Project Advanced Line Finding Project

by Thomas Wieczorek, February 2017

Advanced Lane Finding Project

The goals / steps of this project are the following:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply a distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image ("birds-eye view").
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Rubric Points

Camera Calibration

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.

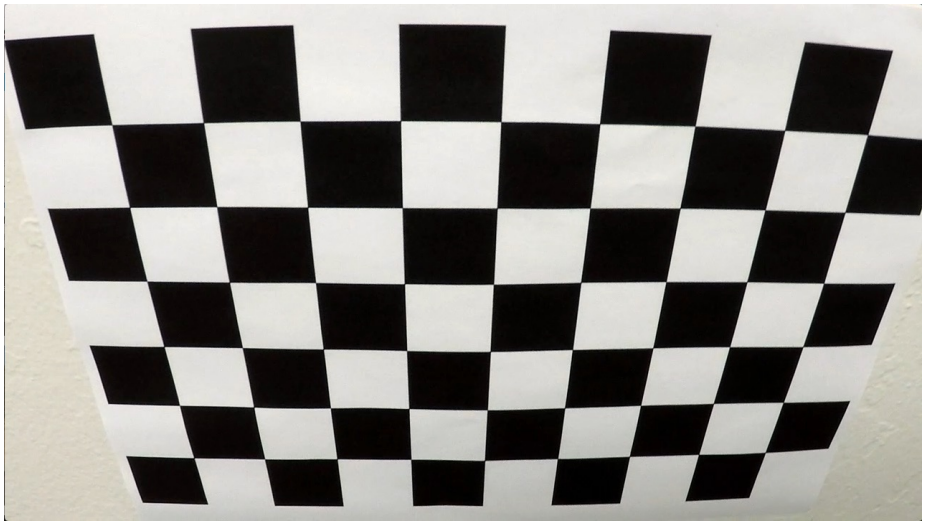
The code for this step is contained in the first code cell of the IPython notebook located in `implementation.ipynb` in the chapter `1. Camera calibration matrix and distortion coefficients given a set of chessboards`.

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.

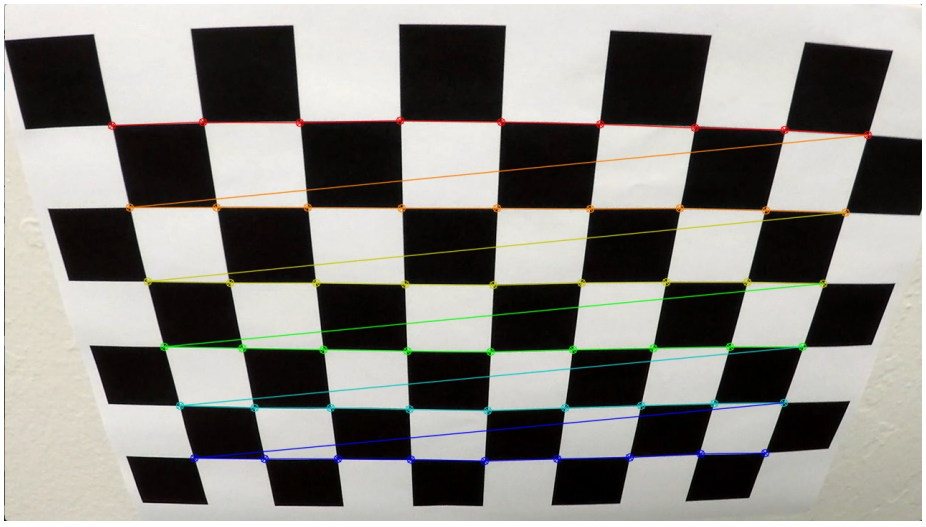
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function.

