**Udacity Self-Driving Car Nanodegree**

**Project 4- Advanced Lane Finding**

**Author: Johannes Betz**

**Date: 07/21/2017**

Table of Contents

[1. Introduction and goals 3](#_Toc489548571)

[2. Camera Calibration 3](#_Toc489548572)

[3. Pipeline for single Images 4](#_Toc489548573)

[a. Distortion Correction 4](#_Toc489548574)

[b. Creating a binary Treshold image: Color Transformation, Gradient Transformation 5](#_Toc489548575)

[c. Perspective Transformation 8](#_Toc489548576)

[d. Finding lane line pixels and fit position with a polynomial 9](#_Toc489548577)

[e. Calculating the radius of a curvature and defining the position of the vehicle 11](#_Toc489548578)

[f. Example Image 12](#_Toc489548579)

[4. Pipeline for a video 13](#_Toc489548580)

[5. Discussion 14](#_Toc489548581)

# Introduction and goals

The following writeup includes the code and the description for an advance lane finding algorithm.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.

Every section provides one part of the project rubics.

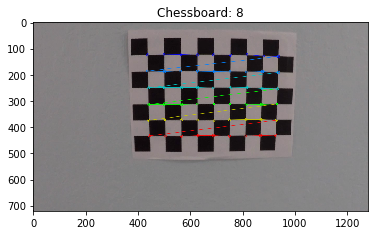
# Camera Calibration

For the camera calibration 20 chessboard pictures are provided in the folder camera\_cal. Each chessboard picture can be used for the camera calibration. The code looks like this:

**def calibrate\_camera**(calibration\_images, nx, ny):  
  
 # Arrays to store object points and image points from all the images  
 realpoints = [] # 3D points in real world space  
 imagepoints = [] # 2D points in image plane  
  
 # Prepare object points by creating 6x8 points in an array each with 3 columns for the x,y,z coordinates of each corner  
 objp = np.zeros((ny \* nx, 3), np.float32)  
  
 # Use numpy mgrid function to generate the coordinates that we want  
 objp[:, :2] = np.mgrid[0:nx, 0:ny].T.reshape(-1, 2)  
  
 **for** name **in** calibration\_images:  
  
 img = mpimg.imread(name)  
  
 gray = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)  
  
 # Find the chessboard corners  
 ret, corners = cv2.findChessboardCorners(gray, (nx, ny), **None**)  
  
 # If corners are found, add image points and object points  
 **if** (ret):  
 imagepoints.append(corners)  
  
 # Object points will be the same for all of the calibration images  
 # Since they represent a real chessboard  
 realpoints.append(objp)  
  
 **return** cv2.calibrateCamera(realpoints, imagepoints, img.shape[0:2], **None**, **None**)  
  
  
nx = 9  
ny = 6  
calibration\_images = glob.glob("camera\_cal/\*")  
  
# Calibrate the camera  
ret, mtx, dist, rvecs, tvecs = calibrate\_camera(calibration\_images, nx, ny)

First, the number of inner corners per row and column are counted (6 and 9). Then, the function calibrate\_camera is called to calibrate the camera. First, each pictures is read with the imread() function. Then, the picture is turned into greyscale with the cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY) function.

The corners are found by the cv2.findChessboardCorners function, where the grayscaled pictures are integrated. After that, a calibrated image looks like this:



# Pipeline for single Images

The following chapter 3 includes a complete pipeline for including pictures and reading information out o fit. Each step is written in one section.

