Udacity Self-Driving Car Engineer

Report

- **Project:** Advanced Lane Finding
- **Name:** Shao Hongxu

Let me just use some brief sentence to explain the thing I did, basically these are 8 big parts here:

Part 1: import necessary lib
Part 2: compute the camera matrix and distortion coefficients
Part 3: image pre-processing
Part 4: convert image to a "birds-eye view"
Part 5: lane line detection
Part 6: meter calculation
Part 7: warp back the image
Part 8: Summary

To be specific as below, you will find some note on Script, key take away and Area could be improved:

Part 1: import necessary lib:

```
In [5]: import numpy as np
        import cv2
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
```

Part 2: compute the camera matrix and distortion coefficients:

```
for i in range(1,20):

    image_name='calibration'+str(i)
    existing_folder_name='camera_cal/'
    New_foler_name='output_images/'

    fname = existing_folder_name+image_name+".jpg"
    img = cv2.imread(fname)

    objpoints=[]
    imgpoints=[]
    objp=np.zeros((6*9,3), np.float32)
    objp[:,:2]=np.mgrid[0:9,0:6].T.reshape(-1,2)

    gray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    ret, corners = cv2.findChessboardCorners(gray,(9,6),None)

    if ret == True:
        imgpoints.append(corners)
        objpoints.append(objp)
        img=cv2.drawChessboardCorners(img, (9, 6), corners, ret)
        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
        dst = cv2.undistort(img, mtx, dist, None, mtx)
        plt.imshow(dst)
        plt.show()

        cv2.imwrite(New_foler_name+image_name+"_result.jpg", dst)
```

-> Key take away: To archive the goal of distortion correction, the main parameter is to identify **chessboard size, in this case, it is 9x6**.

-> Result:

Original image:

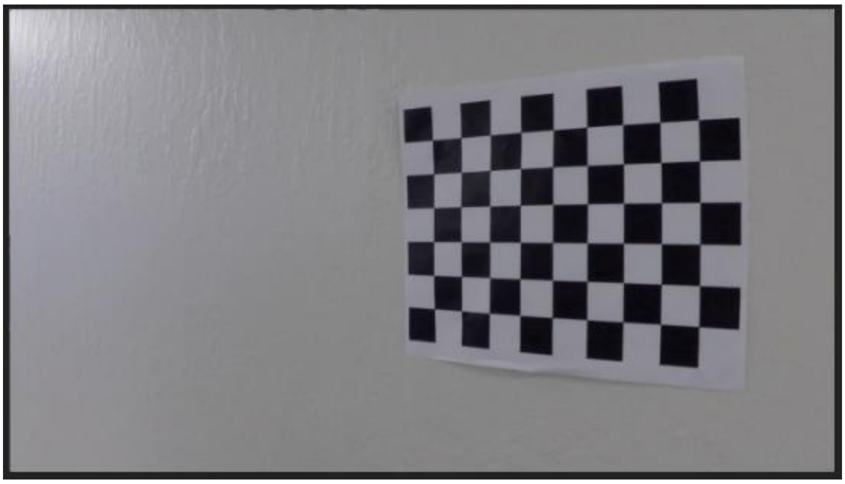
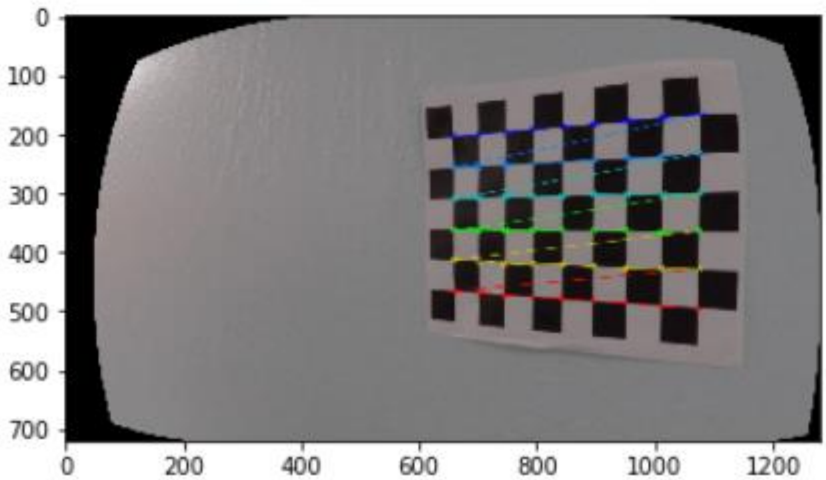


