Project 2: Advanced Lane Finding

The goals / steps of this project are the following:

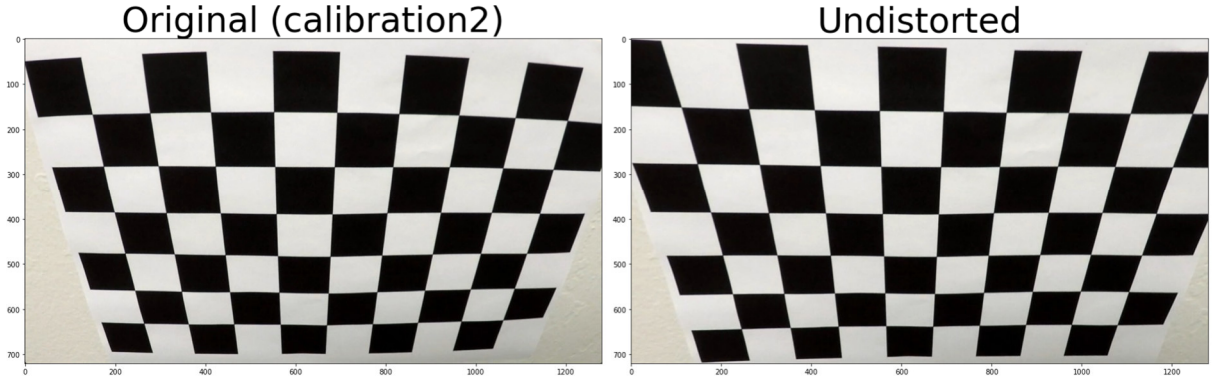
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.

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 "./CarND-Advanced-Lane-Lines/Project 2 – Advanced Lane Finding Single Steps.ipynb"

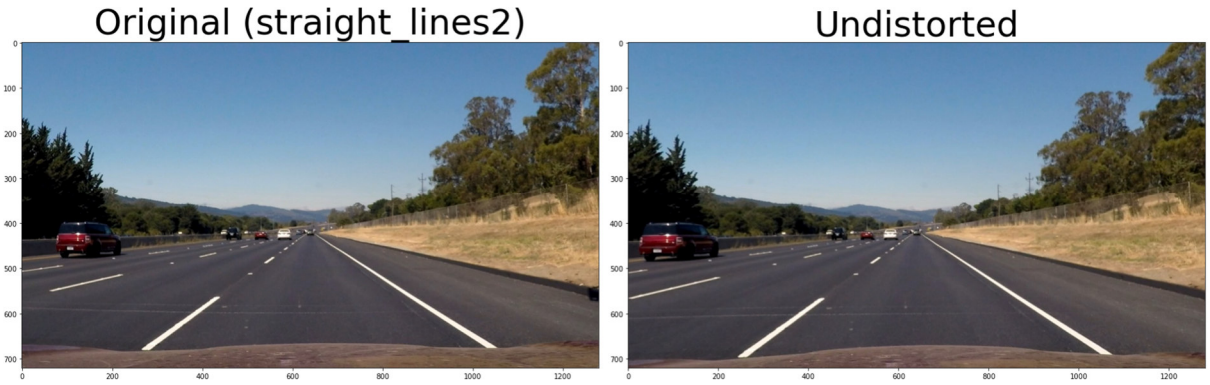
I start by preparing "object points" which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates 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:



2. Apply a distortion correction to raw images

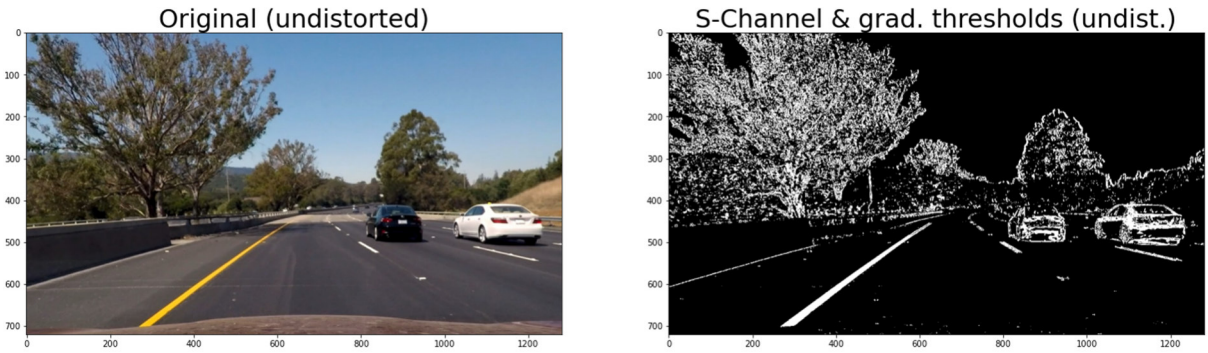
The procedure described above under point 1 led to the following result:



All the other undistorted images of `calibratrion1- calibration 20` are stored as `undist**.png` in the output_images folder.

