Udacity Self-Driving Car Nanodegree

Project 4- Advanced Lane Finding

Author: Johannes Betz

Date: 07/21/2017

Table of Contents

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 rubics.

2. Camera Calibration

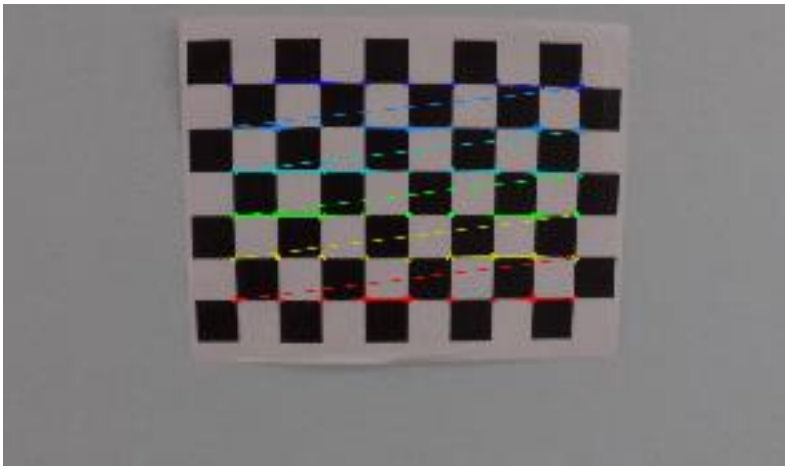
For the camera calibration 20 chessboard pictures are provided in the folder camera_cal. Each chessboard picture can be used for the camera calibration. The code looks like this:

```
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 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:



3. 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.

a. 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:

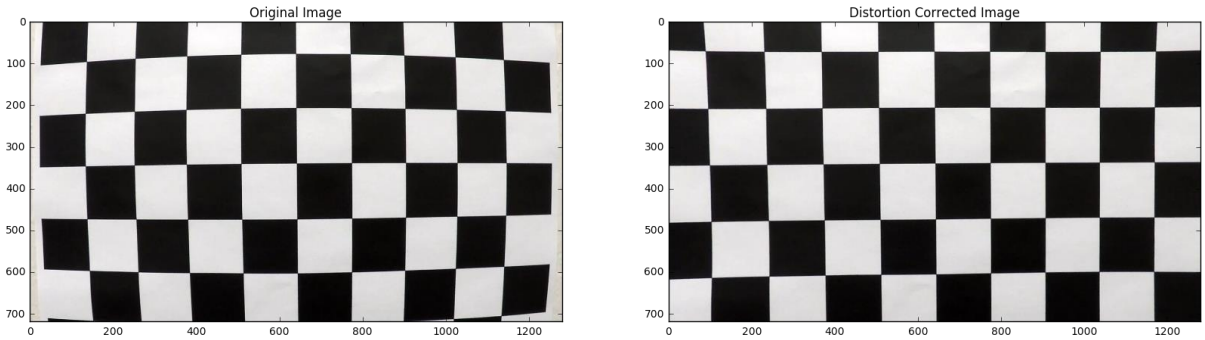
```
def undistort(image):

    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

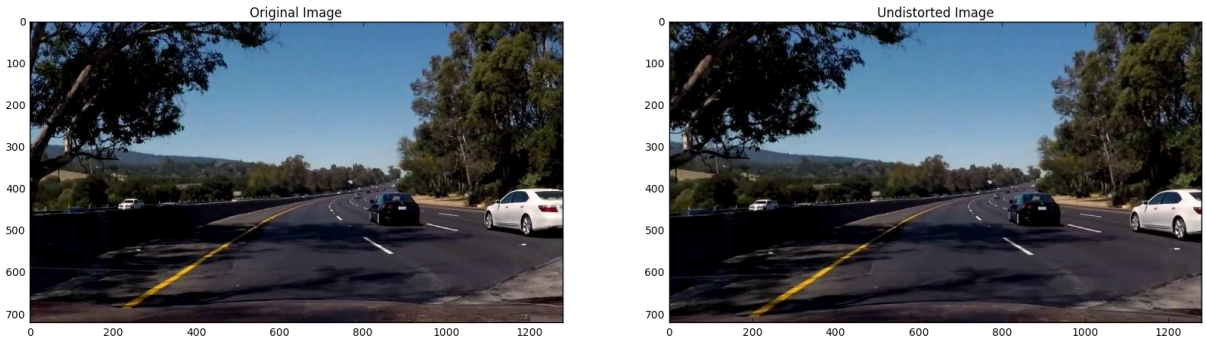
Example Image 1

```
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"])
```



b. 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:

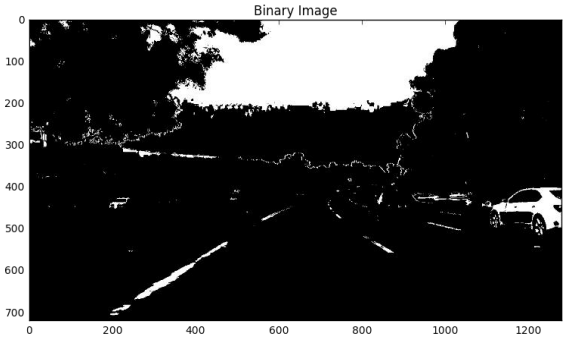
```
def get_sobel_binary(image, thresh_min=20, thresh_max=200):

    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")
```



