Advance Lane Finding Project

Written by Jonas Chan Leong Sean for the Udacity Self Driverless Car Course

In this report, I will discuss the goals/steps taken throughout the duration of the project which are as follows:

1. Compute the camera calibration matrix and distortion coefficients with a given set of chessboard images.
2. Apply a distortion correction to raw images
3. Use multiple different image processing techniques e.g. colour transforms, gradients, etc. to create a thresholded binary image
4. Warp the image using perspective transform to rectify the binary image
5. Detect lane pixels and fit to find the left and right lane boundaries.
6. Determine the curvature of the lane and vehicle position with respect to the centre.
7. Unwarp and overlay the detected lane boundaries back onto the original image.
8. Output a visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

In this section of the report, I will discuss and explain in detail the steps taken and the challenges faced throughout the project. As per the requirements in the rubric, this report is structured in the following order:

1. Camera calibration
2. Pipeline (Single test image)
3. Pipeline (video)

This report contains code snippets from Advance_Lane_Lines.ipynb

Camera Calibration

Prior to calibrating the camera, I first import the set of images required to calibrate the camera. The code for this step is illustrated in the snippet below:

```
def import_calibration_images(path):
    images = glob.glob(path)
    return images
```

This method was used to import 36 checkerboard images taken from different positions which are used to calibrate the camera.

When the images have been successfully imported, I then began the calibration process by preparing "`object points`" or specifically 3D points in real world space, in this case, the 3D points of the chessboard corners in the world. The code for this step is illustrated in the snippet below:

```
def obtain_calibration_parameters(images):
    # prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((6*8,3), np.float32)
    objp[:,:2] = np.mgrid[0:8, 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.

    # Step through the list of calibration images and search for chessboard corners
    for idx, fname in enumerate(images):
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Find the chessboard corners
        ret, corners = cv2.findChessboardCorners(gray, (8,6), None)

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

    if debug == True:
        print("The number of images passed in is: " + str(len(images)))
        print("The number of objpoints obtained is: " + str(len(objpoints)))
        print("The number of imgpoints obtained is: " + str(len(imgpoints)))

    return objpoints, imgpoints
```

This is done with the assumption that the chessboard is fixed on the (x, y) plane at z = 0 so that the object points are the same for each calibration images. Hence, the object points or based on the snippet of the code above, "`objp`" is just a replicated array of coordinates, and the `objpoints` array is appended with a copy of it every time the chessboard corners have been successfully detected in the chessboard image. On the other hand, the `imgpoints` array is appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. The corners of the chessboard corners are computed using OpenCV's built-in function:

`cv2.findChessboardCorners()`

The method above returns the objpoints and imgpoints array. Using the `obtain_calibration_parameters()` method above, the calculated object points and image points were then passed on to calibrate the camera. The code for this step is illustrated in the snippet below:

```
def calibrate_img(img, objpoints, imgpoints):
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img.shape[0:2], None, None)
    return mtx, dist
```

With this function, using the values passed on from the `obtain_calibration_parameters()`, I was able to compute the camera calibration matrix and distortion coefficients specifically using OpenCV's built-in function:

`cv2.calibrateCamera()`

The values returned by this function is then passed on to the method that undistorts the image as illustrated in the snippet below:

```
def undistort_img(img, mtx, dist):
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    return undist
```

This function particularly uses the `cv2.undistort()` function to undistort the image using the camera calibration matrix and distortion coefficients.

The steps taken above is simplified into one single function called `start_camera_calibration()` which automatically calls all the functions mentioned above in order as illustrated below:

```
def start_camera_calibration():
    calibration_parameters = obtain_calibration_parameters(import_calibration_images(calibration_images_path))
    test_image = cv2.imread(calibration_test_image_path)

    mtx, dist = calibrate_img(test_image, calibration_parameters[0], calibration_parameters[1])
    warped = corners_unwarp(test_image, 8, 6, mtx, dist)

    if debug == True:
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()

        ax1.set_title('Test Image', fontsize=30)
        ax1.imshow(convert_BGR_to_RGB(test_image))
        ax2.set_title('Calibrated Image', fontsize=30)
        ax2.imshow(convert_BGR_to_RGB(warped))
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)

    return calibration_parameters
```

This function then returns the computed `calibration_parameters` so it can be used later.

To conclude this section of the report, I applied this distortion correction to a test image using the methods mentioned above and figure 1 below illustrates the result I obtained:

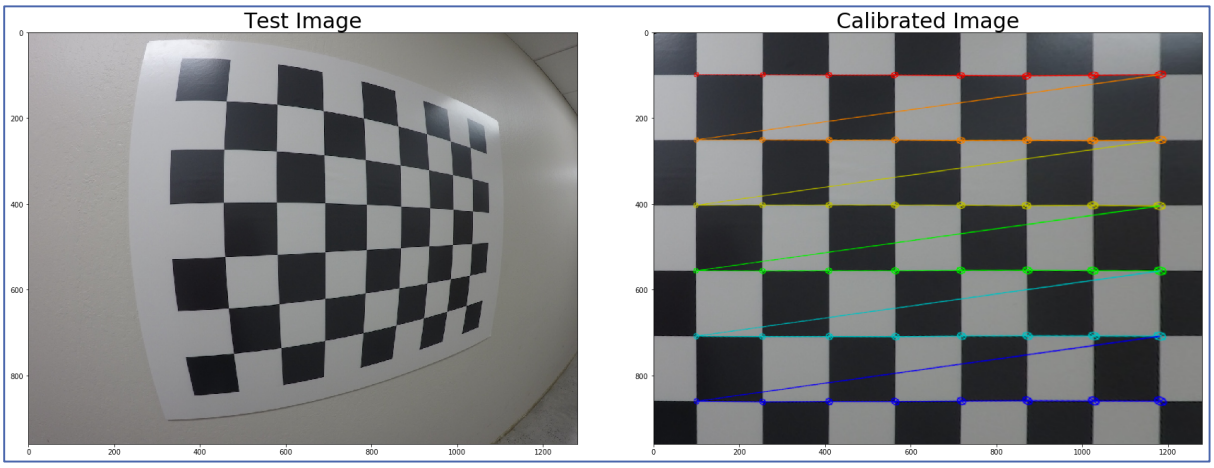

Figure 1: Distortion correction on checkerboard image

Pipeline using a single test image

In this section of the report, I will explain the steps taken to test the pipeline using a single test image of a car driving on the road (lets call it road.png).

Distortion Correction

Since I would have to calibrate the camera according to the new images of the road, I created a function called `undistort_road_image()` which automates the process as illustrated in the figure below.

```
clean_img = None
def undistort_road_image(img, calibration_parameters):
    road_image_calibration_parameters = calibrate_img(img,
                                                      calibration_parameters[0],
                                                      calibration_parameters[1])

    undistorted_road = undistort_img(img,
                                     road_image_calibration_parameters[0],
                                     road_image_calibration_parameters[1])

    global clean_img
    clean_img = undistorted_road

    if debug == True:
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()

        ax1.set_title('Distorted Test Image', fontsize=30)
        ax1.imshow(convert_BGR_to_RGB(img))
        ax2.set_title('Undistorted Test Image', fontsize=30)
        ax2.imshow(convert_BGR_to_RGB(undistorted_road))
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
    return undistorted_road
```

By automate I mean taking road.png as well as the cached `calibration_parameters` and returning an undistorted road.png image (let's call it undistorted_road.png). So, by applying road.png, illustrated in figure 2 to the function above, the image returned is now undistorted (undistorted_road.png) as illustrated in figure 3.

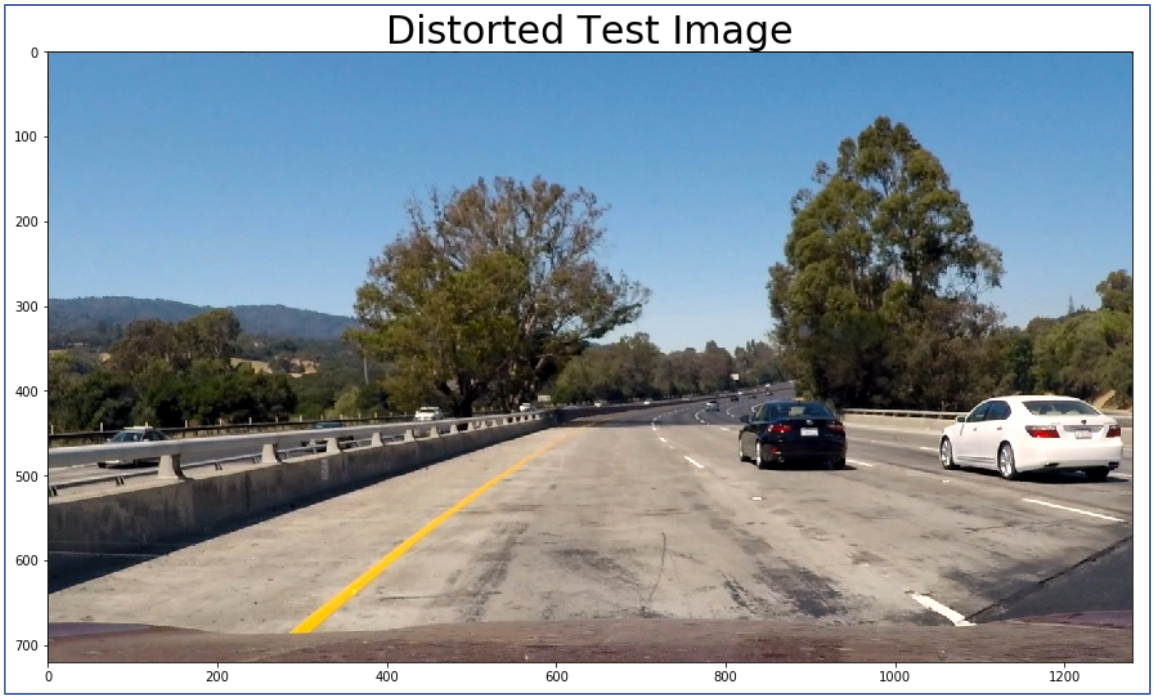

Figure 2: Original road image (before undistortion)

