Advanced Lane Finding using OpenCV

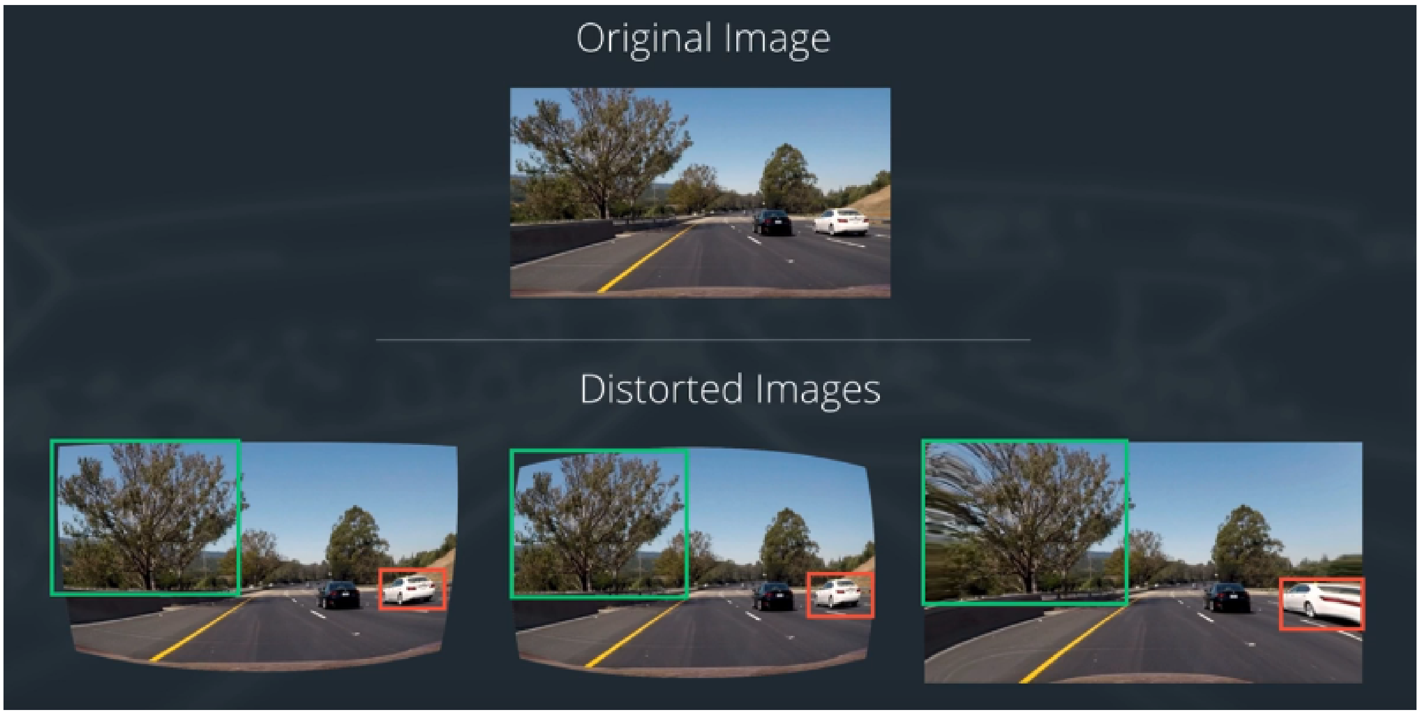
In this project, our goal is to write a software pipeline to identify the lane boundaries in a video from a front-facing camera on a car. The camera calibration images, test road images, and project videos are available here repository.

The goals / steps 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 matrix and distortion coefficients

Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; this transformation isn’t perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that you can get correct and useful information out of them.



Real cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses. This creates an effect that distorts the edges of images, so that lines or objects appear more or less curved than they actually are. This is called radial distortion, and it’s the most common type of distortion.

Another type of distortion, is tangential distortion. This occurs when a camera’s lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is. This makes an image look tilted so that some objects appear farther away or closer than they actually are.

