# Advanced Lane Finding Report

My pipeline for this project consists of 10 steps.These steps are :

- step 1: Import libraries
- step 2: Compute the camera calibration matrix
- step 3: Apply a distortion correction to raw images.
- step 4: Use color transforms and gradients to create a thresholded binary image.
- step 5: Apply a perspective transform to rectify binary image
- step 6 :Detect lane pixels and fit to find the lane boundary.
- step 7: Determine the curvature of the lane and vehicle position with respect to center.
- step 8: Warp the detected lane boundaries back onto the original image.
- step 9: Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.
- step 10: Pipeline Video

## step 1:

I imported libraries at the first step.

## step 2: Compute the camera calibration matrix

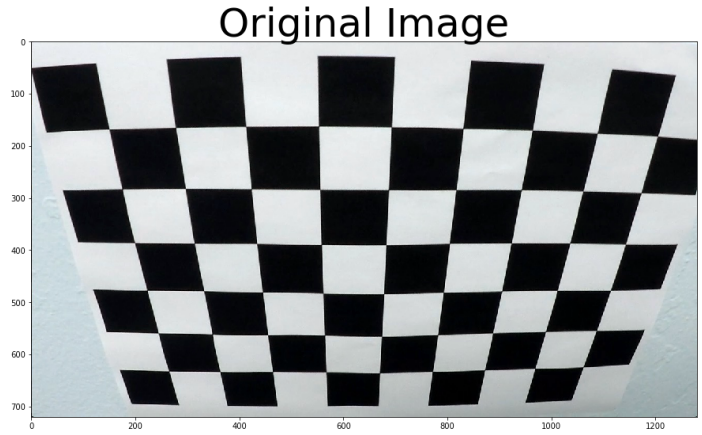
The code for camera calibartion is located in step 2 of "advanced lane finding.ipynb".

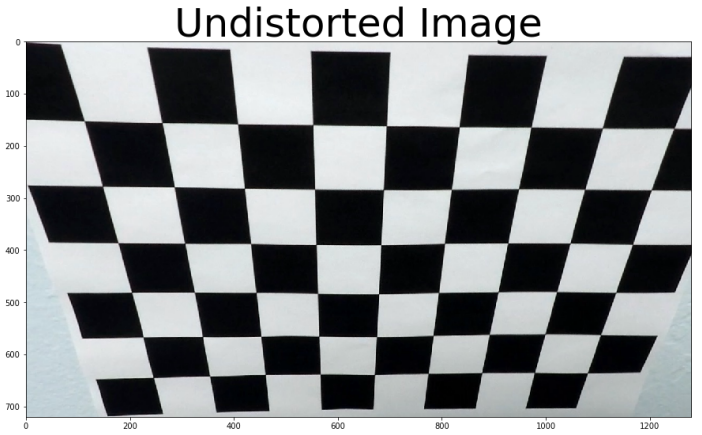
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of 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 `cv2.calibrateCamera()` function.

## step 3: Apply a distortion correction

I applied this distortion correction to the test image using the `cv2.undistort()` function in step 3 of "advanced lane finding.ipynb". And obtained this result:



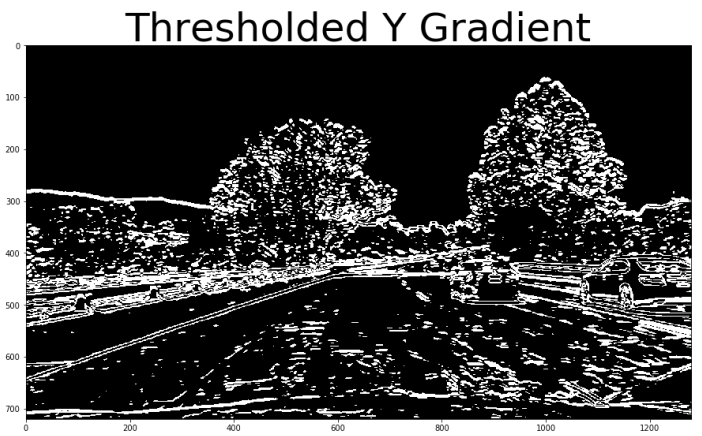

## step 4: Use color transforms and gradients

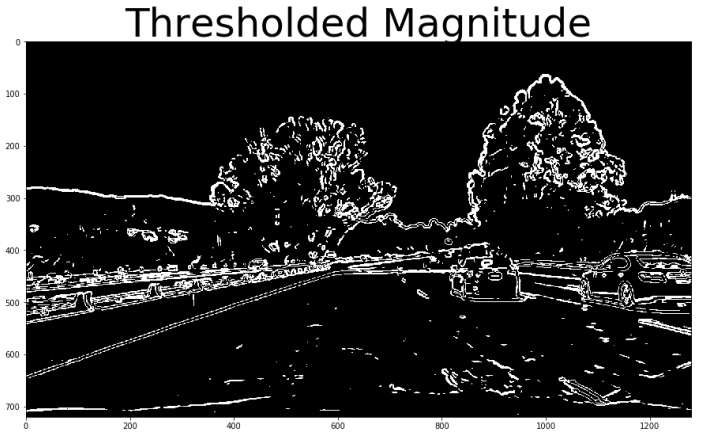
This setion consists of 5 subsections. In each section, I applied different gradiant threshholds and color transform and in the final section combined all threshholds to create a thresholded binary image. This threshhsolds would be found in step 4_1 to step 4_5 of "advanced lane finding.ipynb". The functions that I used for each step were:

- `abs_sobel_thresh()` used for applying sobel of X and Y
- `mag_thresh()` used for applying magnitude of gradinet
- `dir_threshold()` used for applying direction of gradinet
- `hls_select()` used for applying color threshhold
- `combine_threshhold()` used for combining all threshholds

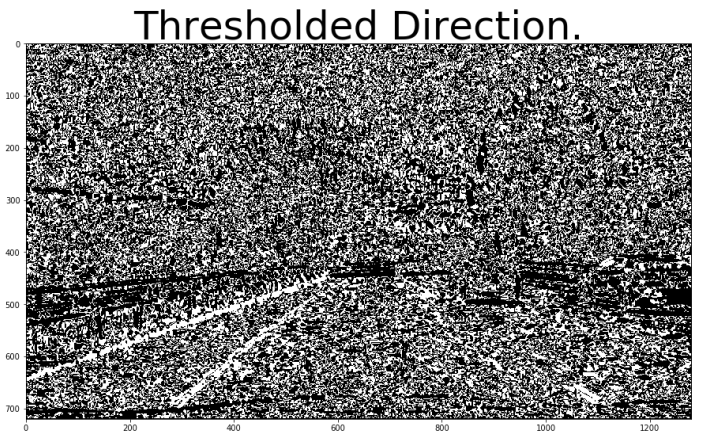
The images below are the results of this step.



