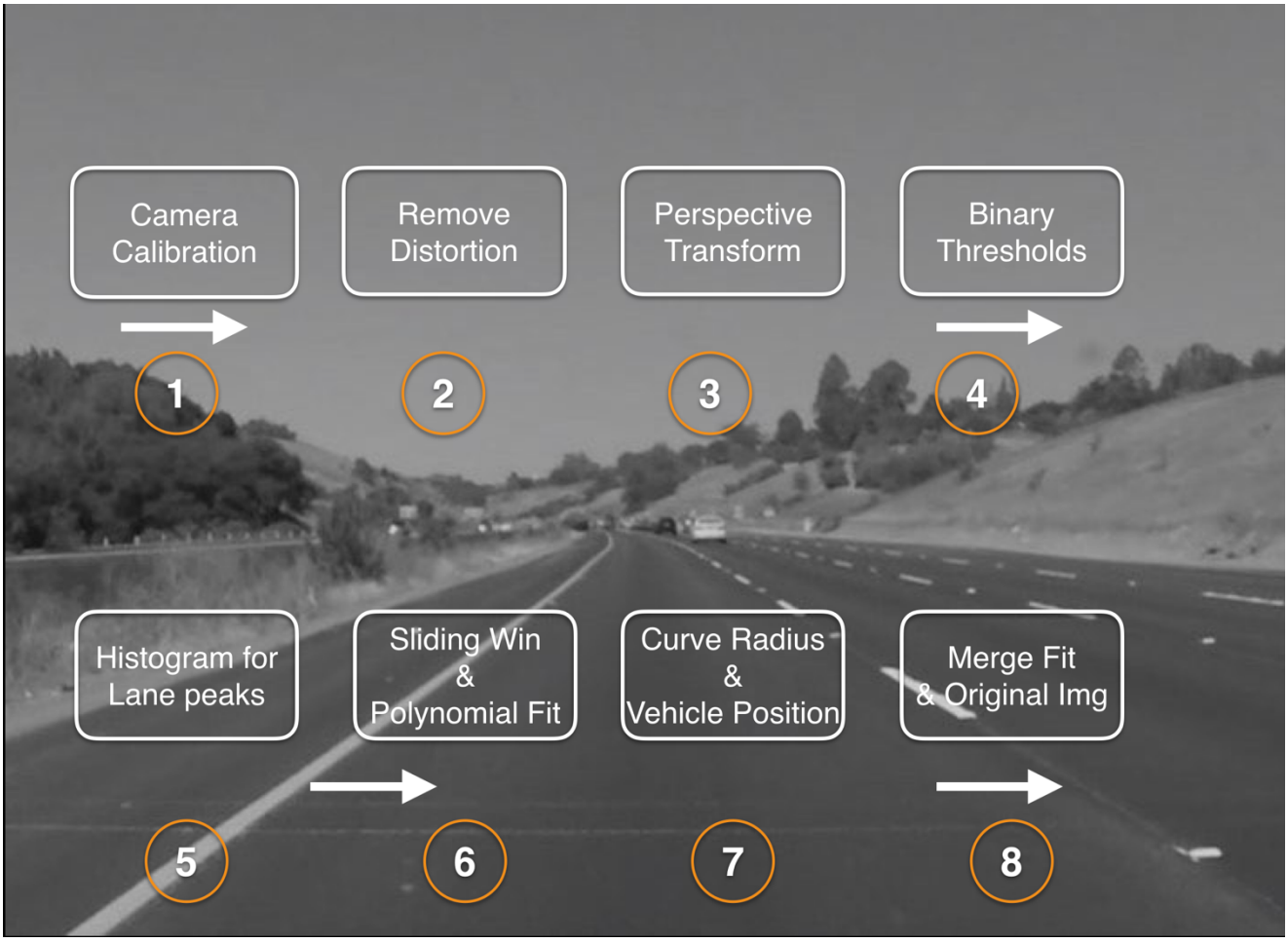
Advance Lane Detection using Python & OpenCV Project 2