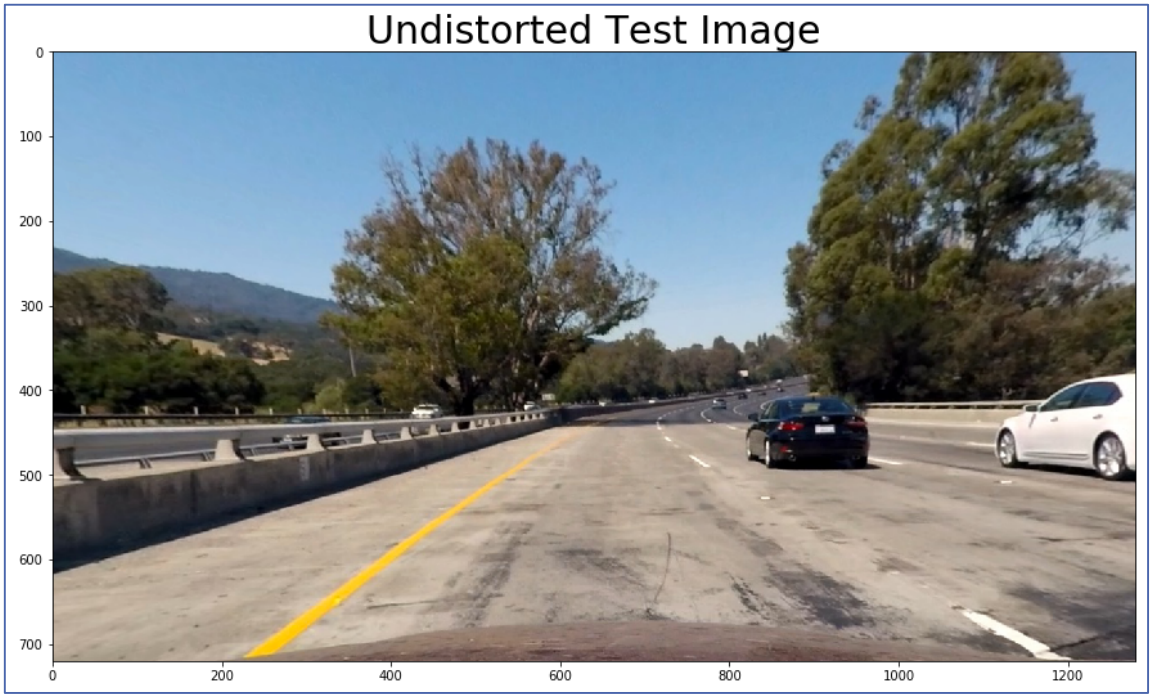

Figure 3: Image after undistortion

Applying Perspective Transform/ Warping

The code for my perspective transform is included in a function called `warp()`. This function takes as input an undistorted image (`img`) which is then passed through this function which returns a warped image of the lane.

```
Minv = None

def warp(img):
    imgg_size = (img.shape[1], img.shape[0])
    img_size = img.shape
    ht_window = np.uint(img_size[0]/1.5)
    hb_window = np.uint(img_size[0])
    c_window = np.uint(img_size[1]/2)

    ctl_window = c_window - .25*np.uint(img_size[1]/2)
    ctr_window = c_window + .25*np.uint(img_size[1]/2)
    cbl_window = c_window - 1*np.uint(img_size[1]/2)
    cbr_window = c_window + 1*np.uint(img_size[1]/2)

    src = np.float32([[cbl_window, hb_window],
                      [cbr_window, hb_window],
                      [ctr_window, ht_window],
                      [ctl_window, ht_window]])

    print("src = " + str(src))

    dst = np.float32([[0,img_size[0]],
                      [img_size[1],img_size[0]],
                      [img_size[1],0],
                      [0,0]])

    print("src = " + str(dst))


    #Get the perspective transform, M
    M = cv2.getPerspectiveTransform(src, dst)
    global Minv
    Minv = cv2.getPerspectiveTransform(dst, src)

    #Create warped image using linear interpolation
    warped = cv2.warpPerspective(img, M, imgg_size, flags=cv2.INTER_LINEAR)

    if (debug == True):
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax1.imshow(convert_BGR_to_RGB(img))
        ax1.scatter(0, 720, s = 80, c = 'r') #top right
        ax1.scatter(1280, 720, s = 80, c = 'r') #bottom right
        ax1.scatter(800, 480, s = 80, c = 'r') #top left
        ax1.scatter(480, 480, s = 80, c = 'r') #bottom left
        ax1.set_title('Original Image', fontsize=30)

        ax2.imshow(convert_BGR_to_RGB(warped))
        ax2.scatter(0, 720, s = 80, c = 'r') #top right
        ax2.scatter(1280, 720, s = 80, c = 'r') #bottom right
        ax2.scatter(1280, 0, s = 80, c = 'r') #top left
        ax2.scatter(0, 0, s = 80, c = 'r') #bottom left
        ax2.set_title('Warped', fontsize=30)
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)

        f, (ax3, ax4) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax3.imshow(convert_BGR_to_RGB(img))
        ax3.set_title('Original Image', fontsize=30)
        ax4.imshow(convert_BGR_to_RGB(warped))
        ax4.set_title('Warped', fontsize=30)
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)

    return warped
```

The source (`src`) and destination (`dst`) points were hardcoded and calculated in the following manner as illustrated in the figure below:

```
img_size = img.shape
ht_window = np.uint(img_size[0]/1.5)
hb_window = np.uint(img_size[0])
c_window = np.uint(img_size[1]/2)

ctl_window = c_window - .25 * np.uint(img_size[1] / 2)
ctr_window = c_window + .25 * np.uint(img_size[1] / 2)
cbl_window = c_window - 1 * np.uint(img_size[1] / 2)
cbr_window = c_window + 1 * np.uint(img_size[1]/2)

src = np.float32([[cbl_window, hb_window],
                  [cbr_window, hb_window],
                  [ctr_window, ht_window],
                  [ctl_window, ht_window]])

print("src = " + str(src))

dst = np.float32([[0,img_size[0]],
                  [img_size[1],img_size[0]],
                  [img_size[1],0],
                  [0,0]])

print("src = " + str(dst))
```