# Distortion Correction

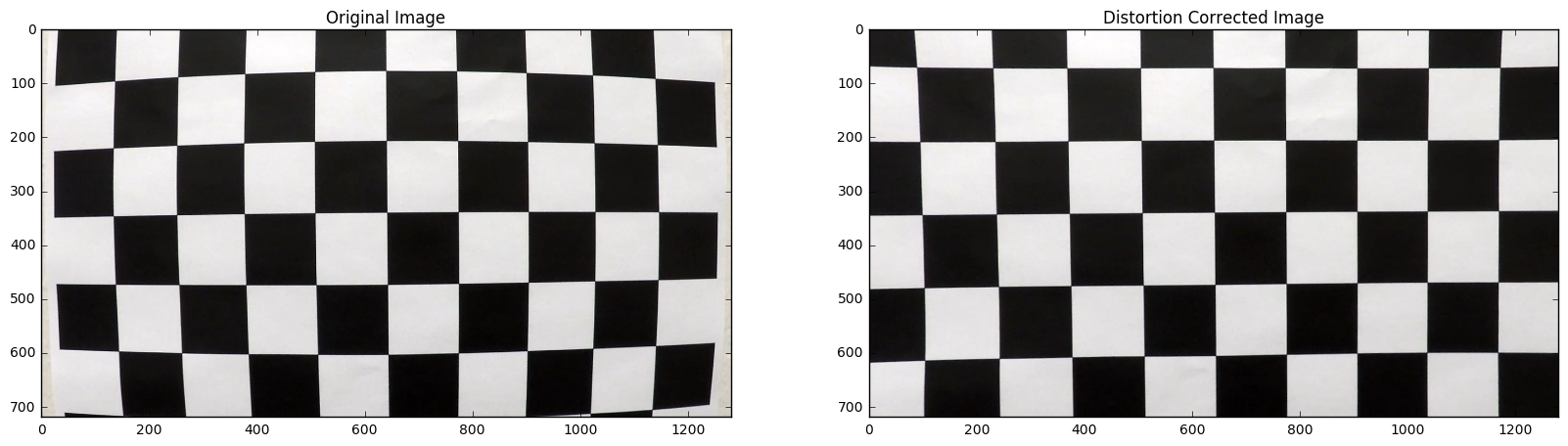
The function\_calibrate camera produces the variable „mtx“. This variable includes the camera coefficient matrix. In addition, the distrotation points are given back. The code for the distoration correction is listed below:

**def undistort**(image):  
  
 **return** cv2.undistort(image, mtx, dist)  
  
  
**def plot\_on\_subplots**(images, titles, cmap=**None**):  
  
 f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))  
  
 **if** cmap:  
 ax1.imshow(images[0], cmap=cmap)  
 **else**:  
 ax1.imshow(images[0])  
 ax1.set\_title(titles[0])  
  
 **if** (cmap):  
 ax2.imshow(images[1], cmap=cmap)  
 **else**:  
 ax2.imshow(images[1])  
  
 ax2.set\_title(titles[1])

The distortation correction is done for two test images

Example Image 1

1. # Perform distortion Correction on one of the calibration images  
   calibration\_image = plt.imread("camera\_cal/calibration1.jpg")  
     
   plot\_on\_subplots([calibration\_image, undistort(calibration\_image)], ["Original Image", "Distortion Corrected Image"])



Example Image 2

# Perform un-distortion on a test images  
test\_image = plt.imread("test\_images/test5.jpg")  
plot\_on\_subplots([test\_image, undistort(test\_image)], ["Original Image", "Distortion Corrected Image"])



# Creating a binary Treshold image: Color Transformation, Gradient Transformation

After the pictures are distorted, the goal was to create a binary threshold image. To create a binary threshold image, different steps are executed:

1. Apply the sobel operator: converts the image to grayscale, applies a sobel operator in the X direction, takes the absolute value, scales the result in the range 0-255 and performs a thresholding operation.
2. HLS Channel binary: performs thresholding on either the H, L or S channel of the image depending on the input parameter.
3. Greyscale Treshold: converts to grayscale using the cv2 cvtColor function and performs thresholding using the thresh parameter.
4. Color Selection: Performs color selection on the image converted to HLS color space by only selecting Yellow and White colors and returns the binary mask for the same
5. Combine: combines the above explained 4 operations to produce a binary image of the input image.

The Code for the Binary Treshold image is listed here:

**def get\_sobel\_binary**(image, thresh\_min=20, thresh\_max=200):  
  
 gray = cv2.cvtColor(image, cv2.COLOR\_RGB2GRAY)  
  
 # Take the sobel derivative in the x direction  
 sobelx = cv2.Sobel(gray, cv2.CV\_64F, 1, 0)  
  
 # Absolute x derivative to accentuate lines away from horizontal  
 abs\_sobelx = np.absolute(sobelx)  
  
 # Scale to 8 bit grayscale image  
 scaled\_sobel = np.uint8(255 \* abs\_sobelx / np.max(abs\_sobelx))  
  
 # Apply a threshold  
 sxbinary = np.zeros\_like(scaled\_sobel)  
 sxbinary[(scaled\_sobel >= thresh\_min) & (scaled\_sobel <= thresh\_max)] = 1  
  