I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

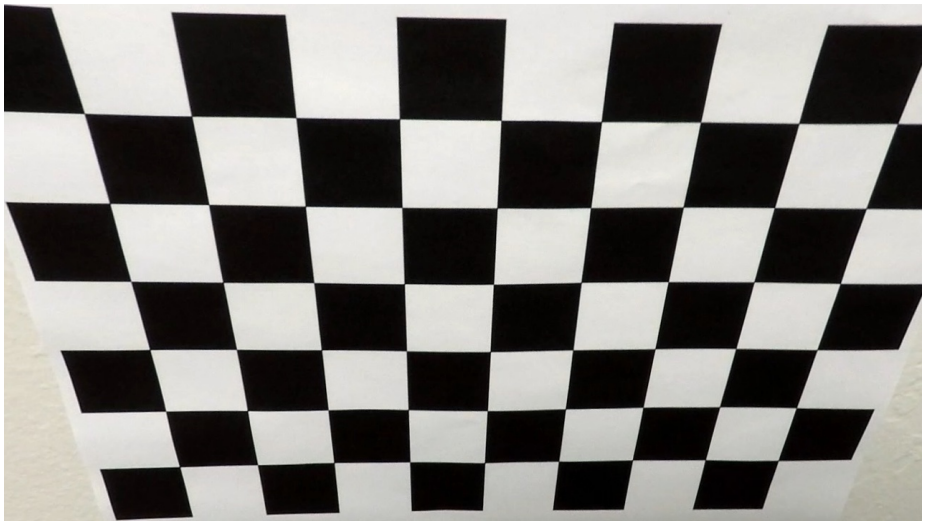
Original:



Marked corners:



Undistorted:



2. Apply a distortion correction to raw images.

Using the parameters from above, I applied the distortion correction on a raw image. The difference can be seen clearly at the tree on the upper left corner.

Original:

Undistorted:

Threshold binary image

3. Use color transforms, gradients, etc., to create a thresholded binary image.

I used a combination of color and gradient thresholds to generate a binary. By using trial-and-error I optimized the parameters:

```
def thresholded_binary(img):
    #grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    gradx = abs_sobel_thresh(img, orient='x', thresh_min=20, thresh_max=100)

    hls_binary = hls_select(img, thresh=(140, 255))
    hls_binary_l = hls_select_l(img, thresh=(80, 255))

    combined = np.zeros_like(gradx)
    combined[(((gradx == 1) | (hls_binary == 1)) & (hls_binary_l == 1))] = 1

    return combined
```

Here are some results of the binary thresholds on the test images:

As you can see, the road lines are always very visible with minor noise.

4 Apply a perspective transform to rectify binary image ("birds-eye view").

To get a birds-eye view, I used the method `cv2.getPerspectiveTransform` in addition to `cv2.warpPerspective`. Those methods need corners from the source-picture and destination picture, to where the pixels should be warped.

I used the Tool `Gimp` to determine the corners for the src-coners. I choose 4 corners for the destinations, which construct an even rectangle:

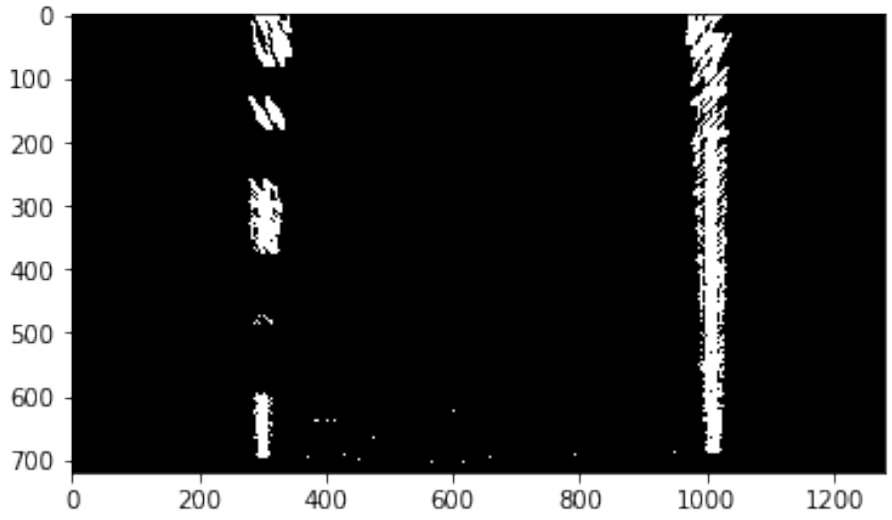
```
def birds_eye_view(img):
    corners = np.float32([[253, 697],[585,456],[700, 456],[1061,690]])
    top_left = np.array([corners[0, 0], 0])
    top_right = np.array([corners[3, 0], 0])
    offset = [50, 0]

    img_size = (img.shape[1], img.shape[0])
    src = np.float32([corners[0], corners[1], corners[2], corners[3]])
    dst = np.float32([corners[0] + offset, top_left + offset, top_right - offset, corners[3]

    M = cv2.getPerspectiveTransform(src, dst)

    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

    return warped
```