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 image. Here is an example of my output for this step:



All the other result binary images (`CoTrGr**.png`) are stored in the output_images folder.

First I converted the RGB image to an HSL image (3th cell, line 21,22) because its S channel is doing a fairly robust job of picking up the lines.

The second step was to take the derivative in x ($Sobel_x$ Operator) in a grayscaled image (line 30-38).

I chose thresholds between 20 and 100 for the x gradient (Sobel) and color thresholds between 170 and 255 for the S-Channel of the HSL color space.

In the last step I combined the two binary thresholds (line 51-52)

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

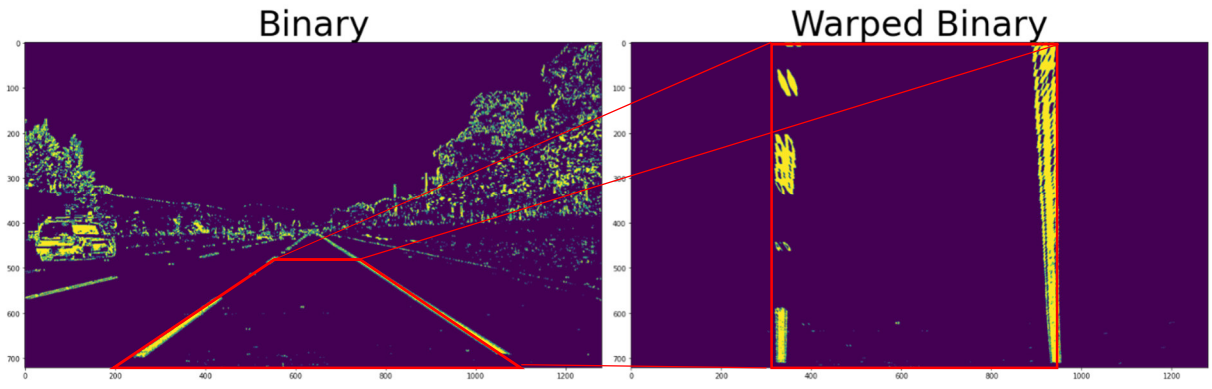
The code for my perspective transform includes a function called warper. The `warper()` function takes an image (img) as input. First, I had to define source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

```
# Defininig 4 source points dst = np.float32([[,],[,],[,],[,]])
src = np.float32([
            [200,720],      # left bottom
            [574,460],      # left top
            [715,460],      # right top
            [1120,712]])    # right bottom

# Defininig 4 destination points dst = np.float32([[,],[,],[,],[,]])
margin = 320
dst = np.float32([
            [margin,img.shape[0]],                  # left bottom
            [margin,0],                             # left top
            [img.shape[1]-margin,0],                # right top
            [img.shape[1]-margin,img.shape[0]]])    # right bottom
```

With the source and destination points and the resulting matrix M out of the `cv2.getPerspectiveTransform(src, dst)` function, the image can be warped by using the `cv2.warpPerspective()` function.

I verified that my perspective transform was working as expected by drawing the src and dst points onto a binary test image and its warped counterpart to verify that the lines appear parallel in the warped image. Here you can see the result:



The farther I choose the both upper points (top left and top right) on the road, the worse the quality will be when warping.

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

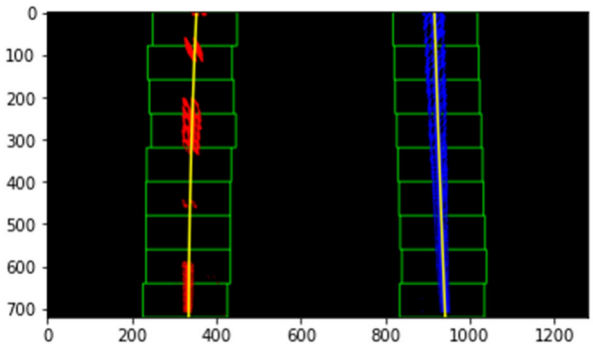
5.1. Sliding window

By using a histogram to add up pixel values along each column of my warped binary image, I could identify two peaks – one on the left half of the image and one on the other half. I used them as my starting points to search for the lines at the bottom of the image.

Then I defined 9 windows on every side in which to search for the lane line pixels. Finding pixels in the bottom window leads to the sliding of the next window in direction of the pixel weighting.

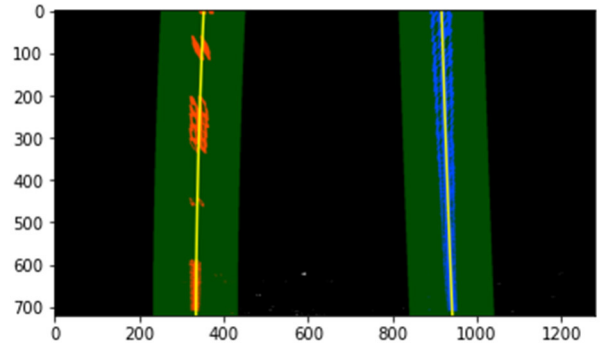
Now that I have found all the pixels belonging to each line through the sliding window method, the next step was to fit a second order polynomial to the line.