Image after distortion correction:

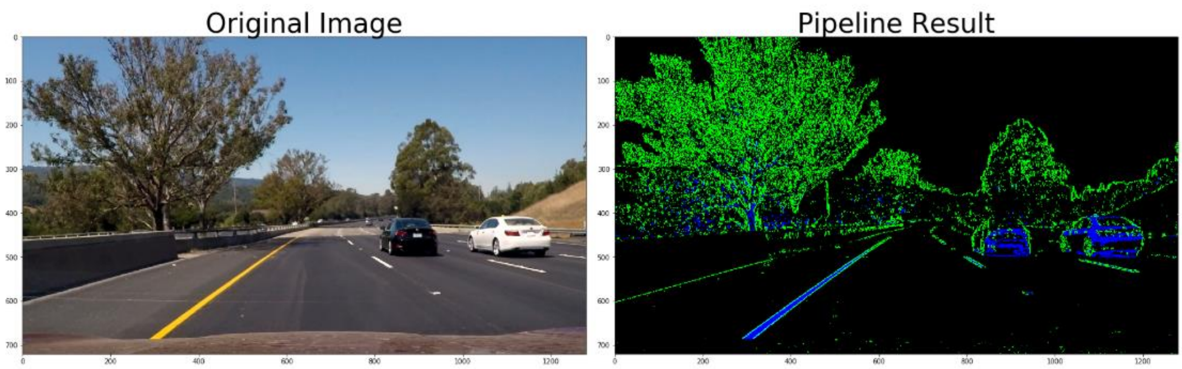


Part 3: image pre-processing

Create a function like in Udacity lesson, draw the pipeline for input image.

```
# Edit this function to create your own pipeline.
def pipeline(img, s_thresh=(170, 255), sx_thresh=(20, 100)):
    img = np.copy(img)
    # Convert to HLS color space and separate the V channel
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    l_channel = hls[:,:,1]
    s_channel = hls[:,:,2]
    # Sobel x
    sobelx = cv2.Sobel(l_channel, 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
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= sx_thresh[0]) & (scaled_sobel <= sx_thresh[1])] = 1

    # Threshold color channel
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh[1])] = 1
    # Stack each channel
    color_binary = np.dstack(( np.zeros_like(sxbinary), sxbinary, s_binary)) * 255
    return color_binary

result = pipeline(img)
```

As a result:

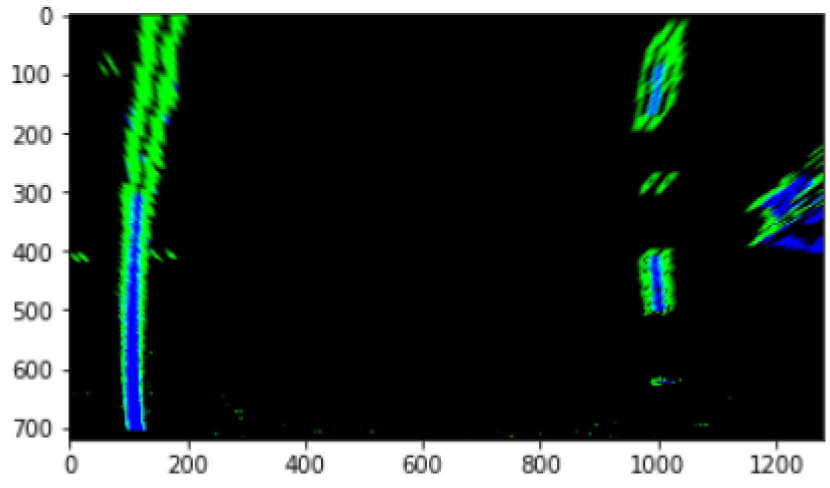


Part 4: Convert image to a "birds-eye view"

convert the pipeline image into bird view using below function. Here you can see, I have identified 4 points to map between pipeline image & bird view image.

```python
def warp(img):

    src=np.float32(
        [[575,483],
         [771,483],
         [1092,680],
         [296,680]])
    dst=np.float32(
        [[100,300],
         [1000,300],
         [1000,700],
         [100,700]])
    img_size=(img.shape[1], img.shape[0])
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

    return warped
```