This resulted in the following source (`src`) and destination (`dst`) points as illustrated in Table 1) below:

Table 1: Source and destination points

Source	Destination	Position
0, 720	0, 720	Bottom Left
1280, 720	1280, 720	Bottom Right
800, 480	1280, 720	Top Right
480, 480	0, 0	Top Left

To better illustrate this as well as to verify that my perspective transform function was working accordingly, figures 4 and 5 illustrate the warped image and their corresponding source and destination points on the undistorted and warped images respectively. Table 2 below illustrates the position of the points and their corresponding colours in the images.

Table 2: Colour and position of points

Position	Colour indicator
Bottom Left	Red
Bottom Right	Green
Top Right	Blue
Top Left	Yellow

Figure 4: Undistorted test image with colour referenced source points

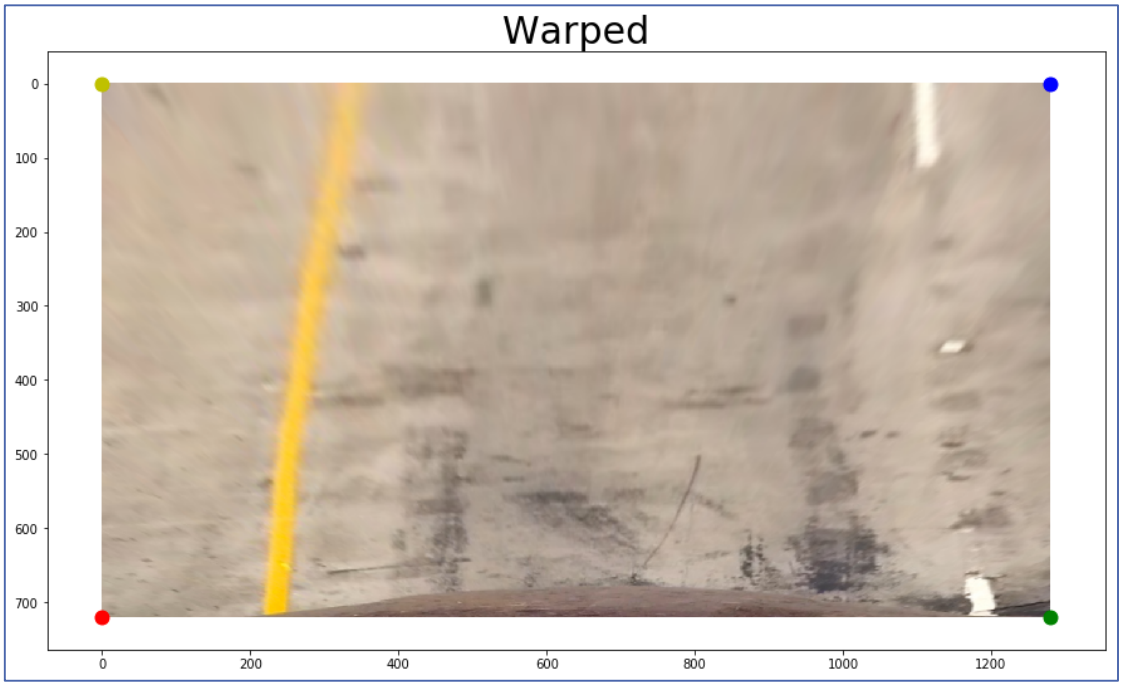

Figure 5: Warped test image with colour referenced destination points

Creating binary images using colour transforms and various other methods

Based on an infinite amount (I'm exaggerating here but its close) of testing, I found that using colour transforms as well as various other methods (let's call this IMAGE PROCESSING) play a VERY important role in this project in order to filter out the lane lines. The main challenge here was to find the perfect combination of image processing sequences such that the lane lines could be seen throughout the entire drive which were affected as what I've observed and faced are three main "obstacles" which are:

1. Change in road colours (I'm assuming the darker section of the road has been recently tarred whereas the lighter one hasn't been) (video frames 581 to approx. 613*)
2. Turns in the road
3. Shadows on the road/ Different lighting conditions (video frames 1030 to 1050*)

*Yes I know the specific frame range. Had to convert the video to frames to analyse if the method chosen were susceptible for all the obstacles.

I've tested multiple colour transforms including, but not limited to the following:

1. Absolute sobel X thresholding
2. Absolute sobel Y thresholding
3. Magnitudinal sobel thresholding
4. Directional sobel thresholding
5. HLS colourspace (specifically the saturation channel)
6. Combinational thresholding

To simplify the process of switching between different thresholding methods, I included all 6 of the above into one function called

```
thresholding_type_select()
```

which allowed me to switch between any one by passing in a string of the name of the thresholding methods. This function takes in 5 inputs, mainly, the thresholding type, the minimum thresholding value, the maximum thresholding value as well as the kernel type and returns a binary image of the resulting thresholding type.

To keep things short, I'll just include the resulting binary images for the method mentioned above here and elaborate more on the one I ended up using.

