

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
2. Camera Calibration	3
3. Pipeline for single Images	4
a. Distortion Correction.....	4
b. Creating a binary Treshold image: Color Transformation, Gradient Transformation.....	5
c. Perspective Transformation.....	8
d. Finding lane line pixels and fit position with a polynomial	8
e. Calculating the radius of a curvature and defining the position of the vehicle	8
f. Example Image	8
4. Pipeline for a video	9
5. Discussion.....	9

1. 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 rubrics.

2. 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)

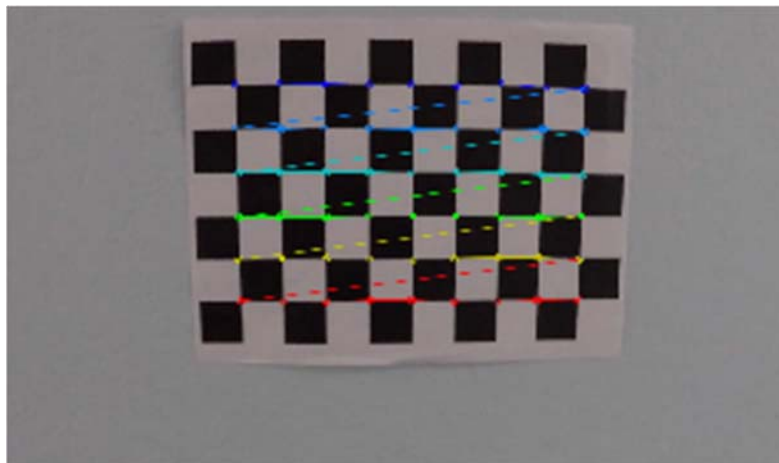
nx = 9
ny = 6
calibration_images = glob.glob("camera_cal/*")

# 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 picture is read with the `imread()` function. Then, the picture is turned into grayscale with the `cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)` function.

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



3. Pipeline for single Images

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

a. Distortion Correction

The function `_calibrate_camera` produces the variable „`mtx`“. This variable includes the camera coefficient matrix. In addition, the distortion points are given back. The code for the distortion 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])

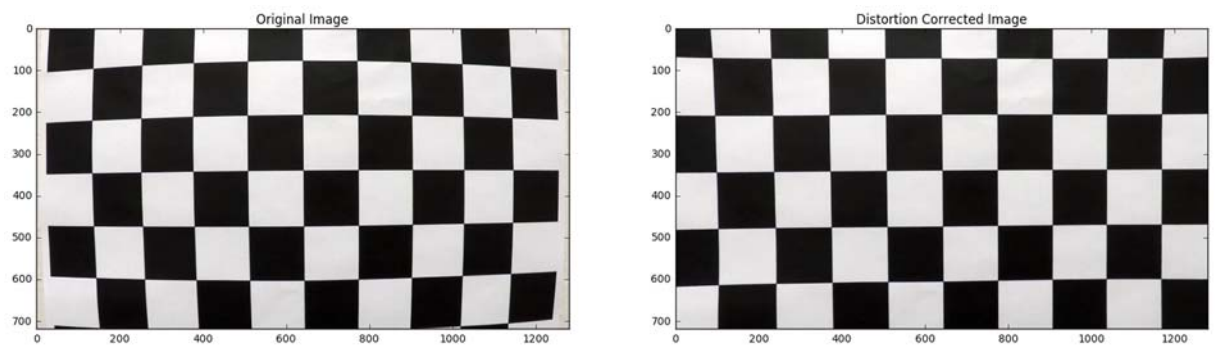
ax2.set_title(titles[1])
```

The distortion 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"])
```



b. Creating a binary Threshold 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 Threshold 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

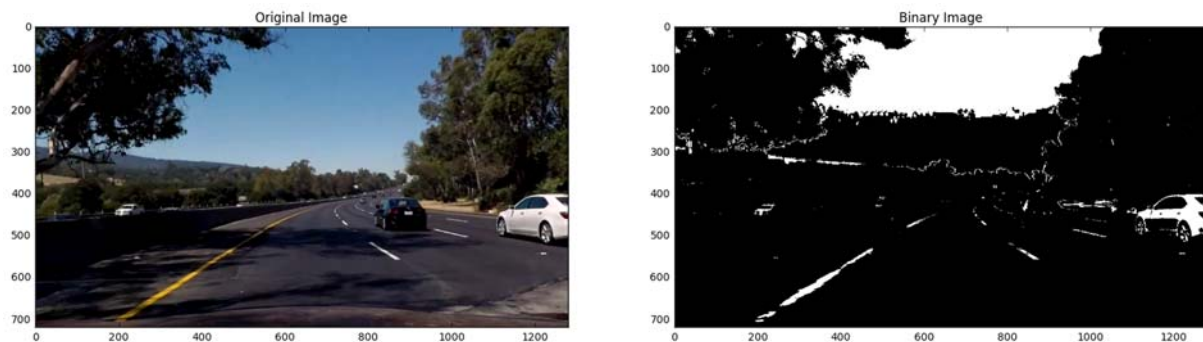
```

The complete binary image pipeline is displayed with 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")

```



- c. Perspective Transformation
- d. Finding lane line pixels and fit position with a polynomial
- e. Calculating the radius of a curvature and defining the position of the vehicle
- f. 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:



4. Pipeline for a video
5. Discussion