As a result, you can see below image as a bird view image output:



Here the key take away is, better to specific the area for 2 lines only. I have tried to include as much information as much from original pipeline image, but the result not that properly. Sometimes left side road corner has been considered as lane line. And I realized I should give more restriction, and that indeed improved the code as a result.

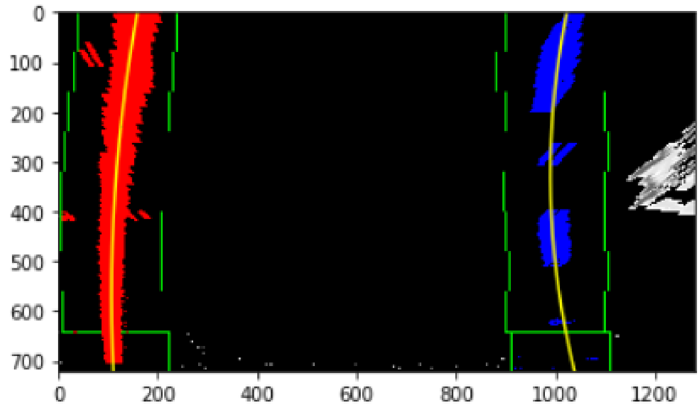
Part 5: lane line detection

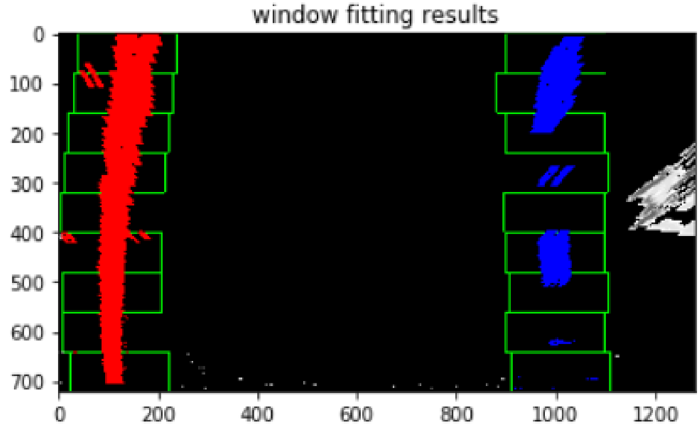
For next step, is to identify the left & right lane line respectively in different colour, here is the lesson code to archive that:

```python
# Assuming you have created a warped binary image called "binary_warped"
# Take a histogram of the bottom half of the image
histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)
# 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 = []
```

```python
# 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)
```

As a result, code is able to detect the 2 lines with colour red on left & blue on right.





Part 6: meter calculation

At the same time, lane meters will be calculated using below code:

```
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])
print(left_curverad, right_curverad)
# Example values: 1926.74 1908.48


# 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

# Fit new polynomials to x,y in world space
left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*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
return output
```

```python
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])
print(left_curverad, right_curverad)
# Example values: 1926.74 1908.48


# 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

# Fit new polynomials to x,y in world space
left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*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

# Calculate vehicle center
xMax = output.shape[1]*xm_per_pix
yMax = output.shape[0]*ym_per_pix
car_centre = xMax / 2
Left_line = left_fit_cr[0]*yMax**2 + left_fit_cr[1]*yMax + left_fit_cr[2]
Right_line = right_fit_cr[0]*yMax**2 + right_fit_cr[1]*yMax + right_fit_cr[2]
Middle_line = Left_line + (Right_line - Left_line)/2
meter_from_middle = Middle_line - car_centre
if meter_from_middle<0:
    meter_from_middle=-meter_from_middle
font = cv2.FONT_HERSHEY_SIMPLEX
fontColor = (0, 0, 0)
cv2.putText(result, 'Left curvature: {:.2f} m'.format(left_curverad), (600, 50), font, 1, fontColor, 2)
cv2.putText(result, 'Right curvature: {:.2f} m'.format(right_curverad), (600, 80), font, 1, fontColor, 2)
cv2.putText(result, 'from center: {:.2f} m'.format(meter_from_middle), (600, 110), font, 1, fontColor, 2)
plt.imshow(result)
plt.show()

print(meter_from_middle)
```