 **return** sxbinary  
  
  
**def get\_channel\_index\_hls**(chanel\_name):  
  
 **if** (chanel\_name == "h"):  
 **return** 0  
 **elif** (chanel\_name == "l"):  
 **return** 1  
 **elif** (chanel\_name == "s"):  
 **return** 2  
  
  
**def get\_hls\_channel\_binary**(image, channel\_name='s', thresh\_min=180, thresh\_max=255):  
  
 # Convert to HLS color space  
 hls = cv2.cvtColor(image, cv2.COLOR\_RGB2HLS)  
  
 # Extract the desired channel  
 channel\_index = get\_channel\_index\_hls(channel\_name)  
 channel = hls[:, :, channel\_index]  
  
 # Apply the threshold  
 channel\_binary = np.zeros\_like(channel)  
 channel\_binary[(channel >= thresh\_min) & (channel <= thresh\_max)] = 1  
  
 **return** channel\_binary  
  
  
**def get\_grayscale\_thresholded\_img**(img, thresh=(130, 255)):  
  
 gray = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)  
  
 binary = np.zeros\_like(gray)  
  
 binary[(gray > thresh[0]) & (gray < thresh[1])] = 1  
  
 **return** binary  
  
  
**def get\_color\_selection**(image):  
  
 hsv = cv2.cvtColor(image, cv2.COLOR\_RGB2HSV)  
  
 lower\_yellow = np.array([0, 100, 100], dtype=np.uint8)  
 upper\_yellow = np.array([190, 250, 255], dtype=np.uint8)  
  
 upper\_white = np.array([200, 200, 200], dtype=np.uint8)  
 lower\_white = np.array([255, 255, 255], dtype=np.uint8)  
  
 # Get the white pixels from the original image  
 mask\_white = cv2.inRange(image, upper\_white, lower\_white)  
  
 # Get the yellow pixels from the HSV image  
 mask\_yellow = cv2.inRange(hsv, lower\_yellow, upper\_yellow)  
  
 # Bitwise-OR white and yellow mask  
 mask = cv2.bitwise\_or(mask\_white, mask\_yellow)  
  
 **return** mask  
  
**def get\_binary\_image**(image):  
  
  
 sobel\_binary = get\_sobel\_binary(image)  
  
 s\_channel\_binary = get\_hls\_channel\_binary(image)  
  
 l\_channel\_binary = get\_hls\_channel\_binary(image, channel\_name="l", thresh\_min=200)  
  
 gray\_scale\_thresholded\_image = get\_grayscale\_thresholded\_img(image)  
  
 color\_sel = get\_color\_selection(image)  
  
 combined\_binary = np.zeros\_like(s\_channel\_binary)  
  
 combined\_binary[(color\_sel == 255) | ((s\_channel\_binary == 1) & (l\_channel\_binary == 1))  
 | ((sobel\_binary == 1) & (gray\_scale\_thresholded\_image == 1)) | (l\_channel\_binary == 1)] = 1  
  
 **return** combined\_binary

The complete binary image pipeline is displayed wit the one of the test files and is executed like this:

bin\_image = get\_binary\_image(test\_image)  
plot\_on\_subplots([test\_image , bin\_image], ["Original Image", "Binary Image"], cmap="gray")



# Perspective Transformation

The next step is applying a perspective transform to rectify the binary image, which is usually named as birds-eye view. The test image is picked and a hough lines were drawn on the image. After that, the end points of the hough lines were used as the scry points in the perspective transformation. An additional rectangle derice from the src points are used as the dst points in the hough transformation. With the “getPerspectiveTransfroM” from openCv the transformation matrix is calcucalted. The code for doing this is listed below:

lines, lines\_image = hough\_lines(roi\_image, rho=rho, theta=theta, threshold=threshold,  
 min\_line\_len=min\_line\_length,  
 max\_line\_gap=max\_line\_gap)  
  
M, M\_inv = compute\_transformation\_matrix(image, lines)  
  
persp\_transform = transform\_perspective(roi\_image, M)