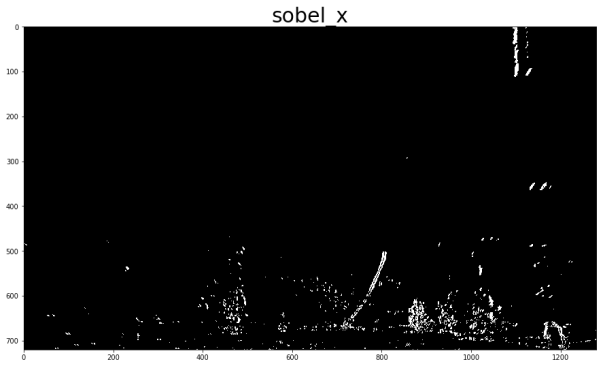

Figure 6: Resulting image using absolute sobel X thresholding

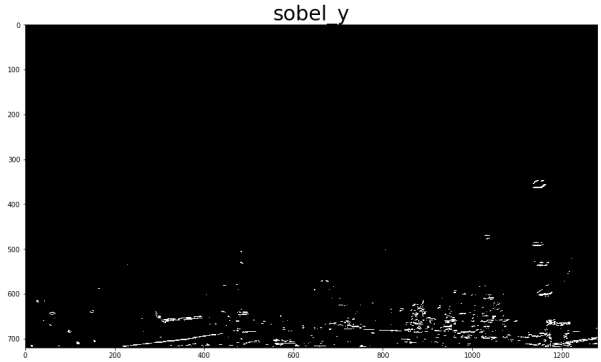

Figure 7: Resulting image using absolute sobel Y thresholding

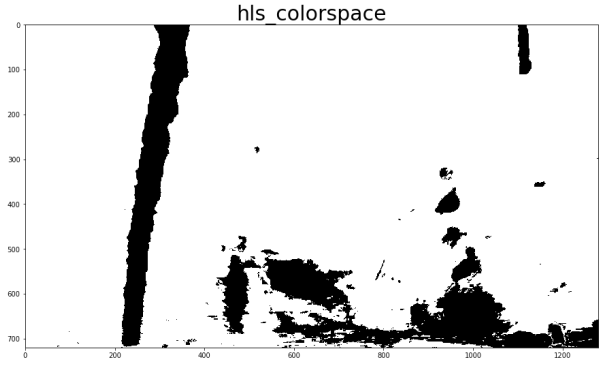

Figure 8: Resulting image using HLS colorspace thresholding

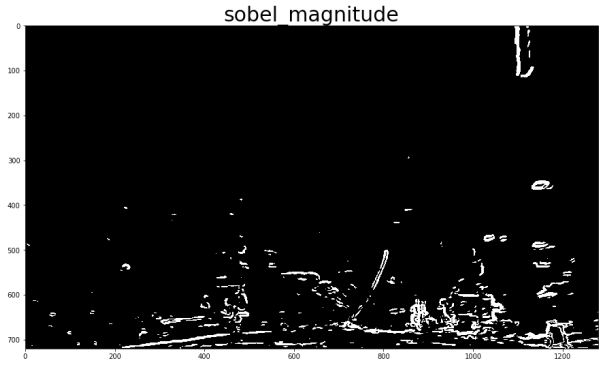

Figure 9: Resulting image using magnitudinal sobel thresholding

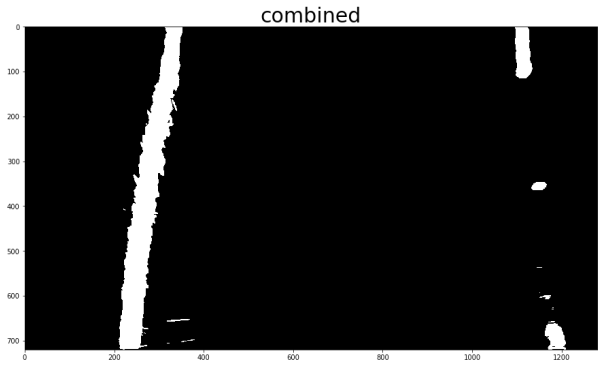

Figure 10: Resulting image using combinational thresholding

As illustrated from figure 6 to 10, its very clear that the combined thresholding method was the obvious one I chose. In this particular circumstance, where the lines for the right lane is separated at a distance, it is very important that the lines stand out for the lanes to be detected. Figure 11 below illustrates the `thresholding_type_select()` function whereas figure 12 below illustrates a snippet of the combined thresholding.

```
warped = None
def thresholding_type_select(thresholding_type, img, thresholding_min, thresholding_max, kernel_size = 5):

    if(thresholding_type == "sobel_x"):
        thresholded_image = abs_sobel_thresh(img,
                                             orient='x',
                                             thresh_min = thresholding_min,
                                             thresh_max = thresholding_max)

    elif(thresholding_type == "sobel_y"):
        thresholded_image = abs_sobel_thresh(img,
                                             orient='y',
                                             thresh_min = thresholding_min,
                                             thresh_max = thresholding_max)

    elif(thresholding_type == "sobel_magnitude"):
        thresholded_image = mag_thresh(img,
                                       sobel_kernel = kernel_size,
                                       mag_thresh = (thresholding_min, thresholding_max))

    elif(thresholding_type == "sobel_directional"):
        thresholded_image = dir_threshold(img,
                                          sobel_kernel = kernel_size,
                                          thresh = (thresholding_min, thresholding_max))

    elif(thresholding_type == "hls_colorspace"):
        thresholded_image = hls_select(img,
                                       thresh = (thresholding_min, thresholding_max))

    elif(thresholding_type == "combined"):
        thresholded_image = combined(img)

    #Preview the thresholded image
    if (debug == True):
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax1.imshow(convert_BGR_to_RGB(img))
        ax1.set_title('Original Image', fontsize=30)
        ax2.imshow(thresholded_image, cmap='gray')
        ax2.set_title(thresholding_type, fontsize=30)
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)

    global warped
    warped = thresholded_image
    return thresholded_image
```