The goal of this project is to develop a pipeline to process a video stream from a forward-facing camera mounted on the front of a car, and output an annotated video which identifies:

- The positions of the lane lines
- The location of the vehicle relative to the center of the lane
- The radius of curvature of the road

Pipeline architecture:



1. Calibrate Camera
2. Remove Distortion using the calibration matrix
3. Perform Perspective Transform
4. Extract Channel & Apply Binary Thresholds
5. Find Lane peaks using Histogram

6. Search Lanes using sliding window
7. Find radius of the curve & Vehicle's position from center of the lane
8. Merge Lane lines and ploy fit & Original images
9. Repeat step 1 to 8 for each frame on the video streams.

Prework:

I wrote a function called `display_img_grid()` that shows all the test images we're working on using matplotlib as a 3 rows 2 columns grid.

```
def  display_img_grid(imgs, cols = 2, rows =3 , cmap=None):
    """
    Display a list of images in a single figure with matplotlib.
        Parameters:
            imgs: List of np.arrays compatible with plt.imshow.
            cols (Default = 2): Number of columns in the figure.
            rows (Default = 3): Number of rows in the figure.
            cmap (Default = None): Used to display gray images.
    """
    plt.figure(figsize=(10, 11))
    for i, img in enumerate(imgs):
        plt.subplot(rows, cols, i+1)
        plt.imshow(img, cmap = cmap)
        plt.xticks([])
        plt.yticks([])
    plt.tight_layout(pad=0, h_pad=0, w_pad=0)
    plt.show()
```

---

1. Camera Calibration

Calculating the camera calibration matrix and distortion coefficients using a series of images of a chessboard.

```
def calibrate_camera(img,hPt,vPt,objp):
    # Use cv2.calibrateCamera() and cv2.undistort()
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (hPt,vPt), None)
    #defining a blank mask to start with
    cal_img = np.copy(img)
      # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)
        cal_img = cv2.drawChessboardCorners(cal_img, (hPt,vPt), corners, ret)
    return cal_img
    #Combine white and yellow mask images
    mask = cv2.bitwise_or(white_mask, yellow_mask)
    masked_image = cv2.bitwise_and(img, img, mask = mask)
```

2. Remove Distortion using the calibration matrix

Undistort image using camera calibration matrix

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

3. Perspective Transform

Transform the undistorted image to a "birds eye view" of the road which focuses only on the lane lines and displays them in such a way that they appear to be relatively parallel to each other.
This will make it easier later on to fit polynomials to the lane lines and measure the curvature.

```
# Perspective transform undistort image
# return unwarped perspective transformed image
def transform(undist,src,dst,img_size):
    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(undist, M, img_size)
    return warped
```

4. Extract Channel & Apply Binary Thresholds

In this step I attempted to convert the warped image to different color spaces and create binary thresholded images which highlight only the lane lines and ignore everything else. I found that the following color channels and thresholds did a good job of identifying the lane lines in the provided test images:

- The S Channel from the HLS color space, with a min threshold of 180 and a max threshold of 255, did a fairly good job of identifying both the white and yellow lane lines, but did not pick up 100% of the pixels in either one, and had a tendency to get distracted by shadows on the road.
- The L Channel from the LUV color space, with a min threshold of 225 and a max threshold of 255, did an almost perfect job of picking up the white lane lines, but completely ignored the yellow lines.
- The B channel from the Lab color space, with a min threshold of 155 and an upper threshold of 200, did a better job than the S channel in identifying the yellow lines, but completely ignored the white lines.

I chose to create a combined binary threshold based on the three above mentioned binary thresholds, to create one combination thresholded image which does a great job of highlighting almost all of the white and yellow lane lines.
Note: The S binary threshold was left out of the final combined binary image and was not used in detecting lane lines because it added extra noise to the binary image and interfered with detecting lane lines accurately.