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

The following pictures show different image-steps: 1) The undistorted image 2) Bird Eye-View from undistorted image 3) Binary-Threshold image 2) Bird Eye-View from Binary-Threshold image

5. Detect lane pixels and fit to find the lane boundary.

To detect the lanes I first created a Bird Eye-View image from Binary-Threshold image (see 4.). The result looks like this:



Then I calculated a histogram and used argmax to determine two peaks - one peak left of the midpoint and one peak right of the midpoint:

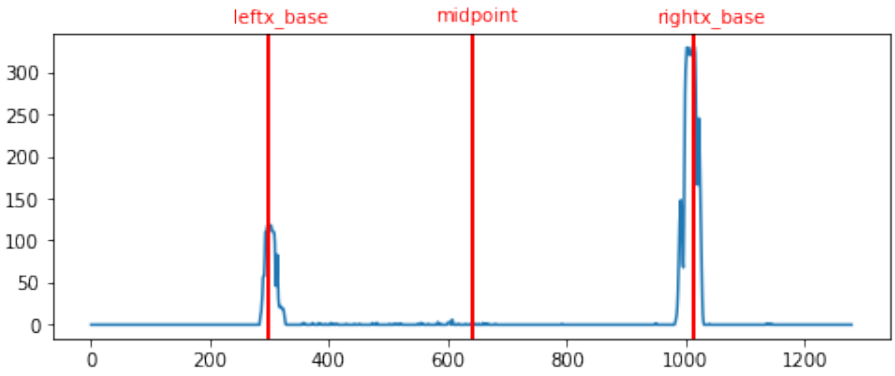
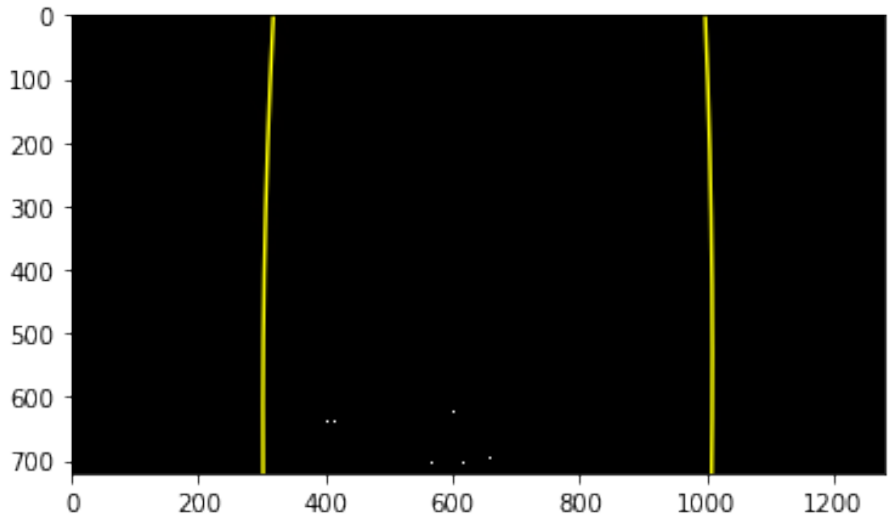


