Advanced Lane Finding

Report on Udacity CarND Project No. 4, Nov. cohort 2016

Albert Killer, Feb 2017

1 Project assignments

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.

The entire Python code containing above mentioned steps is provided in following file:

`p4-adv-lane-finding.py`

To address each task the code is structured into various functions (see list below). These functions are used by the main function `main_pipeline`, which itself is called within the main section after deciding between image input either from a local folder (i.e. `test_images`) or video file (i.e. `project_video.mp4`).

`plot_results`

`calibrate_camera`

`thresholding_pipeline`

`region_of_interest`

`perspective_transform`

`sliding_windows`

`fit_polynomial`

`measure_curvature`

`project_lanelines`

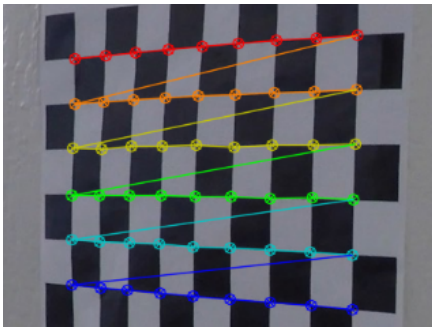
`main_pipeline`

`process_video`

The following approach is inspired by Udacity [1] and code provided by Udacity within the lessons was used and modified for this project's purpose.

2 Camera calibration

To get realistic shapes and sizes of all the objects shown in the images the input data must be undistorted. Therefor the process of "camera calibration" is applied. We import a collection of calibration images showing a chessboard pattern captured by a camera from different angle inducing various radial and tangential distortion types.



The library OpenCV is used to search for the chessboard's corners assuming a predefined pattern (i.e. 9 squares horizontal, 6 squares vertical). After storing object points (3d points in real world space) and image points (2d points in image plane) of each detected chessboard pattern in to defined arrays, camera calibration is processed using `cv2.calibrateCamera()`. The results are later on used within `main_pipeline` to undistort imported image data by using the correct camera matrix `mtx` and distortion coefficients `dst`.

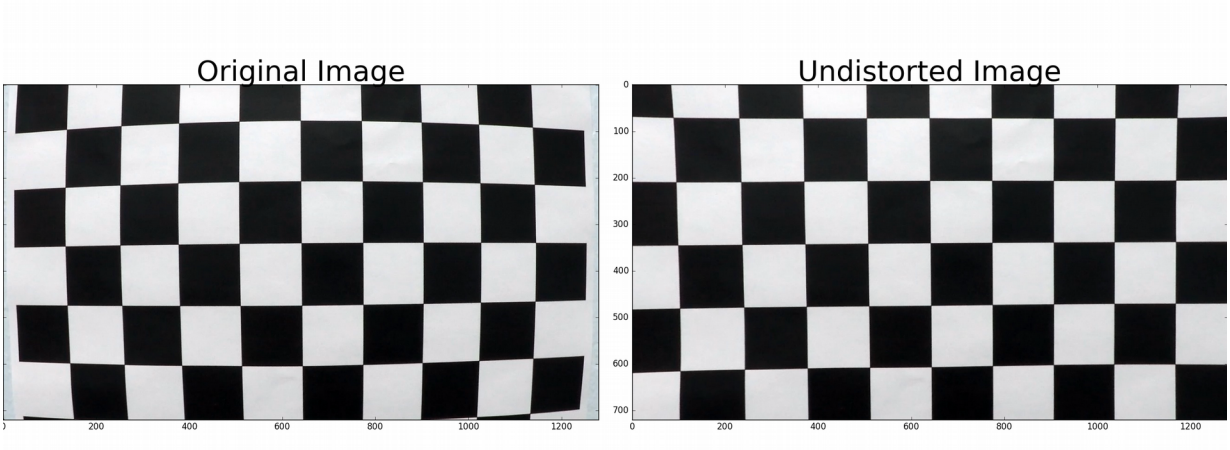


Figure 1: Using the results of camera calibration for removing distortion.

Camera calibration only has to be executed the first time the code is run. Both camera matrix and distortion coefficient are saved into pickle files for the following steps applied on several images.