**def** compute\_transformation\_matrix(image, hough\_lines):

*"""Given the image and hough lines for lane lines in the image,*

*compute the transformation matrix to perform a perspective transform for*

*viewing the lane line from above.*

*Parameters:*

*image: image with hough lines drawn on it.*

*hough\_lines: Coordinates of the two lines*

*"""*

**if**(**not** hough\_lines):

**return** **None**, image

*# Decide the final hieght of the transformed points*

y\_limit = image.shape[0] \* 0.4

*# Determine the original and destination points based on the hough lines*

line1, line2 = lines

*# Top Right, # Bottom Right, # Top left, # Bottom left*

original\_img\_pts = [[line1[0], line1[1]], [line1[2], line1[3]], [line2[0], line2[1]], [line2[2], line2[3]]]

destination\_pts = [[line1[2], y\_limit], [line1[2], line1[3]], [line2[2], y\_limit], [line2[2], line2[3]]]

*# Define calibration box in source(original) and destination(warped/desired) image*

image\_shape = (image.shape[1], image.shape[0])

*# Four source coordinates*

src = np.float32(original\_img\_pts)

*# Four desired points*

dst = np.float32(destination\_pts)

*# Compute the perspective transformation matrix*

M = cv2.getPerspectiveTransform(src, dst)

*# Compute the inverse perspective transformation matrix*

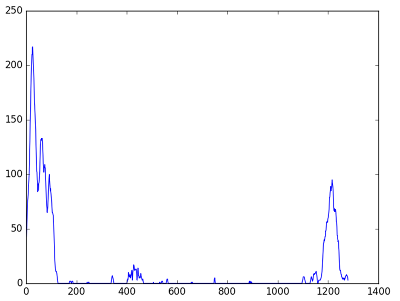
M\_inv = cv2.getPerspectiveTransform(dst, src)

**return** (M, M\_inv)

# Finding lane line pixels and fit position with a polynomial

The next step includes finding the Lane line pixels it fitting al together in a polynom. First of all, a histogram is used. Here, I had problems with my Python edition because somethings an format error appeard. The histogram is done with the following code:

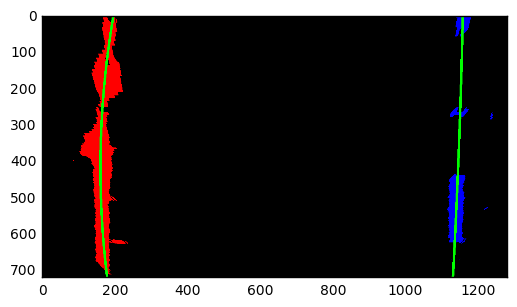
**def get\_lane\_lines\_base**(image):  
  
 histogram = np.sum(image[image.shape[0] // 2:, :], axis=0)  
 plt.imshow(histogram)  
 plt.show(histogram)  
  
 indexes = find\_peaks\_cwt(histogram, np.arange(1, 550))  
  
 **return** [(indexes[0], image.shape[0]), (indexes[-1], image.shape[0])]



With this histogram, the pixels values are added along teach column in the images. Starting at the base of each line, a sliding window is used to find the pixels belonging to that lane line. This technique is used for both the left and the right lane lines. With the pixel positions a curve is fitted into each of the left and right lane pixels (cv function polylines)

This is done by the fowling functions:

**def get\_lane\_pixels**(image, lane\_base):  
  
 window\_size = 100 \* 2  
  
 x\_base = lane\_base[0]  
  
 **if** (x\_base > window\_size):  
 window\_low = x\_base - window\_size / 2  
 **else**:  
 window\_low = 0  
  
 window\_high = x\_base + window\_size / 2  
  
 # Define a region  
 window = image[:, window\_low:window\_high]  
  
 # Find the coordinates of the white pixels in this region  
 x, y = np.where(window == 1)  
  
 # Add window low as an offset  
 y += np.uint64(window\_low)  
  
 **return** (x, y)  
  
**def get\_curved\_lane\_line**(pixels):  
 x, y = pixels  
 degree = 2  
 **return** np.polyfit(x, y, deg=degree)  
  
  
**def draw\_curved\_line**(image, line):  
 p = np.poly1d(line)  
 x = list(range(0, image.shape[0]))  
 y = list(map(int, p(x)))  
 pts = np.array([[\_y, \_x] **for** \_x, \_y **in** zip(x, y)])  
  
 pts = pts.reshape((-1, 1, 2))  
  
 cv2.polylines(image, np.int32([pts]), **False**, color=(255, 0, 0), thickness=50)  
 **return** pts