Figure 1:

Using a sliding window approach in combination with `np.polyfit` I calculated a quadratic equation to approximate the lines:



6. Determine the curvature of the lane and vehicle position with respect to center.

To determine the curvature of the line, I had to calculate the meters per picel in y and x dimension and convert the values in meter:

```
# 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
    y_eval = np.max(ploty)
```

Next I used this forumla to calculate the estimated radius of the curve:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

(based on http://www.intmath.com/applications-differentiation/8-radius-curvature.php)

Implementation of the formula in my system:

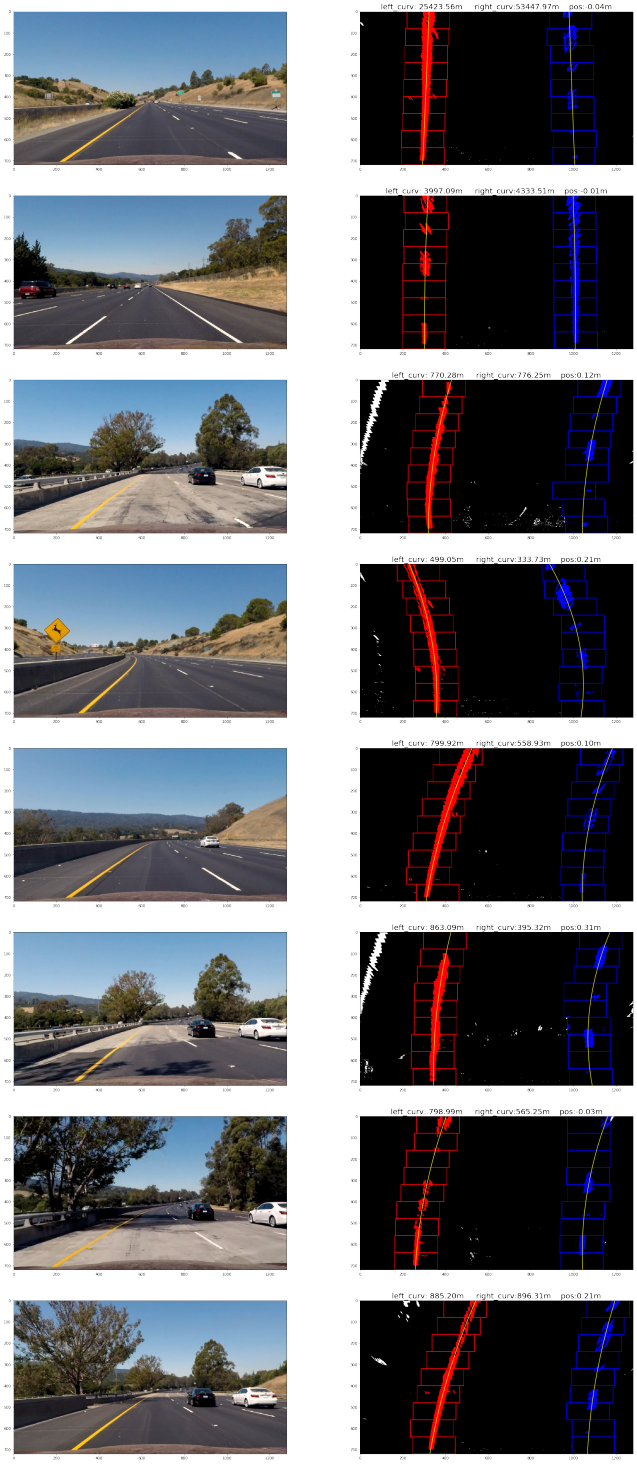
```
    # Fit new polynomials to x,y in world space
    left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
    right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)
    # Calculate the new radii of curvature
```

```
    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) /
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5)
```