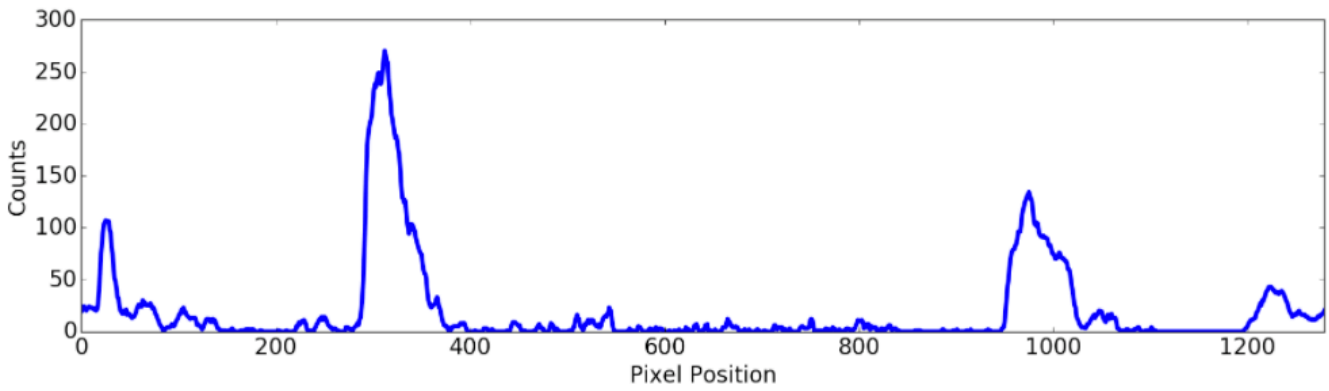
```
  def chennel_extraction(img):
```

```
    img = np.copy(img)
    s_channel = cv2.cvtColor(img, cv2.COLOR_BGR2HLS)[:,:,2]

    l_channel = cv2.cvtColor(img, cv2.COLOR_BGR2LUV)[:,:,0]

    b_channel = cv2.cvtColor(img, cv2.COLOR_BGR2Lab)[:,:,2]

    # Threshold color channel
    s_thresh_min = 180
    s_thresh_max = 255
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh_min) & (s_channel <= s_thresh_max)] = 1

    b_thresh_min = 145
    b_thresh_max = 200
    b_binary = np.zeros_like(b_channel)
    b_binary[(b_channel >= b_thresh_min) & (b_channel <= b_thresh_max)] = 1

    l_thresh_min = 225
    l_thresh_max = 255
    l_binary = np.zeros_like(l_channel)
    l_binary[(l_channel >= l_thresh_min) & (l_channel <= l_thresh_max)] = 1

    #color_binary = np.dstack((u_binary, s_binary, l_binary))

    combined_binary = np.zeros_like(s_binary)
    combined_binary[(l_binary == 1)|(s_binary == 1) | (b_binary == 1)] = 1
    return combined_binary
```