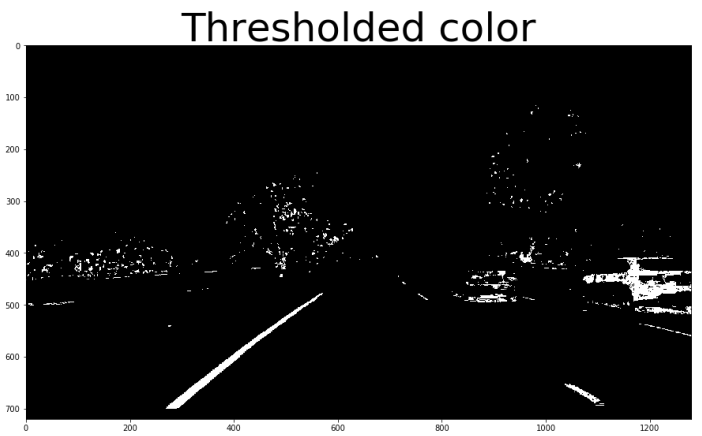



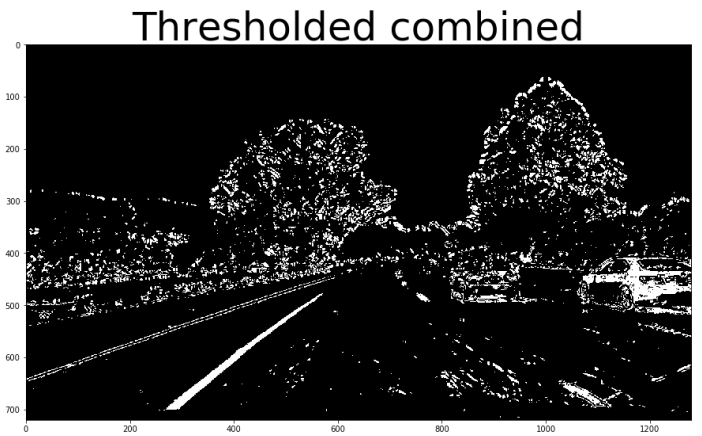













## step 5: Apply a perspective transform

The code for my perspective transform includes a function called `presepective_transform()`, which appears in step 5 of "advanced lane finding.ipynb". The `presepective_transform()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

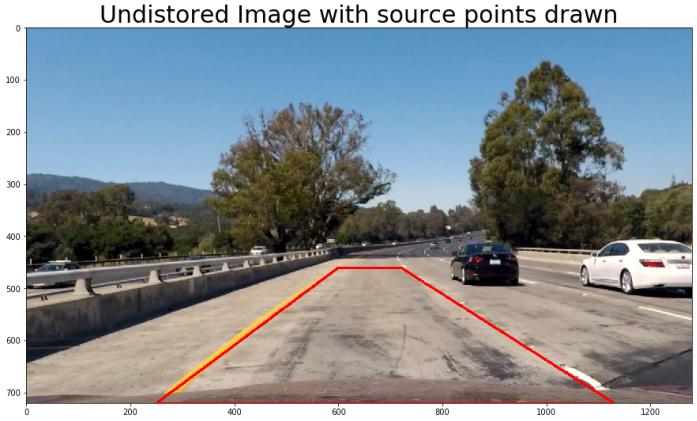
```
b_l_src = [250, img.shape[0]] # bottom left for source
b_r_src = [1130,img.shape[0]] # bottom right for source
t_r_src = [720,460] # top right for source
t_l_src = [600,460] # top left for source

b_l_dst = [220, img.shape[0]] # bottom left for destination
b_r_dst = [1065,img.shape[0]] # bottom right for destination
t_r_dst = [1065,0] # top right for destination
t_l_dst = [220,0] # top left for destination

# suorce points
src = np.float32([b_l_src,b_r_src,t_r_src,t_l_src])
# destination points
dst = np.float32([b_l_dst,b_r_dst,t_r_dst,t_l_dst])
```

This resulted in the following source and destination points:

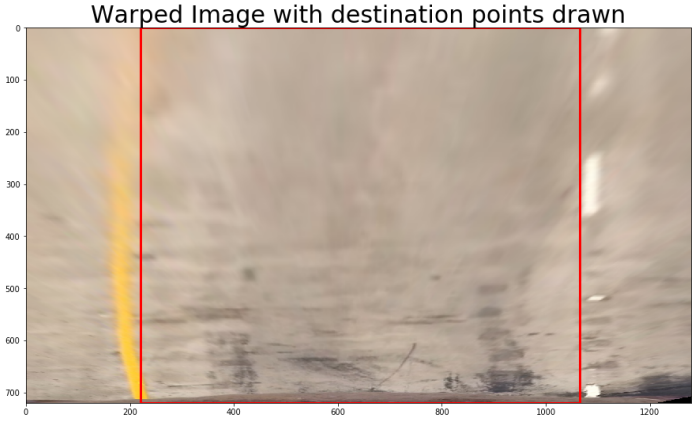
| Source | Destination |
|---|---|
| (250, 720) | (220, 720) |
| (1130, 720) | (1065, 720) |
| (720, 460) | (1065, 0) |
| (600, 460) | (220, 0) |