As a result, code will print out the number after each image conversion:

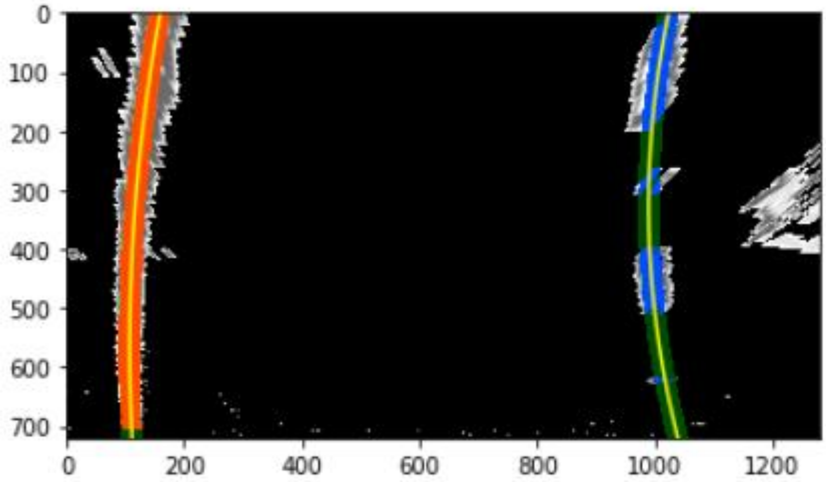
```
  7%|█████                                                          | 89/1261 [00:30<06:39,  2.94it/s]
2081.75107828 1933.55833182
665.603156408 m 579.049046509 m
  7%|█████                                                          | 90/1261 [00:30<06:38,  2.94it/s]
2084.12428449 2066.07743679
667.118875622 m 617.493689589 m
```