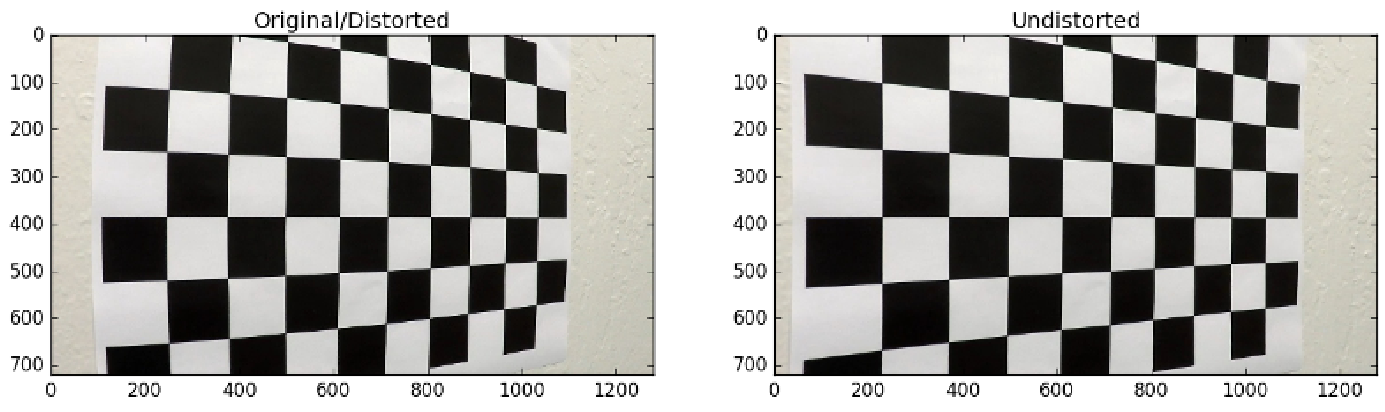
There are three coefficients needed to correct for radial distortion: k1, k2, and k3, and 2 for tangential distortion: p1, p2. In this project the camera calibration is implemented using OpenCV and a chessboard panel with 9x6 corners.

```
import os, glob, pickle
import numpy as np
import cv2
import matplotlib.pyplot as plt

class CameraCalibrator:
    '''Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.'''

    def __init__(self, image_directory, image_filename, binary_filename, nx, ny):
        print ('Initializing CameraCalibrator ...')
        self.__image_directory = image_directory
        self.__image_filename = image_filename
        self.__binary_filename = binary_filename
        self.__nx = nx # the number of inside corners in x
        self.__ny = ny # the number of inside corners in y
        self.mtx = None
        self.dist = None
        self.rvecs = None
        self.tvecs = None
        self.__calibrated = False

    def __calibrate(self):

        # Read in and make a list of calibration images
        calibration_filenames = glob.glob(self.__image_directory+'/'+self.__image_filename)

        # Arrays to store object points and image points from all the images
        object_points = [] # 3D points in real world space
        image_points = [] # 2D points in image plane

        # Prepare object points, like (0,0,0), (1,0,0), (2,0,0), ...,(7,5,0)
        object_p = np.zeros((self.__ny*self.__nx,3),np.float32)
        object_p[:,:2] = np.mgrid[0:self.__nx,0:self.__ny].T.reshape(-1,2) # s,y coordinates

        # Extract the shape of any image
        image = cv2.imread(calibration_filenames[1])
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        shape = gray.shape[::-1] # (width,height)

        # Process each calibration image
        for image_filename in calibration_filenames:
            # Read in each image
            image = cv2.imread(image_filename)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # RGB is standard in matlibplot

            # Convert to grayscale
            gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

            # Find the chessboard corners
            ret, corners = cv2.findChessboardCorners(gray, (self.__nx, self.__ny), None)

            # If found, draw corners
            if ret == True:
                # Store the corners found in the current image
                object_points.append(object_p) # how it should look like
                image_points.append(corners) # how it looks like

                # Draw and display the corners
                cv2.drawChessboardCorners(image, (self.__nx, self.__ny), corners, ret)
                plt.figure()
                plt.imshow(image)
                plt.show()

        # Do the calibration
        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(object_points, image_points, shape, None, None)
        #print(ret, mtx, dist, rvecs, tvecs)

        # Pickle to save time for subsequent runs
        binary = {}
        binary["mtx"] = mtx
        binary["dist"] = dist
        binary["rvecs"] = rvecs
        binary["tvecs"] = tvecs
        pickle.dump(binary, open(self.__image_directory + '/' + self.__binary_filename, "wb"))

        self.mtx = mtx
        self.dist = dist
        self.rvecs = rvecs
        self.tvecs = tvecs
        self.__calibrated = True

    def __load_binary(self):
        '''Load previously computed calibration binary data'''
        with open(self.__image_directory + '/' + self.__binary_filename, mode='rb') as f:
            binary = pickle.load(f)
        self.mtx = binary['mtx']
        self.dist = binary['dist']
        self.rvecs = binary['rvecs']
        self.tvecs = binary['tvecs']
        self.__calibrated = True

    def get_data(self):
        '''Getter for the calibration data. At the first call it gerenates it.'''
        if os.path.isfile(self.__image_directory + '/' + self.__binary_filename):
            self.__load_binary()
        else:
            self.__calibrate()
        return self.mtx, self.dist, self.rvecs, self.tvecs

    def undistort(self, image):
        if  self.__calibrated == False:
            self.get_data()
        return cv2.undistort(image, self.mtx, self.dist, None, self.mtx)

    def test_undistort(self, image_filename, plot=False):
        '''A method to test the undistort and to plot its result.'''
        image = cv2.imread(image_filename)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # RGB is standard in matlibplot
        image_undist = self.undistort(image)

        # Ploting both images Original and Undistorted
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
        ax1.set_title('Original/Distorted')
        ax1.imshow(image)
        ax2.set_title('Undistorted')
        ax2.imshow(image_undist)
        plt.show()
```