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image. The `cv2.warpPerspective()` was used for warping the image. The image below is the result:

Undistored Image with source points drawn



Warped Image with destination points drawn



# step 6: Detect and fit lane pixels

This step consists of 3 sections. It can be found in step 6_1 to 6_3 in step 6 of "advanced lane finding.ipynb". After applying calibration, thresholding, and a perspective transform to a road image, I have a binary image where the lane lines stand out clearly. Then I 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. To do this, we have:

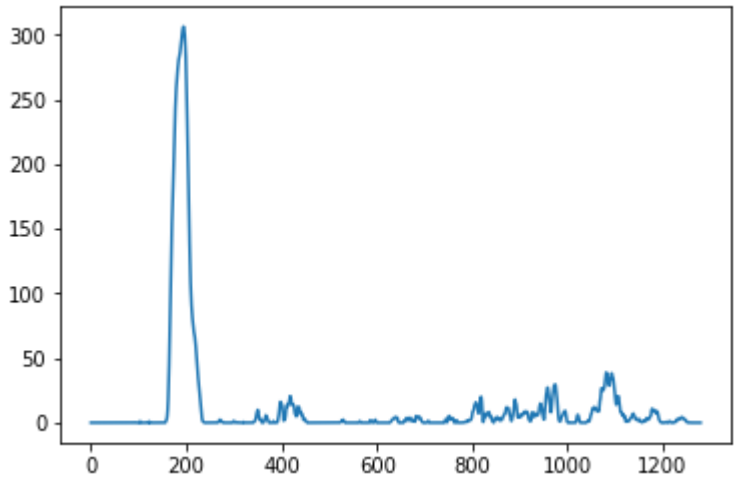
1. Finding the Lines (Histogram Peaks): Plotting a histogram of where the binary activations occur across the image is one potential solution.
2. Finding the Lines (Sliding Window): Use sliding windows moving upward in the image to determine where the lane lines go.
3. Finding the Lines (Search from Prior): It helps to track the lanes through sharp curves and tricky conditions

**Histogram Peaks**

The code below is used to find the peaks in a histogram.

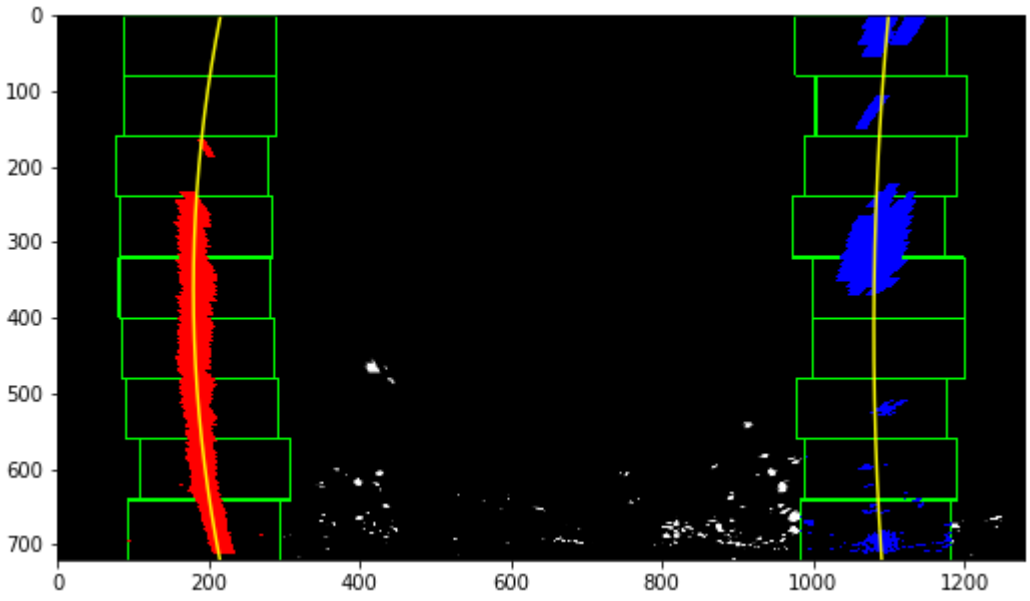
```
warped_combined = presepective_transform(combined)[0]
histogram = np.sum(warped_combined[warped_combined.shape[0]//2:,:], axis=0)
# Visualize the resulting histogram
plt.plot(histogram)
```