5 Find Lane peaks using Histogram

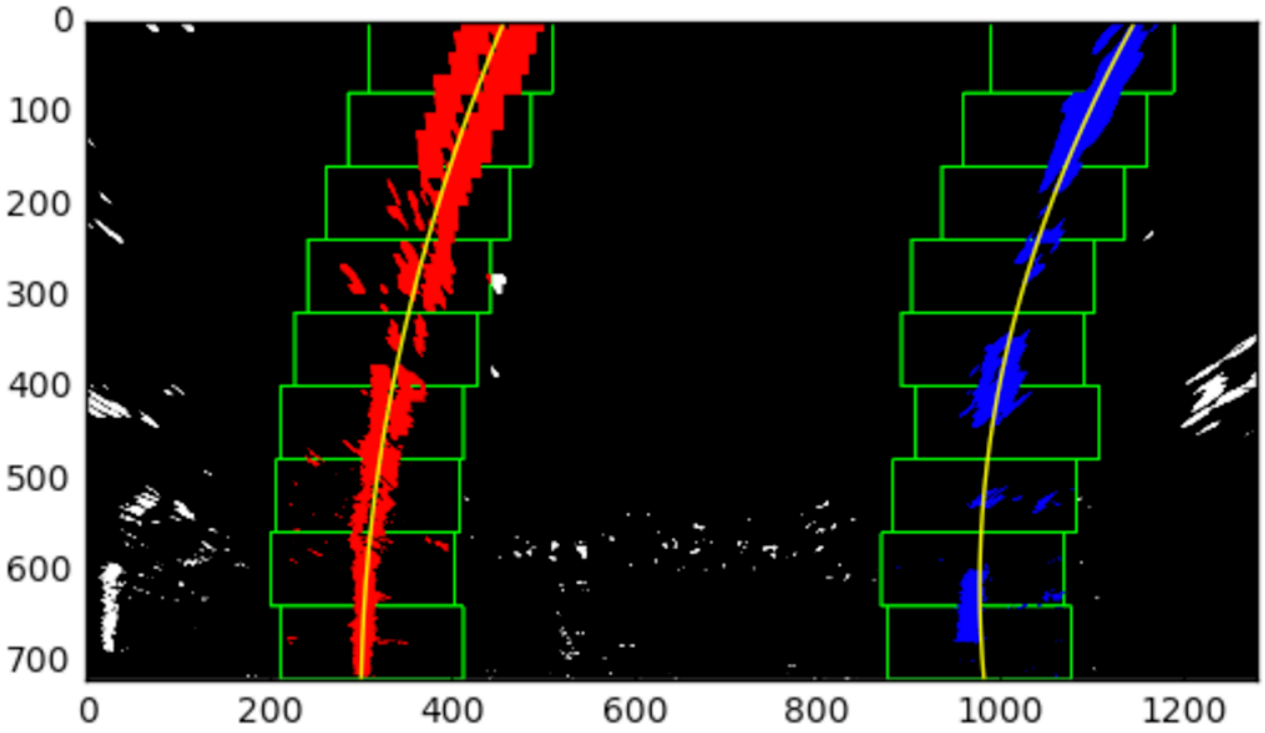
Find Lane using Histogram peaks

.



6. Search Lanes using sliding window

With the histogram we are adding up the pixel values along each column in the image. In our 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. We can use that as a starting point for where to search for the lines. From that point, we can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.



7. Find radius of the curve & Vehicle's position from center of the lane

---

At this point I was able to use the combined binary image to isolate lane line pixels and fit a polynomial to each of the lane lines. The space in between the identified lane lines is filled in to highlight the driveable area in the lane. The position of the vehicle was measured by taking the average of the x intercepts of each line.

The equation for calculating radius of curvature was discovered at this page: http://www.intmath.com/applications-differentiation/8-radius-curvature.php

In the function fill_lane() below, lane lines are detected by identifying peaks in a histogram of the image and detecting nonzero pixels in close proximity to the peaks.

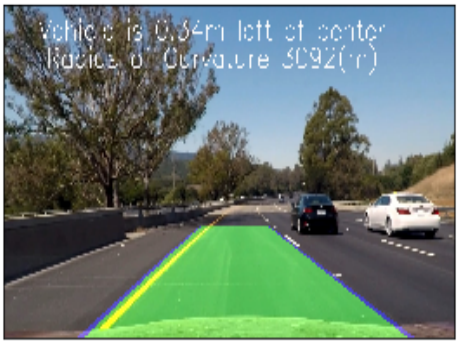
8. Merge Lines & Original Images

Merge the lane boundaries and numerical estimation of lane curvature and vehicle position with the original images