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, objp is just a replicated array of coordinates, and objpoints will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. imgpoints will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output objpoints and imgpoints to compute the camera calibration and distortion coefficients using the OpenCV cv2.calibrateCamera() function. I applied this distortion correction to the test image using the cv2.undistort() function and obtained this result:

The code for this step is contained in the file "./camera_calibration.py". Applying this step to a sample image, you'll get a result like this:

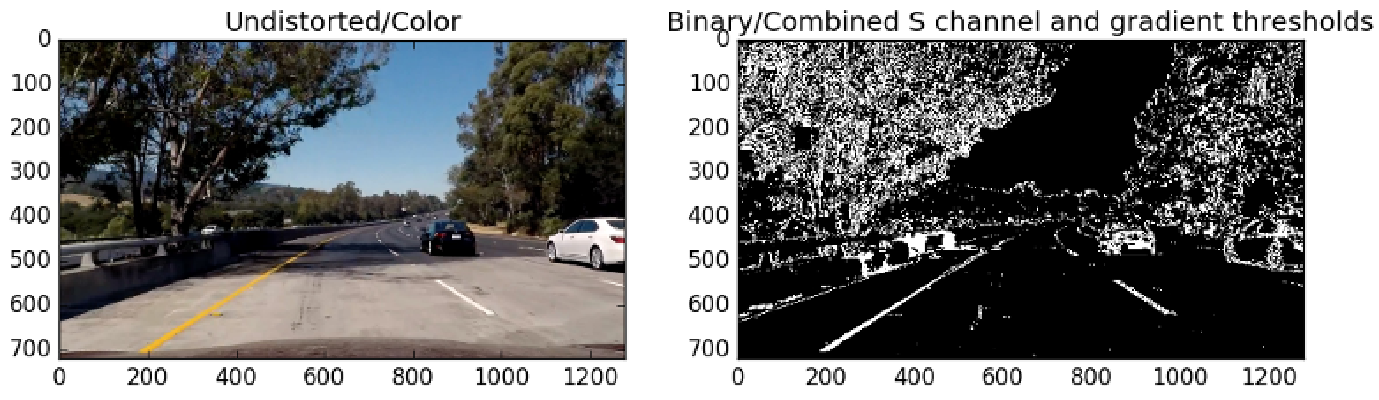


Use color transforms, gradients, etc., to create a thresholded binary image.

I used a combination of color and gradient thresholds to generate a binary image as implemented in the method compute_binary_image() which can be found in lane_detection.py. There are various combinations of color and gradient thresholds to generate a binary image where the lane lines are clearly visible.

```
def compute_binary_image(self, color_image, plot=False):
        # Convert to HLS color space and separate the S channel
        # Note: img is the undistorted image
        hls = cv2.cvtColor(color_image, cv2.COLOR_RGB2HLS)
        s_channel = hls[:,:,2]

        # 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(color_image, cv2.COLOR_RGB2GRAY)

        # Sobel x
        sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0) # Take the derivative in x
        abs_sobelx = np.absolute(sobelx) # Absolute x derivative to accentuate lines away from horizontal
        scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))

        # Threshold x gradient
        thresh_min = 20
        thresh_max = 100
        sxbinary = np.zeros_like(scaled_sobel)
        sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

        # Threshold color channel
        s_thresh_min = 170
        s_thresh_max = 255
        s_binary = np.zeros_like(s_channel)
        s_binary[(s_channel >= s_thresh_min) & (s_channel <= s_thresh_max)] = 1

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

        if (plot):
            # Ploting both images Original and Binary
            f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
            ax1.set_title('Undistorted/Color')
            ax1.imshow(color_image)
            ax2.set_title('Binary/Combined S channel and gradient thresholds')
            ax2.imshow(combined_binary, cmap='gray')
            plt.show()

        return combined_binary
```

Here's an example of my output for this step.



