a sliding window technique. In this example, I created nine windows for each lane line and detected lane line from bottom and moved upward using the following code:

```
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)
    plt.plot(histogram)



    # Create an output image to draw on and visualize the resu
lt
    out_img = (np.dstack((binary_warped, binary_warped, binary
_warped))).astype(np.uint8)

    # Find the peak of the left and right halves of the histog
ram
    # These will be the starting point for the left and right
lines
    midpoint = np.int(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # 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

    # Set height of windows - based on nwindows above and imag
e shape
    window_height = np.int(binary_warped.shape[0]//nwindows)
    # Identify the x and y positions of all nonzero pixels in
the image
```

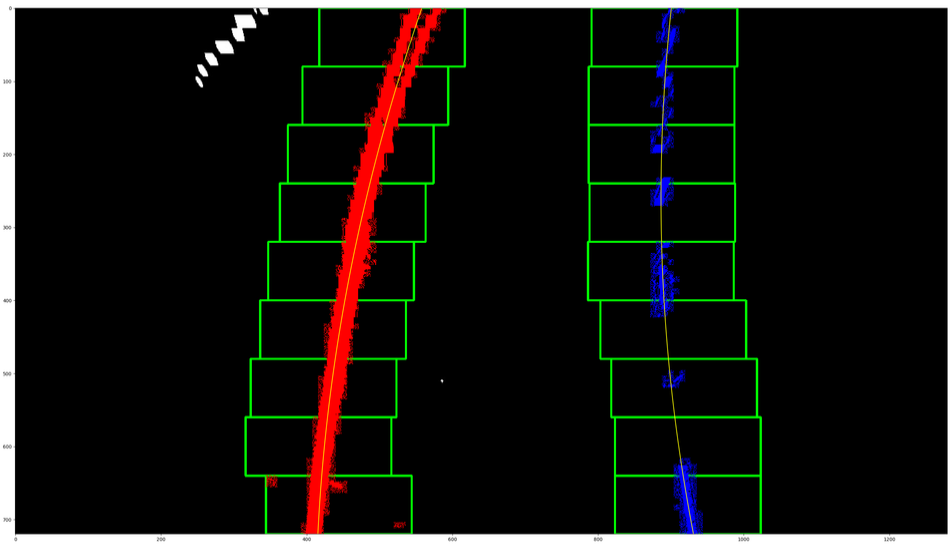
```
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    # Current positions to be updated later for each window in
nwindows
    leftx_current = leftx_base
    rightx_current = rightx_base

    # 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_he
ight
        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 wi
ndow #
        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy
< win_y_high) &
        (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_h
igh)).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 you found > minpix pixels, recenter next window o
n their 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_righ
t_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 full
y
        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



def fit_polynomial(binary_warped):

    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(b
inary_warped)
    # 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_wa
rped.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]*plot
y + 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]
    return left_fit, right_fit, ploty, leftx, lefty, rightx, r
ighty, left_fitx, right_fitx
```

Here is the final result:



Note in video, we can speed up the process for line fitting by using the coefficients we have obtained from previous frame. The idea is to search around the previous fitted lane line given a specific margin, so we can avoid searching the entire image. Here is the code I have used:

```
def fitpolynomial_with_previous_fitting_value(binary_warped, l
eft_fit_before, right_fit_before,ploty):
```

```
    margin = 120
    left_fit_current, right_fit_current = (None, None)
    img_shape = binary_warped.shape
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    left_lane_inds = ((nonzerox > (left_fit_before[0]*(nonzero
y**2) + left_fit_before[1]*nonzeroy +
                    left_fit_before[2] - margin)) & (nonzerox
< (left_fit_before[0]*(nonzeroy**2) +
                    left_fit_before[1]*nonzeroy + left_fit_bef
ore[2] + margin)))
    right_lane_inds = ((nonzerox > (right_fit_before[0]*(nonze
roy**2) + right_fit_before[1]*nonzeroy +
                    right_fit_before[2] - margin)) & (nonzerox
< (right_fit_before[0]*(nonzeroy**2) +
                    right_fit_before[1]*nonzeroy + right_fit_b
efore[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]

    left_fit_current = np.polyfit(lefty, leftx, 2)
    right_fit_current = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty_current = np.linspace(0, img_shape[0]-1, img_shape
[0])
```

```
    ### TO-DO: Calc both polynomials using ploty, left_fit and
right_fit ###
    left_fitx_current = left_fit[0]*ploty**2 + left_fit[1]*plo
ty + left_fit[2]
    right_fitx_current = right_fit[0]*ploty**2 + right_fit[1]*
ploty + right_fit[2]


return left_fit_current, right_fit_current, ploty, leftx, left
y, rightx, righty, left_fitx_current, right_fitx_current
```

Lane Lines Curvature and Car Position Estimation

Now we have the polynomial coefficients to fit both left lane line and right lane line. Now we want to extract more useful information such as road curvature and how much does the car off from the center. To calculate the curvature, I used the following equation:

$$R_{curve} = \frac{(1+(2ay+b)^2)^{\frac{3}{2}}}{|2a|}$$

To calculate the relative position of the car from the center, I fitted two lane lines with respect height (720 in this case) and use the center pixel (1280/2=640 in this case) to measure the absolute offset.

The code is here:

```
def measure_curvature_pixels(left_fit, right_fit, ploty, left
x, lefty, rightx, righty):
    '''
    Calculates the curvature of polynomial functions in pixel
s.
    '''
    # Define y-value where we want radius of curvature
    # We'll choose the maximum y-value, corresponding to the b
ottom of the image
    y_eval = np.max(ploty)


    left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])
**2)**1.5) / np.absolute(2*left_fit[0])
    right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit
[1])**2)**1.5) / np.absolute(2*right_fit[0])
```

```
    return left_curverad, right_curverad


def measured_curvature_meters(left_fit, right_fit, ploty, left
x, lefty, rightx, righty):
    '''
    Calculates the curvature of polynomial functions in meter
s.
    '''
    ym_per_pix = 30/720 # meters per pixel in y dimension
    xm_per_pix = 3.7/700 # meters per pixel in x dimension
    leftx = leftx[::-1]   # Reverse to match top-to-bottom in
y
    rightx = rightx[::-1]  # Reverse to match top-to-bottom in
y


    # Determine center position, assuming image 1280x720
    leftx_mid = left_fit[0]*720**2 + left_fit[1]*720 + left_fi
t[2]
    rightx_mid = right_fit[0]*720**2 + right_fit[1]*720 + righ
t_fit[2]
    center_offset_meter = abs((640-(rightx_mid+leftx_mid)/2)*x
m_per_pix)


    y_eval = np.max(ploty)


    left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pi
x, 2)
    right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per
_pix, 2)
    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix
+ left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
```

```
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pi
x + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr
[0])


return left_curverad, right_curverad,center_offset_meter
```

Final Result

After finish writing all the codes, here is the final result I have for a test image:

Pipeline (video)

Here's the Advanced Lane Lines Detection Video

Discussion

My approach to this project yields a really nice result for the project video. However, it fails to detect lane lines correctly in the challenged videos, where roads are curved greatly. I believe the reason for my algorithm to fail in those situation is because of the fixed masking vertices, i.e., I do not change the region of interest as the road situation changes. For future improvements, I think the mask vertices should change based on the calculated lane line curvature. In addition, having a second camera could also help since we could do feature matching in that case to detect lane lines more accurately.