7. Apply on video streams

Now, we'll use the above functions to detect lane lines from a video stream. The video inputs are in root folder. The video outputs are generated in output_videos folder.

```
# Pipeline for finding Lanes
def process_frame(image):
    img_size = (image.shape[1], image.shape[0])
    undist = undistort(np.copy(image))
     # Perform perspective transform
    offset = 0
    src = np.float32([[490, 482],[810, 482],
                      [1250, 720],[0, 720]])
    dst = np.float32([[0, 0], [1280, 0],
                     [1250, 720],[40, 720]])
    unwarp = transform(undist, src, dst,img_size)

    #combined_binary = chennel_extraction(unwarp)
    # Generate binary thresholded images
    b_channel = cv2.cvtColor(unwarp, cv2.COLOR_RGB2Lab)[:,:,2]
    l_channel = cv2.cvtColor(unwarp, cv2.COLOR_RGB2LUV)[:,:,0]

    # Set the upper and lower thresholds for the b channel
    b_thresh_min = 145
    b_thresh_max = 200
    b_binary = np.zeros_like(b_channel)
    b_binary[(b_channel >= b_thresh_min) & (b_channel <= b_thresh_max)] = 1

    # Set the upper and lower thresholds for the l channel
    l_thresh_min = 215
    l_thresh_max = 255
    l_binary = np.zeros_like(l_channel)
    l_binary[(l_channel >= l_thresh_min) & (l_channel <= l_thresh_max)] = 1

    combined_binary = np.zeros_like(b_binary)
    combined_binary[(l_binary == 1) | (b_binary == 1)] = 1
    # Identify all non zero pixels in the image
    x, y = np.nonzero(np.transpose(combined_binary))

    if Left.found == True: # Search for left lane pixels around previous polynomial
        leftx, lefty, Left.found = Left.found_search(x, y)

    if Right.found == True: # Search for right lane pixels around previous polynomial
        rightx, righty, Right.found = Right.found_search(x, y)


    if Right.found == False: # Perform blind search for right lane lines
```

```
        rightx, righty, Right.found = Right.blind_search(x, y, combined_binary)

    if Left.found == False:# Perform blind search for left lane lines
        leftx, lefty, Left.found = Left.blind_search(x, y, combined_binary)

    lefty = np.array(lefty).astype(np.float32)
    leftx = np.array(leftx).astype(np.float32)
    righty = np.array(righty).astype(np.float32)
    rightx = np.array(rightx).astype(np.float32)


    # Calculate left polynomial fit based on detected pixels
    left_fit = np.polyfit(lefty, leftx, 2)

    # Calculate intercepts to extend the polynomial to the top and bottom of warped image
    leftx_int, left_top = Left.get_intercepts(left_fit)

    # Average intercepts across n frames
    Left.x_int.append(leftx_int)
    Left.top.append(left_top)
    leftx_int = np.mean(Left.x_int)
    left_top = np.mean(Left.top)
    Left.lastx_int = leftx_int
    Left.last_top = left_top

    # Add averaged intercepts to current x and y vals
    leftx = np.append(leftx, leftx_int)
    lefty = np.append(lefty, 720)
    leftx = np.append(leftx, left_top)
    lefty = np.append(lefty, 0)

    # Sort detected pixels based on the yvals
    leftx, lefty = Left.sort_vals(leftx, lefty)
    maxLx = leftx[-1]
    #print("bef lx ",leftx)
    #print("mL",maxLx)
    t_min=maxLx-50
    t_max=maxLx+50
    newleftx = np.copy(leftx)
    newleftx[(leftx <= t_min) | (leftx >= t_max)] = maxLx
    #print("af lx ",newleftx)
    Left.X = newleftx
    Left.Y = lefty

    # Recalculate polynomial with intercepts and average across n frames
    left_fit = np.polyfit(lefty, leftx, 2)
    Left.fit0.append(left_fit[0])
    Left.fit1.append(left_fit[1])
    Left.fit2.append(left_fit[2])
    left_fit = [np.mean(Left.fit0),
                np.mean(Left.fit1),
                np.mean(Left.fit2)]

    # Fit polynomial to detected pixels
    left_fitx = left_fit[0]*lefty**2 + left_fit[1]*lefty + left_fit[2]