Apply a perspective transform to rectify binary image ("birds-eye view").

Next, we want to identify four source points for the perspective transform. In this case, we can assume the road is a flat plane. This isn't strictly true, but it can serve as an approximation for this project. We would like to pick four points in a trapezoidal shape (similar to region masking) that would represent a rectangle when looking down on the road from above.

The easiest way to do this is to investigate an image where the lane lines are straight, and find four points lying along the lines that, after perspective transform, make the lines look straight and vertical from a bird's eye view perspective.

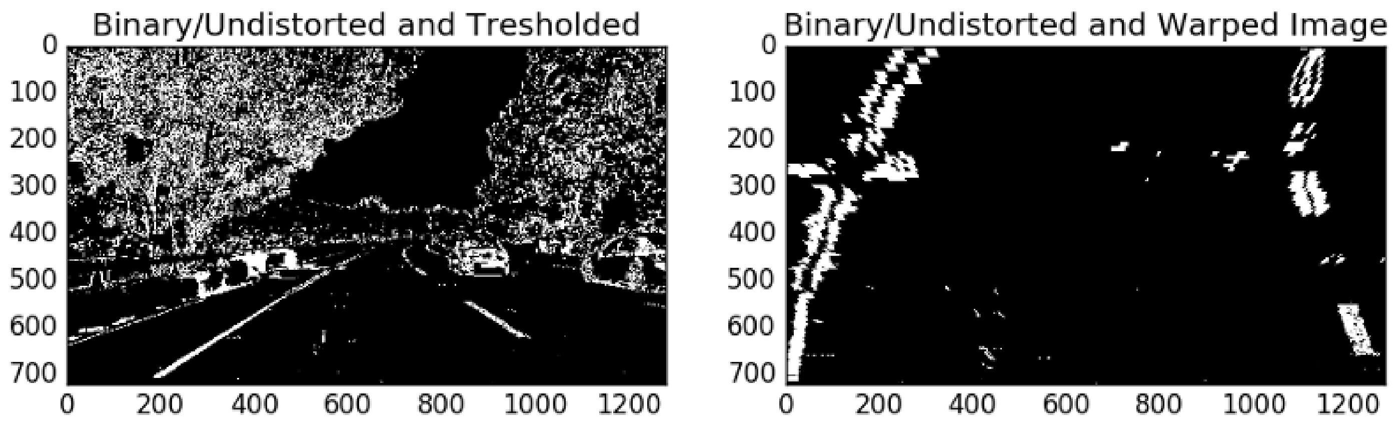
The code for my perspective transform includes 2 functions called `compute_perspective_transform()` and `apply_perspective_transform()`, which appear in the file `lane_detection.py`. The `compute_perspective_transform()` builds a transformation matrix M used by `apply_perspective_transform()` to transform a binary image.

```
def compute_perspective_transform(self, binary_image):
        # Define 4 source and 4 destination points = np.float32([[,],[,],[,],[,]])
        shape = binary_image.shape[::-1] # (width,height)
        w = shape[0]
        h = shape[1]
        transform_src = np.float32([ [580,450], [160,h], [1150,h], [740,450]])
        transform_dst = np.float32([ [0,0], [0,h], [w,h], [w,0]])
        M = cv2.getPerspectiveTransform(transform_src, transform_dst)
        return M

def apply_perspective_transform(self, binary_image, M, plot=False):
        warped_image = cv2.warpPerspective(binary_image, M, (binary_image.shape[1], binary_image.shape[0]),
flags=cv2.INTER_NEAREST)  # keep same size as input image
        if(plot):
            # Ploting both images Binary and Warped
            f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
            ax1.set_title('Binary/Undistorted and Tresholded')
            ax1.imshow(binary_image, cmap='gray')
            ax2.set_title('Binary/Undistorted and Warped Image')
            ax2.imshow(warped_image, cmap='gray')
            plt.show()

        return warped_image
```

I chose the hardcode the source and destination points in the following manner:

```
transform_src = np.float32([ [580,450], [160,h], [1150,h], [740,450]])
transform_dst = np.float32([ [0,0], [0,h], [w,h], [w,0]])
```

The output of these 2 methods looks like this:



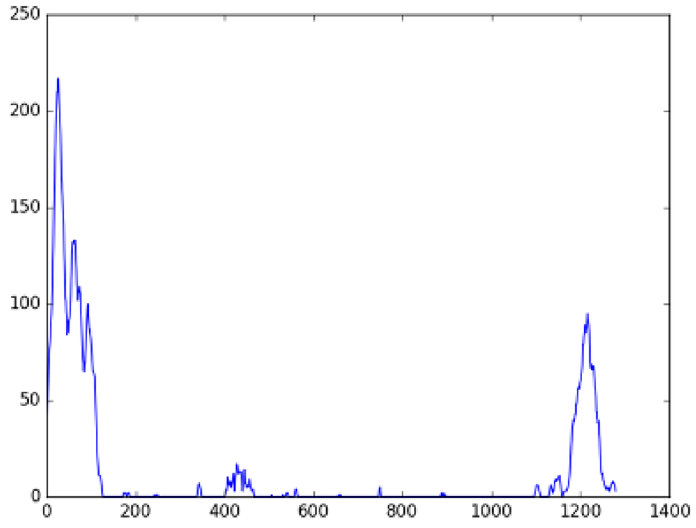
Detect lane pixels and fit to find the lane boundary.

We now have a thresholded warped image and we're ready to map out the lane lines! There are many ways we could go about this, but using peaks in the histogram works well.

After applying calibration, thresholding, and a perspective transform to a road image, we should have a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

I first take a histogram along all the columns in the lower half of the image like this:

```
import numpy as np
histogram = np.sum(img[img.shape[0]/2:,:], axis=0)
plt.plot(histogram)
```