This is the result:



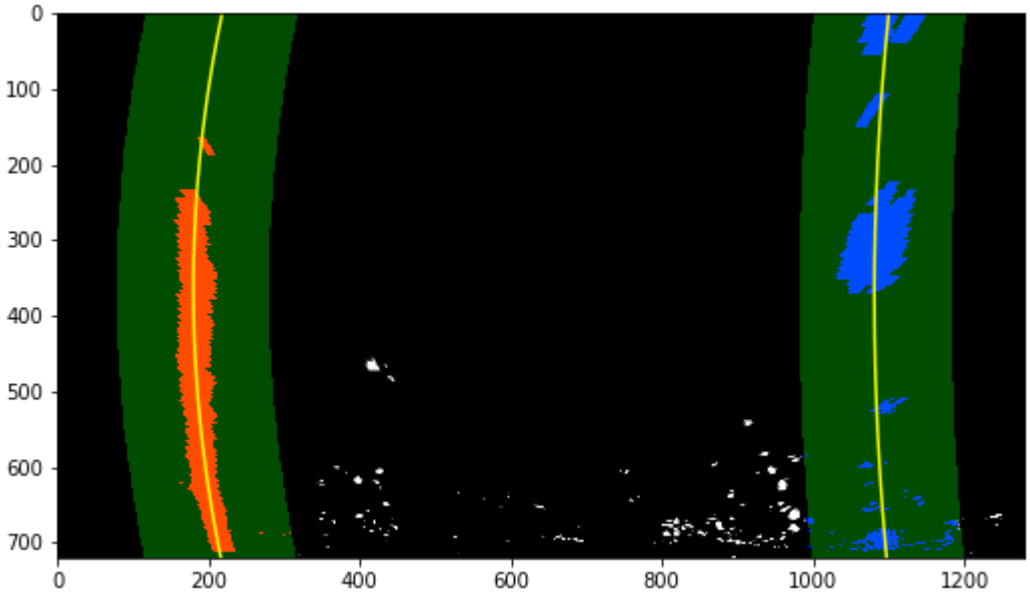

**Sliding Window**

In this section I implemented sliding windows and fit a polynomial using `find_lane_pixels()` function.



**Search from Prior**

The function `search_around_poly()` uses the previosly calculated `left_fit` and `right fit` to try to identify the lane lines in a consecutive image.




# step 7: Determine the curvature of the lane and vehicle position with respect to center

The curvature radius is computed according to the formula described in material of class. Now we have polynomial fits and we can calculate the radius of curvature and vehicle offset. The radius should be the same as real world space, so we transform x and y from pixels space to meters.