3 Pipeline

3.1 Distortion Correction

The first step after importing images within `main_pipeline` is to remove distortion by using the parameters calculated while camera calibration (cp. section 2 above). Comparing the undistorted image with the original data, differences can only be recognized at the edges but have an direct influence on the precision of our later measurements of lane lines (Figure 2 below).

Figure 2: Removing distortion from input image data.

3.2 Binary Image

In the next step we want to figure out the position of the lane lines by marking them clearly in an binary image output. Therefor we use different thresholding methods on the undistorted image, described in the `thresholding_pipeline` function. Similar to Canny edge detection we apply the Sobel operator, which is a way of taking the derivative of the image in each direction and therefore highlighting the edges closer to vertical or horizontal. After this only pixels within a certain threshold of `sxy_thresh=(20, 100)` are getting included.
To improve the results under different light and color conditions we combine the directional gradient output with a color channel threshold. After converting the input image to HSV color space, the saturation channel (S) is extracted and filtered by applying `s_thresh=(90, 255)`.

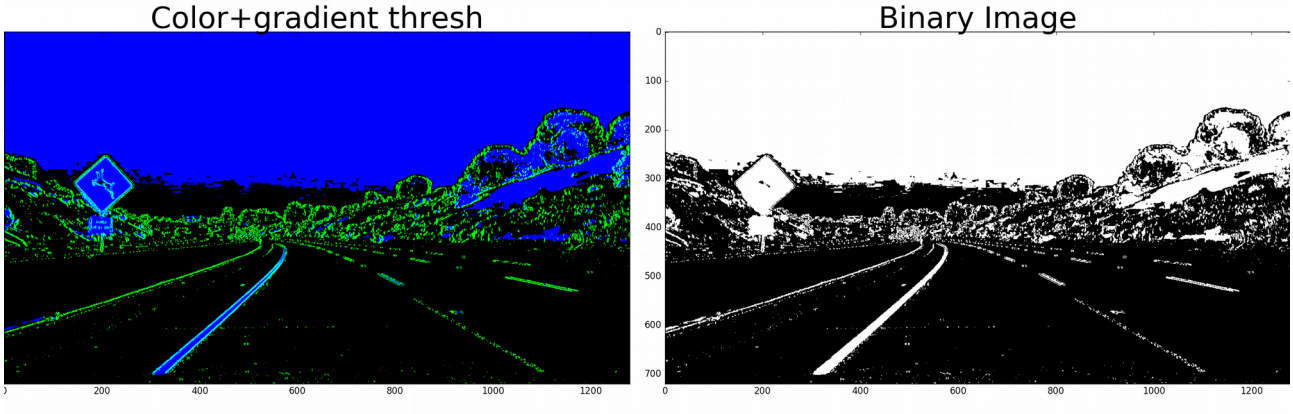


Figure 3: Combining color and directional thresholding.

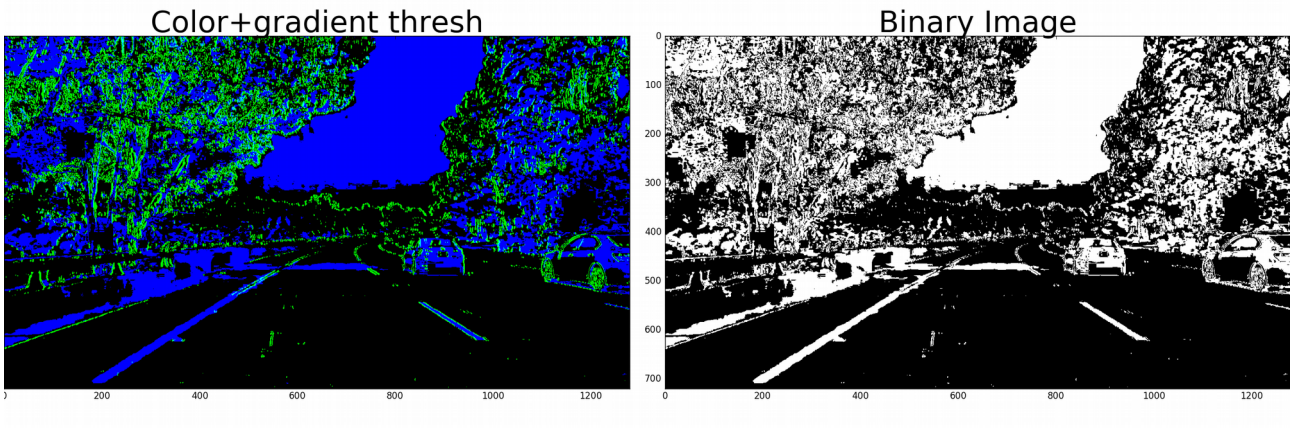


Figure 4: Combining color and directional thresholding.