# Calculating the radius of a curvature and defining the position of the vehicle

If lanes are found, the curvature and position of the center of the car is calculated. Since the two lines are present, for the coefficient of the polynomial the mean value is used. The y coordinate at which the polynomials are evaluated is the bottom of the image. After that, the lines are drawn on a warped image and then unwarped and added to the original image. The last thing is to print out the values of curvature and offset of the center of the car. Here is the result for discussed case:

**def** calculate\_rad\_curvature(image, pixels):

*"""*

*Calculate the radius of curvature for the lane line*

*Parameters:*

*image*

*pixels: x,y coordinates of the pixels that belong to the lane line*

*"""*

y, x = pixels

*# Define conversions in x and y from pixels space to meters*

ym\_per\_pix = 30/image.shape[1] *# meters per pixel in y dimension*

xm\_per\_pix = 3.7/700 *# meteres per pixel in x dimension*

y\_eval = np.max(y)

fit = np.polyfit(y\*ym\_per\_pix, x\*xm\_per\_pix, 2)

**return** int(((1 + (2\*fit[0]\*y\_eval + fit[1])\*\*2)\*\*1.5) \

/np.absolute(2\*fit[0]))

In addition, the distance from the center is calculated:

**def** compute\_distance\_from\_lane(image, left\_base, right\_base):

*"""*

*Compute the distance of the car from the center of the lane*

*Parameters:*

*image*

*left\_base -- coordinates (x,y) of the base of the left lane*

*right\_base -- coordinates (x,y) of the base of the right lane*

*"""*

xm\_per\_pix = 3.7/700 *# meteres per pixel in x dimension*

image\_center = (image.shape[1]/2, image.shape[0])

car\_middle\_pixel = int((left\_base[0] + right\_base[0])/2)

**return** float("**{0:.2f}**".format((car\_middle\_pixel - image\_center[0]) \* xm\_per\_pix))

# Example Image

As the final step, the whole pipeline is exercised with different test files. The whole pipeline for processing one image can be found here:

**def process\_image**(image):  
  
  
 **try**:  
 blurred\_img = gaussian\_blur(image)  
  
 undistorted\_image = undistort(blurred\_img)  
  
 binary\_image = get\_binary\_image(undistorted\_image)  
  
 roi\_image = select\_region\_of\_interest(binary\_image)  
  
 persp\_transform = transform\_perspective(roi\_image, M)  
  
 lane\_base = get\_lane\_lines\_base(persp\_transform)  
  
 left\_base, right\_base = lane\_base  
  
 left\_pixels = get\_lane\_pixels(persp\_transform, left\_base)  
  
 right\_pixels = get\_lane\_pixels(persp\_transform, right\_base)  
  
 warped\_with\_lane\_lines, left\_curv, right\_curv, dist\_from\_center = draw\_lane\_lines(image, left\_pixels,  
 right\_pixels, left\_base,  
 right\_base)  
  
 lane\_lines = transform\_perspective(image=warped\_with\_lane\_lines, M=M\_inv)  
  
 final = weighted\_img(lane\_lines, image)  
  
 cv2.putText(final, "Lane Curvature: " + str(left\_curv) + " (m)", (100, 100), cv2.FONT\_HERSHEY\_SIMPLEX, 2,  
 (255, 255, 255))  
 cv2.putText(final, "Distance from center: " + str(dist\_from\_center) + " (m)", (100, 150),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 2, (255, 255, 255))  
  
 **except** Exception **as** e:  
 print(e)  
 **return** image  
  
 **return** final

An example of the final image can be found below:



# Pipeline for a video

The same pipeline is applied to videos.

**def process**(image):  
 **return** process\_image(image)

white\_output = 'p4.mp4'  
clip1 = VideoFileClip("./project\_video.mp4")  
white\_clip = clip1.fl\_image(process) #NOTE: this function expects color images!!

The video can be seen in this video: <https://youtu.be/DP5LBobb6ks>

# Discussion

First of all, the time for setting up the pipeline is far to little. I needed like two days to set up everything.

The following issues can be arising:

1. Speed is concern -> The hough transformation is a heavy resource function
2. Tuning The treshholds for each transform is consuming work
3. Light conditions changing: Sometimes Line lost (in challenge videos)
4. Smoothing technigques can be applied
5. More FPS can be generated by reducing the picture size
6. Use Neural Networks in the System