The code below is used to determine the curvature and vehicle offset center.

```
## Measuring Curvature

def measure_curvature_real(l_fx, r_fx,img_size):


    ploty = np.linspace(0, img.shape[0] - 1, img.shape[0])
    midell_x = img.shape[1]//2
    car_pos = ( l_fx[-1] +  r_fx[-1])/2
    leftx = l_fx[::-1]  # Reverse to match top-to-bottom in y
    rightx = r_fx[::-1]  # Reverse to match top-to-bottom in y
    left_fit = np.polyfit(ploty, leftx, 2)
    right_fit = np.polyfit(ploty, rightx, 2)
    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]

    # 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
    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)

    # Define y-value where we want radius of curvature
    # We'll choose the maximum y-value, corresponding to the bottom of the image
    y_eval = np.max(ploty)

    #####  Implement the calculation of R_curve (radius 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])  ## Implement the calculation of the left l
ine here
    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])   ## Implement the calculation of the r
ight line here

    ## Horizontal car offset
    offsetx = (midell_x - car_pos) * xm_per_pix

    return offsetx, left_curverad, right_curverad


# Calculate the radius of curvature in meters for both lane lines
offsetx, left_curverad, right_curverad = measure_curvature_real(l_fx, r_fx, img_siz
e)

print(' left_curverad',left_curverad, 'm')
print(' right_curverad', right_curverad,'m')
print(' offsetx',offsetx,'m')
```

# step 8: Warp the detected lane boundaries back onto the original image

Now the lane bounderis and curvature of the lane are found. So we detect the lane lines onto the warped blank version of the image. I used `detect_line()` function which would be found in step 8 of "advanced lane finding.ipynb".


# step 9: Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position

I used `metrics ()` function to add metrics to the image. the parametrs that is added to the image from step 8 are radius of left and right curvatures and vehicle offset which is calculated in step 7.


# step 10: Pipeline video

In this final step, I used all pervious steps to creat a pipline. The code below is my pipeline:

```
class ProcessImage:
    def __init__(self, images):
        images = glob.glob(images)

        # Calibrate camera
        self.ret, self.mtx, self.dist, self.rvecs, self.tvecs = calibrate_camera(im
ages)

    def __call__(self, img):

        # Undistord image
        img = cv2.undistort(img, mtx, dist, None, mtx)
        ksize=15

        # Calculate directional gradient
        gradx = abs_sobel_thresh(img, orient='x', sobel_kernel=ksize, thresh=(30, 1
00))

        # Calculate gradient magnitude
        mag_binary = mag_thresh(img, sobel_kernel=ksize, mag_thresh=(70, 100))

        # Calculate gradient direction
        dir_binary = dir_threshold(img, sobel_kernel=ksize, thresh=(0.7, 1.3))

        # Calculate color threshold
        hls_binary = hls_select(img, thresh=(170, 255))

        # Combine all the thresholds to identify the lane lines
        combined = combine_threshhold(gradx, grady, mag_binary, dir_binary, hls_bin
ary, ksize=15)

        # Apply a perspective transform
        warped_combined, M , M_inv = presepective_transform(combined)

        # find lane detection
        l_flane, r_flane, l_fx, r_fx, o = search_around_poly(warped_combined,l_f,r_
f)

        # Warp the detected lane boundaries
        unwarp_img = detect_line(img, warped_combined, l_fx, r_fx, M_inv)

        # Add metrics
        out_img = metrics (unwarp_img,l_fx,r_fx)

        return out_img
```

I use moviepy.editor to apply this pipeline to a video. [link to my video result (./project_video_solution.mp4)]

```
input_video = './project_video.mp4'
output_video = './project_video_solution.mp4'

clip1 = VideoFileClip(input_video)

# Process video frames with our 'process_image' function
process_image = ProcessImage('./camera_cal/calibration*.jpg')

white_clip = clip1.fl_image(process_image)

%time white_clip.write_videofile(output_video, audio=False)
```

# Discussion

- One problem that I faced was choosing suitable points for source and destination to apply perspective transform. Second problem was, I had to try a lot with gradient and color channnel thresholding.
- The pipeline conflicts when the vehicle bounces and also whenever the slope of the road changes. The reason behind this is due to the perpective transform parameters are not chosen properly. One approach is to find the vanishing point for the image.
- The pipeline might fail when vehicle is not driving within the lane lines and it is too close to the left or right lanes. Because I assumed the vehicle is driving in the center of the lane. Also it might fail when the vehicle is going to change lanes.
- To make it more robust, we can choose more appropriate parameters for perspective transform and choose smaller section to take the transform. Also we can take an average of the points over previous frames to have a smooth transition per frame.