While experimenting on different input images using a set of additional methods like Sobel operator on y-direction, gradient magnitude, gradient direction and the combinations of each, as a matter of fact best results were acquired by only using the before mentioned directional gradient (x-direction) and color thresholding (cp. Figure 3 and 4, above).

3.3 Perspective Transform

One goal of this project is to measure the curve radius of the lines making their way on the road. We can only do this appropriately if we warp a certain region of the input image to a bird's eye view perspective (cp. Figure 5, 6 and 7 below). After testing results on an example with straight lines, the following points turned out to be feasible for the perspective transform of all test images (Table 1)**.**

corner	x, source	y, source	x, destination	y, destination
top_left	540	480	160	0
top_right	754	480	718	0
bottom_right	1190	718	1190	718
bottom_left	160	718	160	718

Table 1: Corner points marking a trapezoidal shape within 1280x720 pixel image.

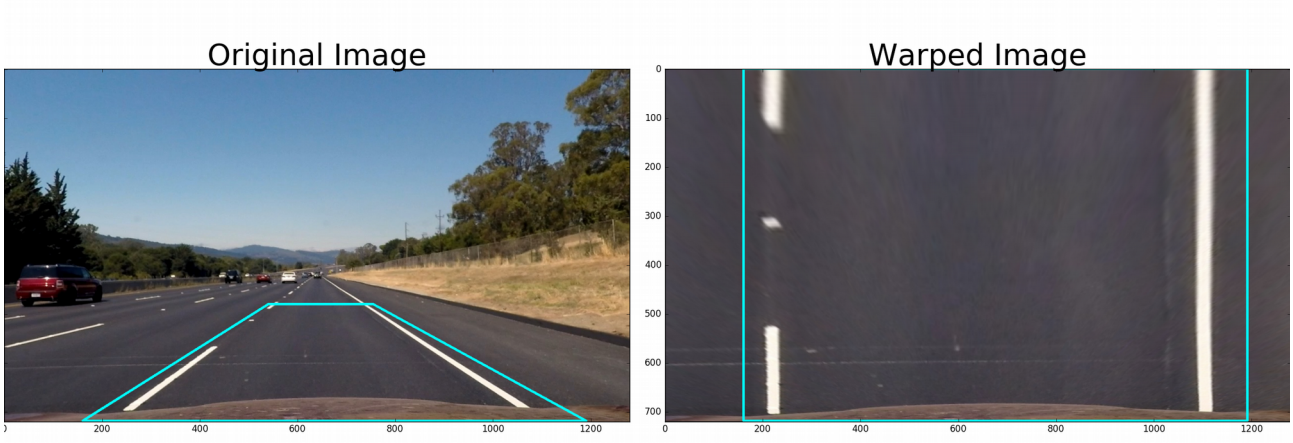


Figure 5: Result of perspective transform on road with straight lines.

Figure 6: Result of perspective transform on road starting a curve.