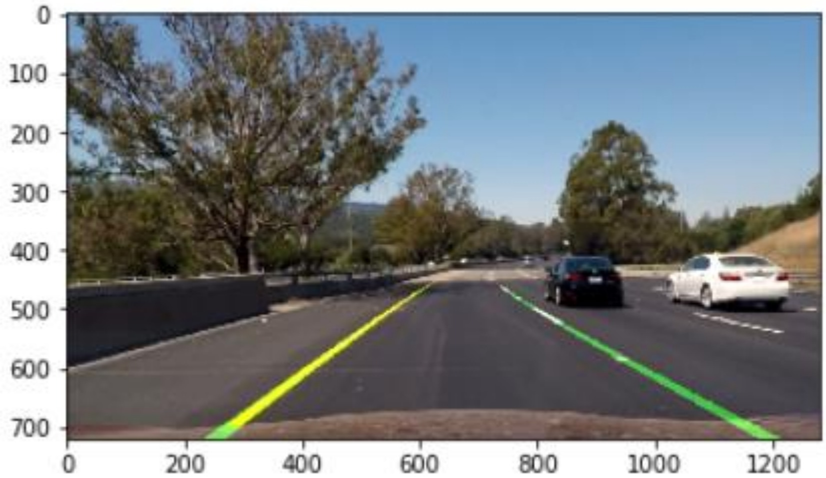
Part 7: warp back the image

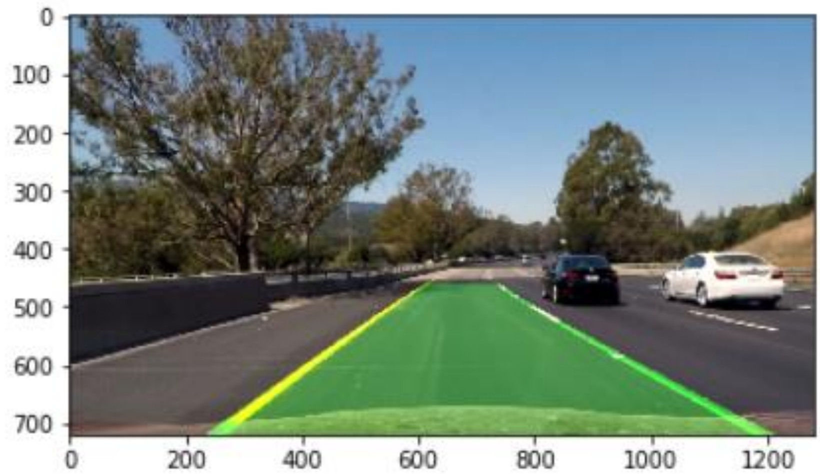
Next step after lane line detection, is to convert lane line back from bird view into normal view. Combine them together into a single picture using below code:

```python
def warp_back(img):

    src=np.float32(
        [[575,483],
         [771,483],
         [1092,680],
         [296,680]])
    dst=np.float32(
        [[250,300],
         [1000,300],
         [1000,700],
         [250,700]])
    img_size=(img.shape[1], img.shape[0])
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    warped_back = cv2.warpPerspective(img, Minv, img_size, flags=cv2.INTER_LINEAR)

    return warped_back
```

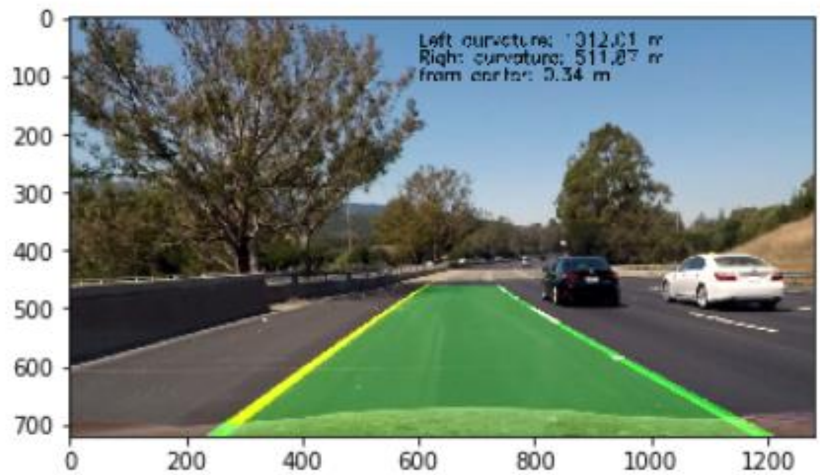
As you can see, same conversion matrix used for warping, but opposite way. As a result, image after all method will be shown as below with lane line in light green.








```
3092.50536434 1706.85209933
```




```
0.342263378891
1012.00863148 m 511.869998977 m
```

Inside project, I made video generation code as a separate set and to let first part of code used only for showing single image result. For video part, code will capture set of images from video and put them through the same conversion logic as we have done for image above. And showing process as below:

```
#video
white_clip = clip.fl_image(process_image) #NOTE: this function expects color images!!
white_clip.write_videofile(white_output, audio=False)
```

```
[MoviePy] >>>> Building video project_video_result.mp4
[MoviePy] Writing video project_video_result.mp4
100%|████████████████████████████████████████████████████████████| 1260/1261 [09:04<00:00,  2.31it/s]
[MoviePy] Done.
[MoviePy] >>>> Video ready: project_video_result.mp4
```

Part 8: Summary

These are the steps I could think of for project 4, as a base line. Still a lot of improvement needed to achieve a better accuracy.

Like the meter calculation, maybe it can be given a better exchange rate, now left & right meters are too much different most of the time.

Overall, I have learnt a lot from this project and compared with the first project, the lane line detection solution is much much better!