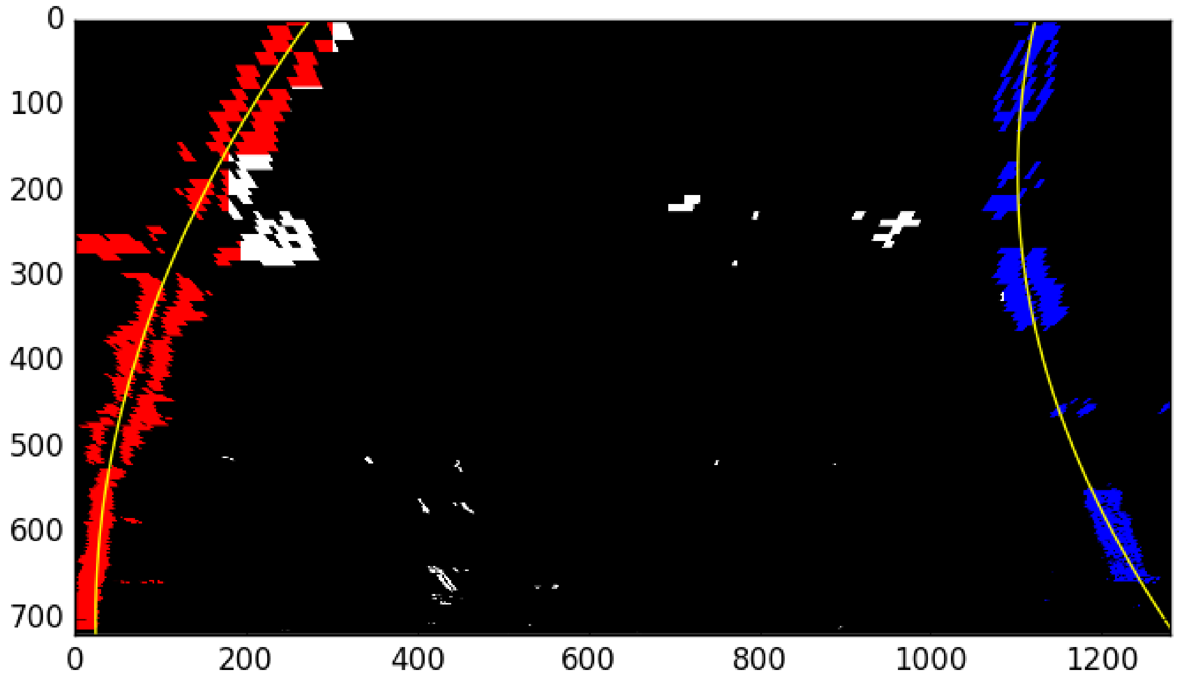


With this histogram I am adding up the pixel values along each column in the image. In my thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. I can use that as a starting point for where to search for the lines. From that point, I can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

```
def extract_lanes_pixels(self, 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)
        #plt.show()
        # Create an output image to draw on and  visualize the result
        out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
        # Find the peak of the left and right halves of the histogram
        # 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 height of windows
        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 for each window
        leftx_current = leftx_base
        rightx_current = rightx_base
        # Set the width of the windows +/- margin
        margin = 100
        # Set minimum number of pixels found to recenter window
        minpix = 50
        # 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_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 you found > minpix pixels, recenter next window on 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_right_inds]))

        # Concatenate the arrays of indices
        left_lane_inds = np.concatenate(left_lane_inds)
        right_lane_inds = np.concatenate(right_lane_inds)

        # 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, left_lane_inds, right_lane_inds

def poly_fit(self, leftx, lefty, rightx, righty, left_lane_inds, right_lane_inds, binary_warped, plot:False):

        # Fit a second order polynomial to each
        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] )
        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]

        # 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])
        out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
        out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
        out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

        if(plot):
            plt.imshow(out_img)
            plt.plot(left_fitx, ploty, color='yellow')
            plt.plot(right_fitx, ploty, color='yellow')
            plt.xlim(0, 1280)
            plt.ylim(720, 0)
            plt.show()

        return left_fit, right_fit, ploty, left_fitx, right_fitx
```

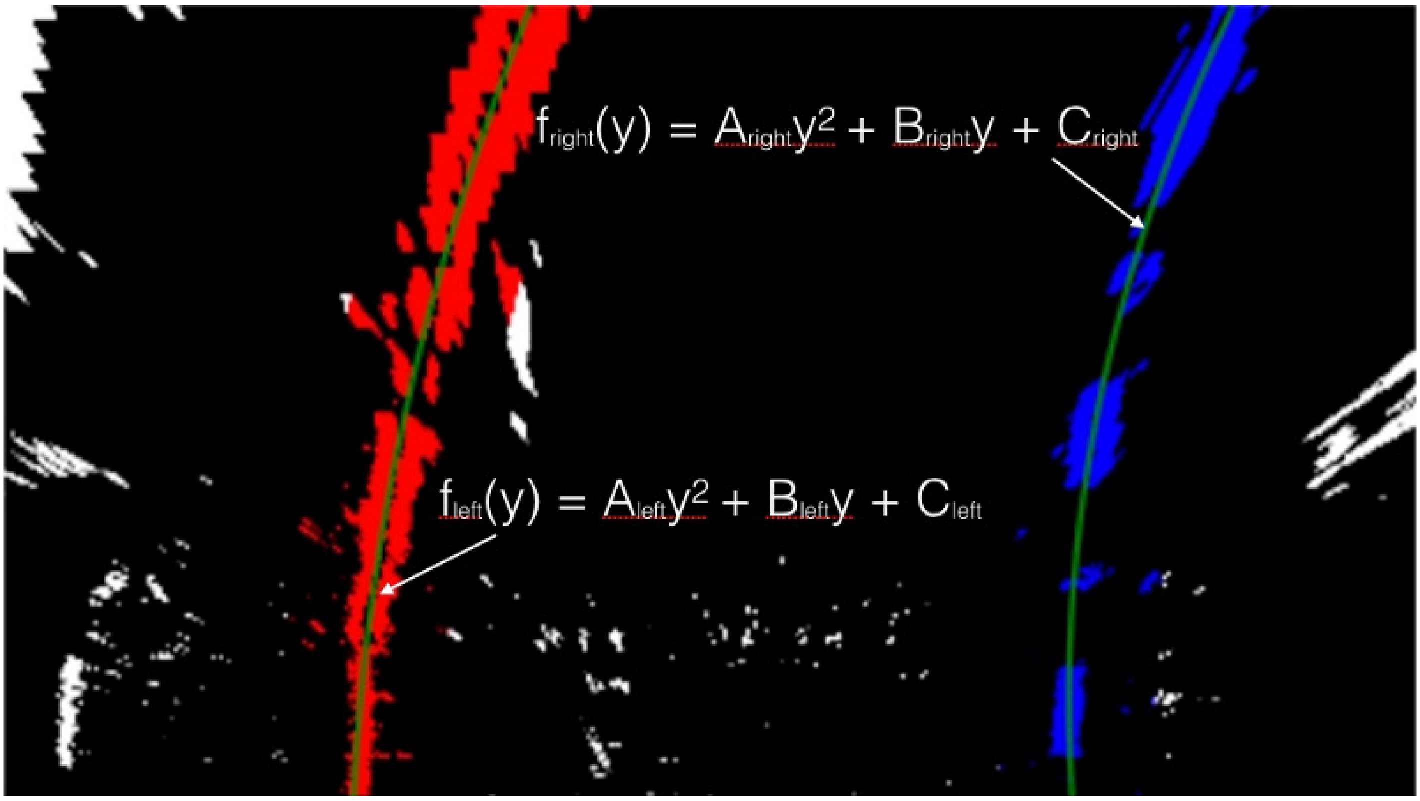
The output should look something like this:



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

Self-driving cars need to be told the correct steering angle to turn, left or right. We can calculate this angle if we know a few things about the speed and dynamics of the car and how much the lane is curving. One way to calculate the curvature of a lane line, is to fit a 2nd degree polynomial to that line, and from this we can easily extract useful information.