c. 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)
```

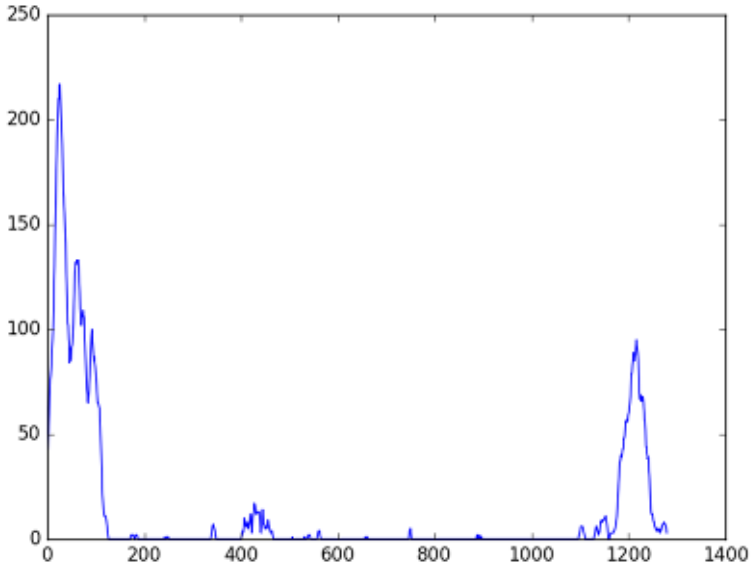
```
def compute_transformation_matrix(image, hough_lines):
    """Given the image and hough lines for lane lines in the image,
    compute the transformation matrix to perform a perspective transfor
m for
    viewing the lane line from above.

    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]], [li
ne2[0], line2[1]], [line2[2], line2[3]]]

    destination_pts = [[line1[2], y_limit], [line1[2], line1[3]], [line
2[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)
```

d. 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])]
```

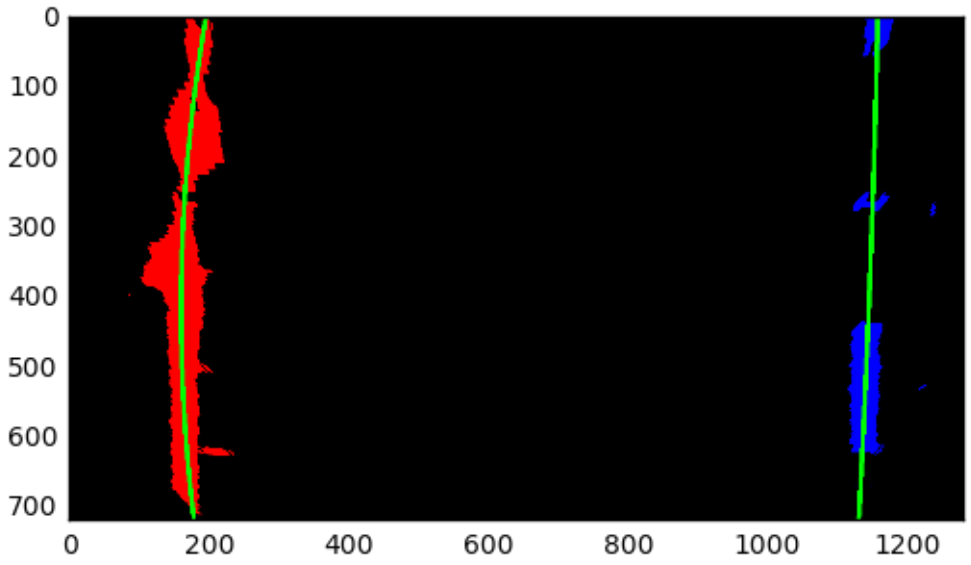


With this histogram, the pixels values are added along teach column in the images. Starting at the base of each line, a sliding window is used to find the pixels belonging to that lane line. This technique is used for both the left and the right lane lines. With the pixel positions a curve is fitted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



e. 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 la
ne 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 la
ne
    """

    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))
```

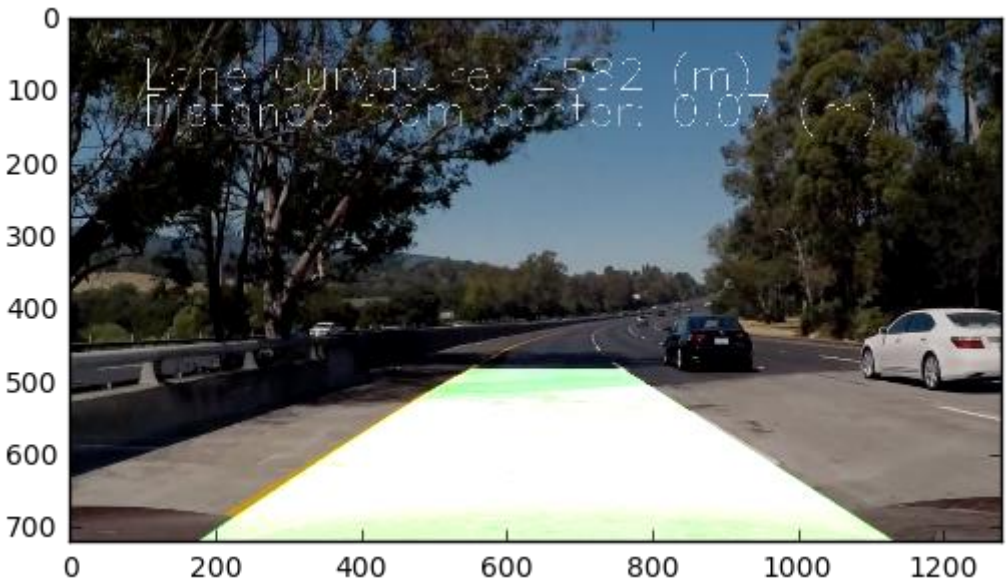
f. Example Image

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

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

5. 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