Figure 11

```
def combined(img, color=False, mag_dir_thresh=False):

    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    s_channel = hls[:,:,2]

    lower_white = np.array([0,210,0], dtype=np.uint8)
    upper_white = np.array([255,255,255], dtype=np.uint8)
    white_mask = cv2.inRange(hls, lower_white, upper_white)

    ## Yellow Color
    lower_yellow = np.array([18,0,100], dtype=np.uint8)
    upper_yellow = np.array([30,220,255], dtype=np.uint8)
    yellow_mask = cv2.inRange(hls, lower_yellow, upper_yellow)

    masked_binary = np.zeros_like(white_mask)
    masked_binary[((white_mask == 255) | (yellow_mask == 255))] = 255

    # Grayscale image
    # NOTE: we already saw that standard grayscaling lost color information for the lane lines
    # Explore gradients in other colors spaces / color channels to see what might work better
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    # Take both Sobel x and y gradients
    sobelx = cv2.Sobel(gray[:,:,2], cv2.CV_64F, 1, 0, ksize=15)
    sobely = cv2.Sobel(gray[:,:,2], cv2.CV_64F, 0, 1, ksize=15)
    # Calculate the gradient magnitude
    gradmag = np.sqrt(sobelx**2 + sobely**2)
    # Rescale to 8 bit
    scale_factor = np.max(gradmag)/255
    scaled_sobel = (gradmag/scale_factor).astype(np.uint8)

    # Threshold x gradient
    thresh_min = 30
    thresh_max = 255
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    # Threshold color channel
    s_thresh_min = 150
    s_thresh_max = 255
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh_min) & (s_channel <= s_thresh_max)] = 1

    # Stack each channel to view their individual contributions in green and blue respectively
    # This returns a stack of the two binary images, whose components you can see as different colors
    color_binary = np.dstack(( np.zeros_like(sxbinary), sxbinary, s_binary))

    # Combine the two binary thresholds
    combined_binary = np.zeros_like(sxbinary)
    combined_binary[(s_binary == 1) | (sxbinary == 1)] = 1
    combined_binary[(combined_binary == 255) | (masked_binary == 255)] = 1
    return combined_binary
```

Figure 12

The combined thresholding method I used are listed in the steps below:

1. The image is first converted to HLS colourspace and a colour mask is applied to the yellow and white lines in which the output of this is identified as `masked_binary` in figure 12. As mentioned in the lessons, the HLS colourspace is chosen because it is found to be the most robust.
2. Then Sobel filters were then chosen, specifically magnitudinal sobel based on the lessons for edge detection
3. Apply thresholding to the x gradient on (2) of which the output is identified as `sxbinary` in figure 12.
4. Apply thresholding on the saturation channel of which the output is identified as `s_binary` in figure 12.
5. Combine all the binaries to produce the final binary output identified as `combined_binary` in figure 12.

Lane Detection

Now that thresholding has been done and a good road binary image has been obtained. I then proceeded to detect the lane. The method I used to do so is particularly by using the peaks in the histogram as well as the sliding window technique. To decide explicitly which pixels are part of the lines and which belongs to the left and the right lane, I first take a histogram along the columns in the lower half of the binary image of the road.png (figure 13) using the following lines of code in the function called `start_lane_detection():`

```
histogram = np.sum(img[img.shape[0]/2:,:], axis=0)
```

and plotted the histogram using the following line of code:

```
plt.plot(histogram)
```

which resulted in the following:

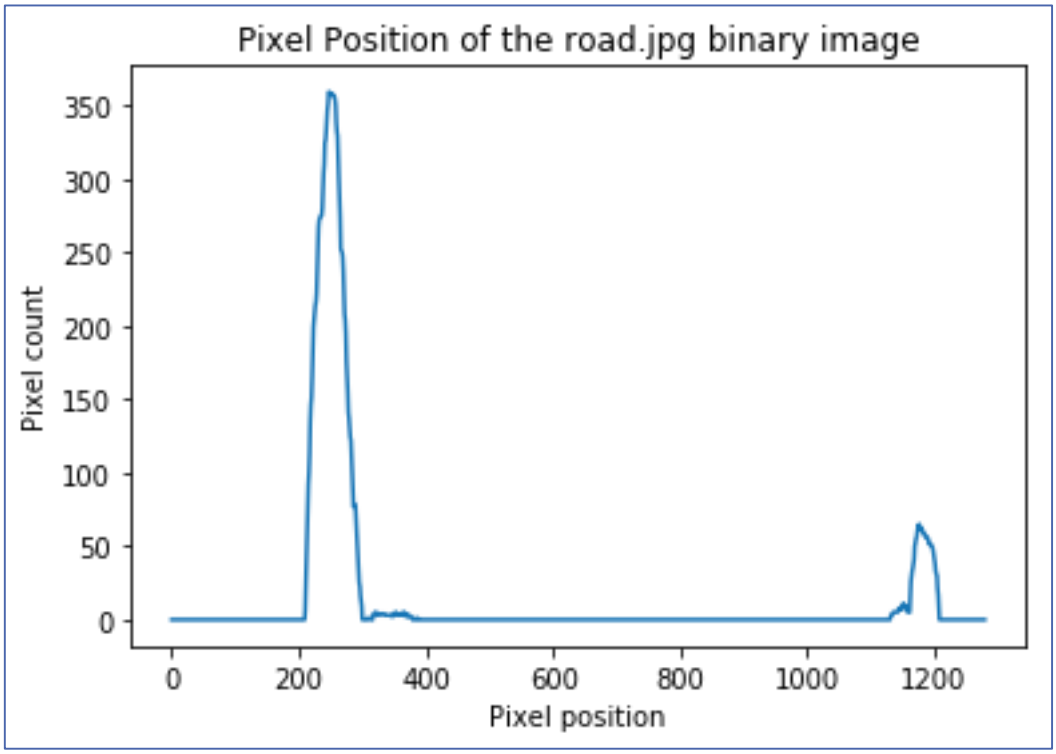

Figure 13

Using the histogram, the pixel values along each column in the image are added and since the pixels in the resulting road.png binary image is either 0 or 1, the two most prominent peaks in the histogram are used to determine the x-position of the base of the lane lines which can be used as a starting point for where to search for the left and right lane lines. The sliding window are then placed around the line centres which are then used to search and "slide" up to the top of the frame.

I then used a 2nd order polynomial regression to obtain the best fit to approximate the lane lines. To improve on this, however, I applied a correctional step to calculate the mean squared error of the of lane lines to make the lane lines more robust especially in areas of where lane lines are affected by shadows. After this step, the windows are then plotted back into the warped image to visualise if they were identified correctly as illustrated in figure 14 below:

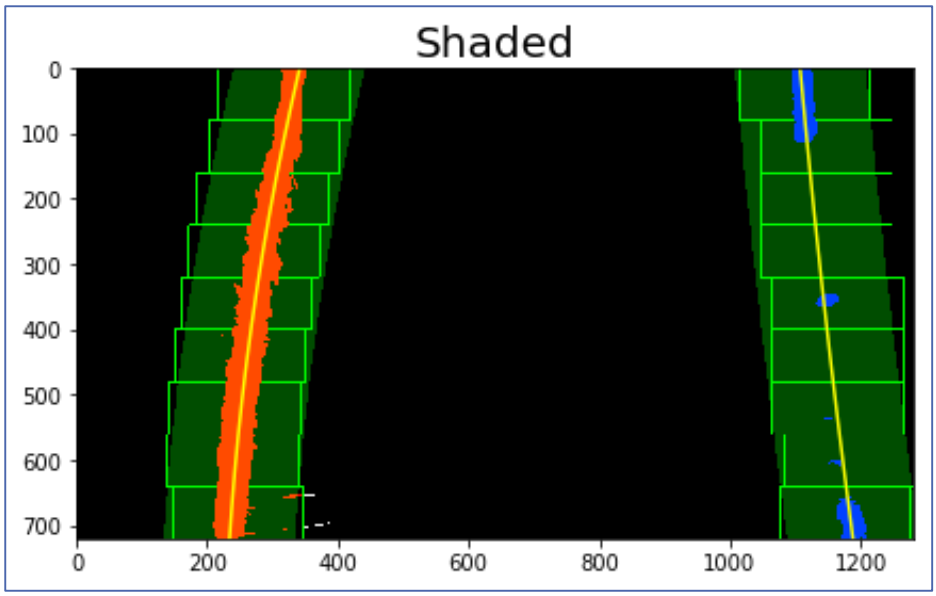

Figure 14

Since the position of the lines are now known (left lane line is highlighted as red and the right lane line is highlighted in blue), a search in a margin of 100 is conducted around the previous line position. The lane lines are then drawn on top of the shaded image above and is illustrated in figure 15 below:

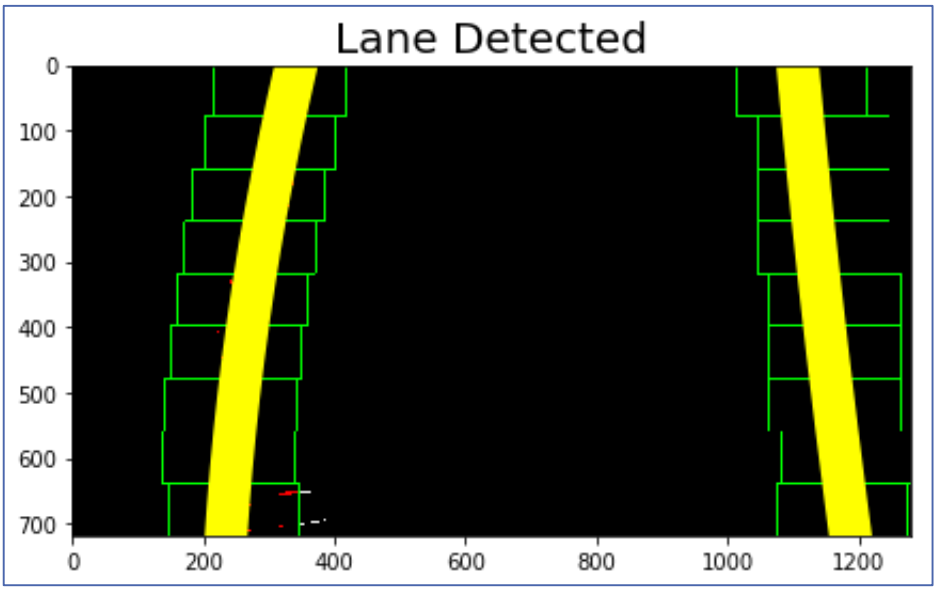

Figure 15

Radius of the curvature

The radius of the curvature is calculated using the following formula illustrated in figure 16 below, where x_{real} and y_{real} are are real world coordinates whereas x_{pix} and y_{pix} is the coordinate in pixel space as explained in the "Measuring Curvature" section of the lesson:

$$x_{real} = M_x x_{pix}$$

$$y_{real} = M_y y_{pix}$$

$$x_{pix} = Ay_{pix}^2 + By_{pix} + C$$

$$\frac{dx_{real}}{dy_{real}} = \frac{M_x}{M_y}[2Ay_{pix} + By_{pix}]$$

$$\frac{d^2x_{real}}{dy_{real}^2} = \frac{M_x}{M_y^2}[2A]$$

$$Radius of Curvature = \frac{(1 + \frac{dx_{real}}{dy_{real}}^2)^{1.5}}{|\frac{d^2x_{real}}{dy_{real}^2}|}$$

Figure 16

Using this formula, which is described in the lessons shown below:

```
# Fit new polynomials to x,y in world space
left_fit_cr = np.polyfit(ploty*ym_per_pix, leftx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, rightx*xm_per_pix, 2)
# Calculate the new radii of curvature
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_pix + right_fit_cr[1])**
2)**1.5) / np.absolute(2*right_fit_cr[0])
# Now our radius of curvature is in meters
print(left_curverad, 'm', right_curverad, 'm')
# Example values: 632.1 m    626.2 m
```

And is implemented in the `start_lane_detection()` function, specifically:

```
def calculate_curvature(thresholded_image, left_lane_inds, right_lane_inds):
    # Define conversions in x and y from pixels space to meters
    ym_per_pix = 3.048/100 # meters per pixel in y dimension, lane line is 10 ft = 3.048 meters
    xm_per_pix = 3.7/719.0 # meters per pixel in x dimension, lane width is 12 ft = 3.7 meters
    left_curverad, right_curverad, center_dist = (0, 0, 0)

    # Define y-value where we want radius of curvature
    # I'll choose the maximum y-value, corresponding to the bottom of the image
    thresholded_img_shape = thresholded_image.shape[0]
    ploty = np.linspace(0, thresholded_img_shape - 1, thresholded_img_shape)
    y_eval = np.max(ploty)

    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = thresholded_image.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    # 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]

    if len(leftx) != 0 and len(rightx) != 0:

        # Fit new polynomials to x, y in world space

        left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 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_
        right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_

        # Now our radius of curvature is in meters

    return left_curverad, right_curverad
```

Using this section of the function, the left lane curvature for this road.png test image is calculated to be **772.21 m** and the right lane curvature is calculated to be **3045.16 m**

Besides that, the positon of the vehicle with respect to the centre of the lane is calculated using the line of code below:

```
x_left_pix = left_fit[0]*(y_eval**2) + left_fit[1]*y_eval + left_fit[2]
x_right_pix = right_fit[0]*(y_eval**2) + right_fit[1]*y_eval + right_fit[2]
position_from_center = ((x_left_pix + x_right_pix)/2 - midx) * xm_per_pix

if position_from_center < 0:
    text = 'to the left'
else:
    text = 'to the right'
cv2.putText(result,'Distance From Center: %.2fm %s' % (np.absolute(position_from_center), text),(20,120)
```

Similarly, using this function, the position of the vehicle for this road.png test image is calculated to be **0.31 m to the right.**

Pipeline using the project video

I have multiple videos uploaded on my channel showing the progress I have gone throughout this project.

Of course, these aren't the only videos I have generated and are only the ones worth noting.

My "first" failed attempt can be found here:

https://www.youtube.com/watch?v=IhQBMwQTsgk

My “second” failed attempt can be found here:

https://www.youtube.com/watch?v=xC1qM0HxkFY

My “third” failed attempt can be found here:

https://www.youtube.com/watch?v=ghcm67IBXK8

My “fourth” failed attempt can be found here:

https://www.youtube.com/watch?v=IsoIgfPBQmM

My “fifth” failed attempt can be found here (this is where I came really close):

https://www.youtube.com/watch?v=PgDKz1fpr3Q

My “sixth” failed attempt can be found here (this is where I came really close):

https://www.youtube.com/watch?v=XblyHcmCkRk

My final successful attempt can be found here (sorted out the issue from the previous video using the mean squared error correction on the polynomial all thanks to the folks down in the Facebook group as well as my mentor!):

https://www.youtube.com/watch?v=161gDwYKtqQ

Discussion and Thoughts

I’ve been told that this project was relatively “easy”, apparently not. Based on the time spent working on this, I think that the project would most likely to fail in areas where light conditions changes rapidly, especially during thunderstorms or surge of lightning. In additions to that, I would think that this lane finding technique is only suitable for roads where there are of course, lanes. There are many roads known as “unmarked” lanes which diverges from laned roads. In this case, this whole technique will fail miserably!