For a lane line that is close to vertical, we can fit a line using this formula: f(y) = Ay^2 + By + C, where A, B, and C are coefficients. A gives us the curvature of the lane line, B gives us the heading or direction that the line is pointing, and C gives us the position of the line based on how far away it is from the very left of an image (y = 0).

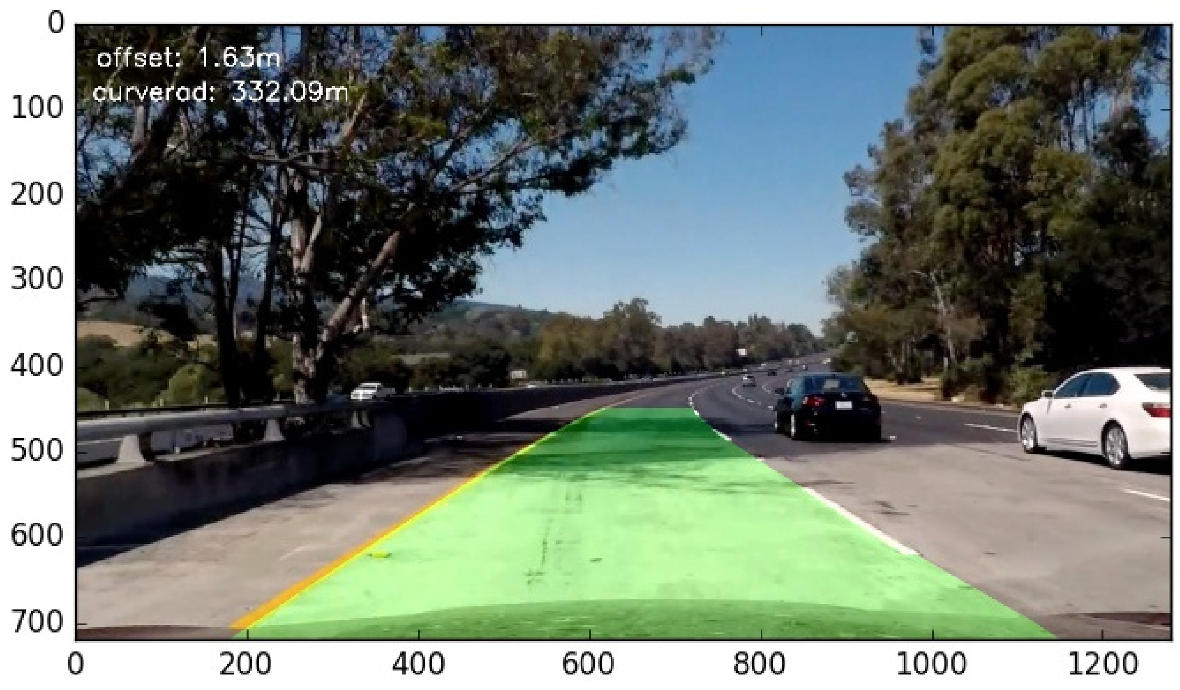

$f_{right}(y) = A_{right}y^2 + B_{right}y + C_{right}$
$f_{left}(y) = A_{left}y^2 + B_{left}y + C_{left}$

Implementation:

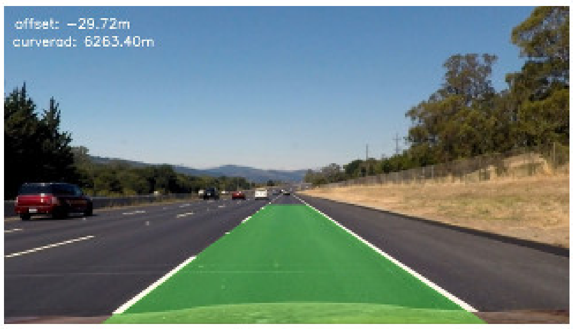
```
def compute_curvature(self, left_fit, right_fit, ploty, left_fitx, right_fitx, leftx, lefty, rightx, righty):

        # Define conversions in x and y from pixels space to meters
        ym_per_pix = 30/720 # meters per pixel in y dimension
        xm_per_pix = 3.7/700 # meters per pixel in x dimension
        y_eval = np.max(ploty)

        fit_cr_left = np.polyfit(ploty * ym_per_pix, left_fitx * xm_per_pix, 2)
        curverad_left = ((1 + (2 * left_fit[0] * y_eval / 2. + fit_cr_left[1]) ** 2) ** 1.5) / np.absolute(2 *
fit_cr_left[0])
        fit_cr_right = np.polyfit(ploty * ym_per_pix, right_fitx * xm_per_pix, 2)
        curverad_right = ((1 + (2 * left_fit[0] * y_eval / 2. + fit_cr_right[1]) ** 2) ** 1.5) / np.absolute(2 *
fit_cr_right[0])

        return (curverad_left + curverad_right) / 2
```