To calculate the position of the car, I subtracted the center of the destination-corners (1061+253)/2 from the center between the lines (left_fitx[719] + right_fitx[719]).

```
center = (1061 + 253)/2 #center between the two destination corners (see bird_eye method)
    car_position = (((left_fitx[719] + right_fitx[719]) / 2) - center) * xm_per_pix
    print("left_fitx: {:.2f}px     right_fitx:{:.2f}px    out_img:{:.2f}px    car_position:{
```

The result:


left_curv: 25423.56m right_curv:53447.97m pos:-0.04m
left_curv: 3997.09m right_curv:4333.51m pos:-0.01m
left_curv: 770.28m right_curv:776.25m pos:0.12m
left_curv: 499.05m right_curv:333.73m pos:0.21m
left_curv: 799.92m right_curv:558.93m pos:0.10m
left_curv: 863.09m right_curv:395.32m pos:0.31m
left_curv: 798.99m right_curv:565.25m pos:-0.03m
left_curv: 885.20m right_curv:896.31m pos:0.21m

7. 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.

After combining all previous steps, I warped the detected lane boundaries back onto the original image, using the inverse PerspectiveTranformation (by switching the two parameters)

```
M_inv = cv2.getPerspectiveTransform(dst, src)
```

The final results of the test images looks like this:

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a video of the system in action:

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

My approach was a lot of trial and error. I faced several issues, for example shadows or other markings disrupted the algorithm. I was able to stabilize the system by comparing the current image with historic images and their values. If the

difference of the lines is over my threshold MAX_ALLOWED_DIFFERENCE, the current image and line-values will be replaced by the last valid historic values. This worked way better!

```
MAX_ALLOWED_DIFFERENCE = 50
def smooth_trough_history(ploty, l_x, r_x):
#...
    #skip l_x and r_x and use the values of the last frame,
    #where the difference is too big to the last picture (to avoid jumping-errors)
    if historic_count >= NUMBER_OF_HISTORIC_IMAGES:
       for k in range(720):
           if (l_x[k] > (Historical_Lines[0][1][k]+MAX_ALLOWED_DIFFERENCE)) or \
                   (l_x[k] < (Historical_Lines[0][1][k]-MAX_ALLOWED_DIFFERENCE)) or \
                   (r_x[k] > (Historical_Lines[0][2][k]+MAX_ALLOWED_DIFFERENCE)) or \
                   (r_x[k] < (Historical_Lines[0][2][k]-MAX_ALLOWED_DIFFERENCE)):

                   return Historical_Lines[0][0], Historical_Lines[0][1], Historical_Lines[0

#...
```

Another minor issue was bugging me: The lines were calculated for every frame, so there were 30 lines per second. This caused the lines to wobble a lot. To smooth the lines, I always calculated the mean over the last 5 images. This looked way better.

```
#calculate mean ploty
        ploty = np.zeros_like(ploty);
        for history in range(NUMBER_OF_HISTORIC_IMAGES):
            for k in range(720):
                ploty[k] += Historical_Lines[history][0][k] / NUMBER_OF_HISTORIC_IMAGES

        #calculate mean l_x
        l_x = np.zeros_like(l_x);
        for history in range(NUMBER_OF_HISTORIC_IMAGES):
            for k in range(720):
                l_x[k] += Historical_Lines[history][1][k] / NUMBER_OF_HISTORIC_IMAGES

        #calculate mean r_x
        r_x = np.zeros_like(r_x);
        for history in range(NUMBER_OF_HISTORIC_IMAGES):
            for k in range(720):
                r_x[k] += Historical_Lines[history][2][k] / NUMBER_OF_HISTORIC_IMAGES
```

The system works quite well, however with a lot of different shadows, quickly changing light conditions it will break. Also it breaks if the light is not clearly visible (for example because of a car in front, or because simply no line exists).

Some ways to improve the system would be: - using a mask - optimize the

parameters (for example for the binary-threshold) by trial and error - using Deep Learning like in the behavorial-project - Better combination of historic frames with the current frame