The next picture is a visualization of the sliding windows with both fitting lines (yellow):



5.2. Search from Prior

Because of inefficiency of this “blind search”, I used the `search_around_poly()` function. The functions search for pixels just around a margin around the previous found lines.



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

The algorithm is implemented in the `measure_curvature_real()` function in the 7th cell. First it defines the conversion factor from pixels to the real scale. Line 49 and 50 calculate the radius of both lines found under point 5. Both curverads are half half weighted combined to the Lane Curverad value:

```
Left Curverad:  5397.22902377388 m Right Curverad:  22556.19559340398 m
Lane Curverad:  13976.712308588929 m
Vehicle position with respect to lane center:  0.011333546926740415 m
```

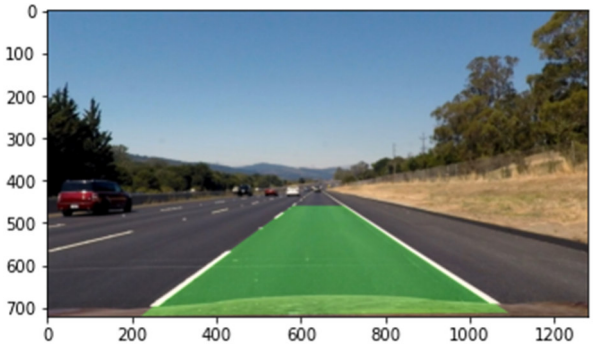
The vehicle position is calculated by the conversion factor too. Taking into account that the center of the image corresponds with the center of the vehicle, the distance from the center of the image to the center of the lane corresponds with the vehicle offset (7th cell line 68 – 74).

```
Vehicle position with respect to lane center:  0.0113335469268 m
```

7. Warp the detected lane boundaries back onto the original image

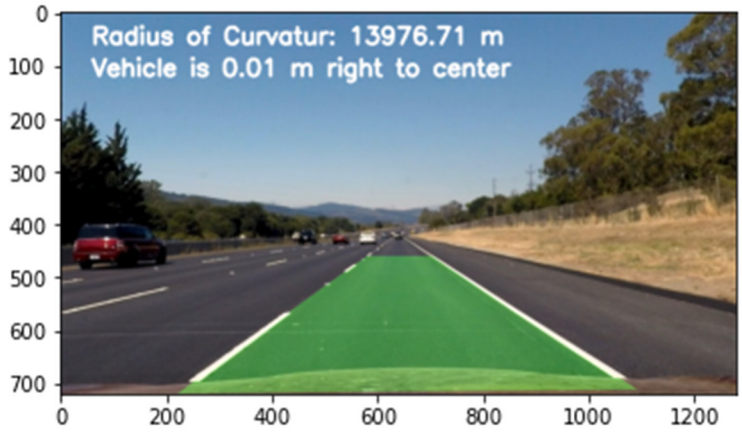
I warped the two lines back in the same way shown under point 4. Only difference is that source points and destination points change their values, so that we use in the `cv2.warpPerspective()` function the inverse Matrix M_{inv}.

Here is the result on a straight_lines2 testimage:



8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position

With function `cv.putText()` I added the information I got out of point 6 (9th cell), taking into account whether the vehicle is on the left or right of the center of the lane.



Pipeline (video)

The video result (project_video_s_thresh_100-200full.mp4) is stored in the test_videos_output folder.

For video analyzing I changed the code (line 213 – 224 in "Project 2 - Advanced Lane Finding version 2.ipynb"). My original code (first submission) could not find the right lane line exactly. Therefor I introduced the global variables "`last_good_left_fitx`" and "`last_good_left_fitx`".

The lane width of 3.7 meter corresponds to roundabout 600 pixels in the image / frame.

So if every value met the condition `right_fitx` > `left_fitx` + 500, the algorithm stored all the current values of `left_fitx` in `last_good_left_fitx` and all the current values of `right_fitx` in `last_good_right_fitx`.

If the condition `right_fitx` > `left_fitx` + 500 is not met, `right_fitx` and `left_fitx` take on the last stored values `last_good_right_fitx` and `last_good_left_fitx`.

Discussion

My algorithm with the condition `right_fitx` > `left_fitx` + 500 (described in the previous chapter) works well if the problem with lane line finding applies only to a few frames. But I fear this approach becomes more critical the more frames there are in which the line is not detected.

Another weak point of this approach is when the track width becomes significantly narrower. A corresponding adjustment of the condition is required here.

By using the `cv2.warpPerspective()` function I chose a y-value of 460 for the both 'top'-points (left and right). That seems like a good compromise to me. With smaller values in the direction of image.shape [0]/2, I fear that the transformation errors will be too large. I should also have problems here with very small curve radii, as the sliding-windows in the upper area of the warped binary image will no longer find any lane lines.

If the y-value is too high, I fear that there will be too great difficulties in detecting lane lines in the event of lane line gaps.