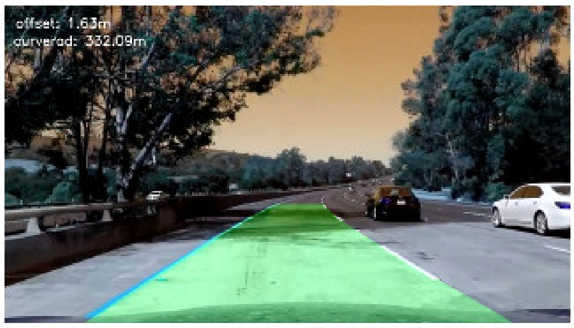
Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

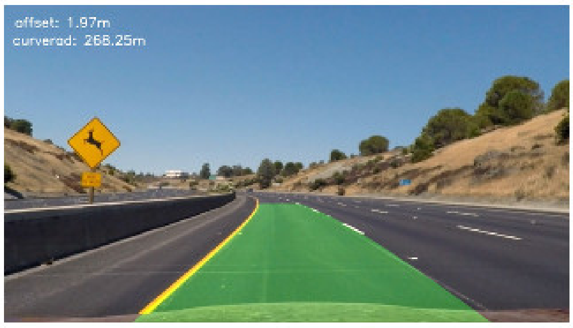
Function render_curvature_and_offset in lane_detection.py is used to warp the detected lane lines back onto the original image and plot the detected lane using a filled polygon. It also plots the curvature and position in the top left corner and at bottom of the image or video frame.

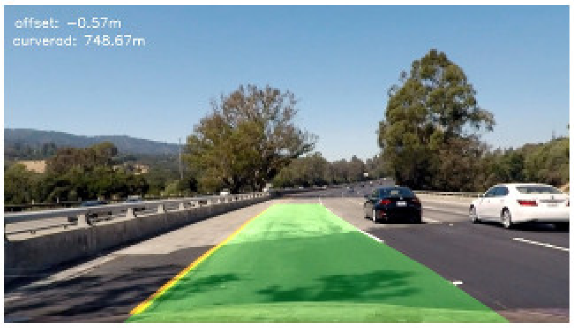


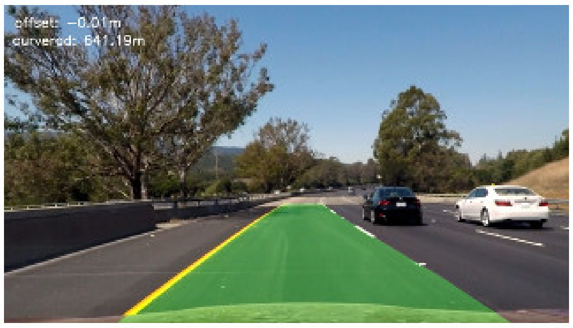
Results for all six test images:

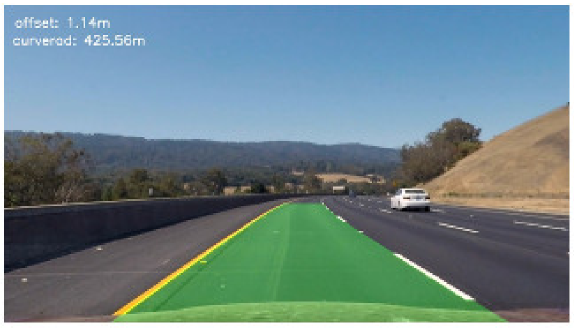












Video pipeline

The same pipeline is applied to videos. The pipeline is implemented in p4.py in the 'process_video' method.

```
def process_video(video_filename, lane_detector, plot=False):
    video_input = VideoFileClip(video_filename + ".mp4")
    video_output = video_input.fl_image(process_image)
    video_output.write_videofile(video_filename + "_output.mp4", audio=False)
```

Here is the link to the project video result.

Here's a link to my video result

Problems / issues faced in the implementation of this project

The main issue associated with this project was the allocated time, the estimated 10 hours was far to little. As a result a number of improvements have not been implemented:

- The class Line.py to encapsulates a single lane line to get a more robust detection by averaging the results computed in each image.
- The challenge videos. The pipeline has not been applied to the other two challenge videos.