    Left.fitx = left_fitx

    # Calculate right polynomial fit based on detected pixels
    right_fit = np.polyfit(righty, rightx, 2)

    # Calculate intercepts to extend the polynomial to the top and bottom of warped image
```

```
    rightx_int, right_top = Right.get_intercepts(right_fit)

    # Average intercepts across 5 frames
    Right.x_int.append(rightx_int)
    rightx_int = np.mean(Right.x_int)
    Right.top.append(right_top)
    right_top = np.mean(Right.top)
    Right.lastx_int = rightx_int
    Right.last_top = right_top
    rightx = np.append(rightx, rightx_int)
    righty = np.append(righty, 720)
    rightx = np.append(rightx, right_top)
    righty = np.append(righty, 0)

    # Sort right lane pixels
    rightx, righty = Right.sort_vals(rightx, righty)
    maxRx = rightx[-1]
    #print("bef rx ",rightx)
    #print("mL",maxRx)
    t_min=maxRx-50
    t_max=maxRx+50
    newrightx = np.copy(rightx)
    newrightx[(rightx <= t_min) | (rightx >= t_max)] = maxRx
    #print("af lx ",newrightx)
    Right.X = newrightx
    Right.Y = righty



    # Recalculate polynomial with intercepts and average across n frames
    right_fit = np.polyfit(righty, rightx, 2)
    Right.fit0.append(right_fit[0])
    Right.fit1.append(right_fit[1])
    Right.fit2.append(right_fit[2])
    right_fit = [np.mean(Right.fit0), np.mean(Right.fit1), np.mean(Right.fit2)]

    # Fit polynomial to detected pixels
    right_fitx = right_fit[0]*righty**2 + right_fit[1]*righty + right_fit[2]
    Right.fitx = right_fitx

    # Compute radius of curvature for each lane in meters
    left_curverad = Left.radius_of_curvature(leftx, lefty)
    right_curverad = Right.radius_of_curvature(rightx, righty)

    # Only print the radius of curvature every 3 frames for improved readability
    if Left.count % 3 == 0:
        Left.radius = left_curverad
        Right.radius = right_curverad

    # Calculate the vehicle position relative to the center of the lane
    position = (rightx_int+leftx_int)/2
    distance_from_center = abs((640 - position)*3.7/700)

    Minv = cv2.getPerspectiveTransform(dst, src)

    warp_zero = np.zeros_like(combined_binary).astype(np.uint8)
    color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
    pts_left = np.array([np.flipud(np.transpose(np.vstack([Left.fitx, Left.Y])))])
    pts_right = np.array([np.transpose(np.vstack([right_fitx, Right.Y]))])
    pts = np.hstack((pts_left, pts_right))
    cv2.polylines(color_warp, np.int_([pts]), isClosed=False, color=(0,0,255), thickness = 40)
```

```
    cv2.fillPoly(color_warp, np.int_(pts), (34,255,34))
    newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image.shape[0]))
    result = cv2.addWeighted(undist, 1, newwarp, 0.5, 0)
    #print(left_fitx)
    #print(right_fitx)
    # Print distance from center on video
    if position > 640:
        cv2.putText(result, 'Vehicle is {:.2f}m left of center'.format(distance_from_center),
(100,80),
                fontFace = 16, fontScale = 2, color=(255,255,255), thickness = 2)
    else:
        cv2.putText(result, 'Vehicle is {:.2f}m right of center'.format(distance_from_center),
(100,80),
                fontFace = 16, fontScale = 2, color=(255,255,255), thickness = 2)
    # Print radius of curvature on video
    cv2.putText(result, 'Radius of Curvature {}(m)'.format(int((Left.radius+Right.radius)/2)),
(120,140),
             fontFace = 16, fontScale = 2, color=(255,255,255), thickness = 2)


    Left.count += 1
    return result
```

Conclution

The project succeeded in detecting the lane lines clearly in the video streams. This project is intended. A possible improvement would be to modify the above pipeline to improve the accuracy 7 performance of curved lane line