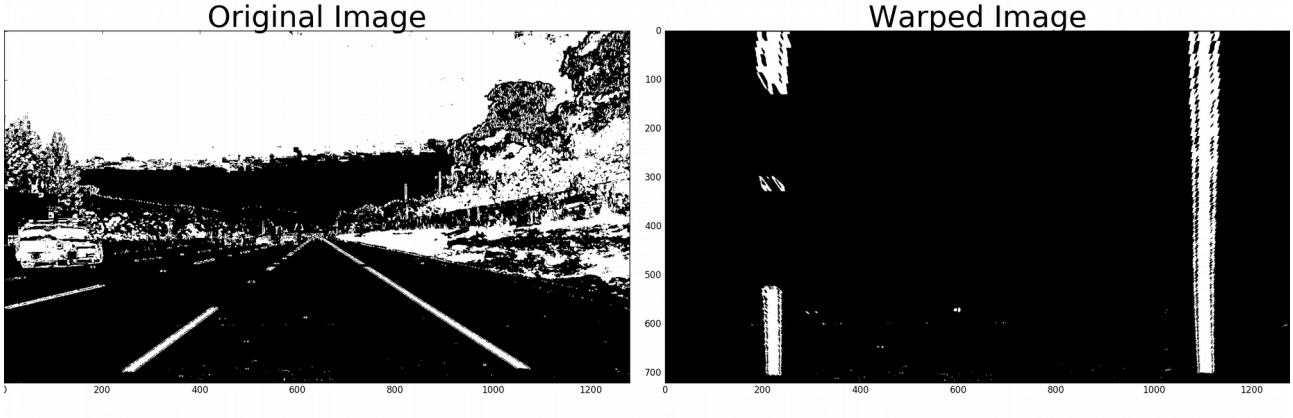


Figure 7: Result of perspective transform on binary image.

The function `perspective_transform` uses OpenCV's `cv2.getPerspectiveTransform()` and `cv2.warpPerspective()` to warp the images. The same function is used later on to reverse the transform and put the visualized results back on the original image's road.

3.4 Identify Lane Lines

The rectified version of the binary image is now taken to measure curvature and the car's distance from the center of the lane. Two functions were created on basis of code taken from the Udacity lesson [1]: `sliding_windows`. By using a histogram to figure out the x-coordinates with the highest density of line pixels, we are able to define starting points for the following search. Small rectangles/windows are set to scan the image for non-zero pixels (=line pixels). During this process the window's position starts with the center of the last one and gets its own position updated according to detected non-zero pixels. Figure 8 shows the applied windows (green rectangles) and respective center (yellow lines) where the center of the lane lines was identified.

Both lines are separated into left (marked red) and right (marked blue) lane line.
As a result we get two second order polynomial curves defined by the located lane line pixels:

$$f(y) = A\,y^2 + B\,y + C$$

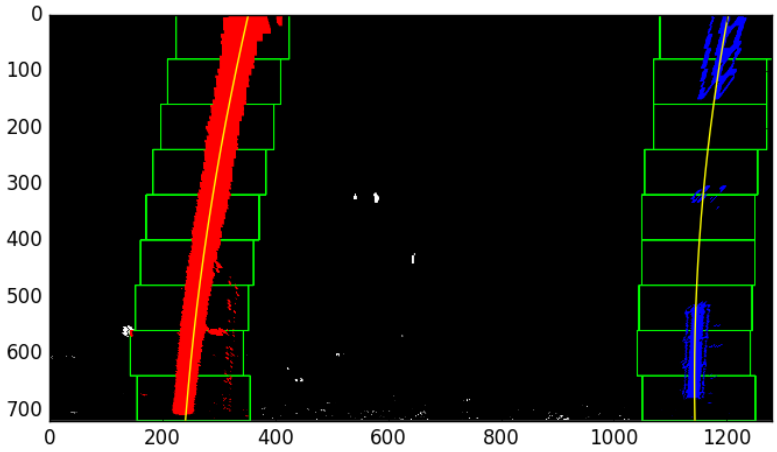


Figure 8: Output of `sliding_windows`.

In addition another function was drafted (`fit_polynomial`), inspired by Udacity [1], in order to accelerate the detection of the lane lines position for processing a video stream for example (see also: Discussion below).

3.5 Measure Curvature

With the lane lines identified, now we can estimate the radius of curvature of the road. Therefore we define the y-position where we want to measure the radius and apply following equation [2].

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

After acquiring the curve radii of both lane lines we convert the respective x- and y-values from pixel to meter, assuming a road designed by U.S. government specifications for highways [3].

```
ym_per_pix = 30 / 720  # max px in height
xm_per_pix = 3.7 / 1030  # 160 to 1190 = 1030 p
```

To get the car's distance from center we further assume the camera to be mounted at the center of the car. Then we only have to get the center of the lane according to the calculated lane line curves in order to calculate the deviation.

```
lane_center = leftx[0] + ((rightx[0] - leftx[0]) / 2)
camera_center = input.shape[1] / 2
distance_center = (camera_center - lane_center) * (3.7 /
                  (bottom_right[0] - bottom_left[0]))
```

Finally both, radius of curvature and position of the vehicle with respect to the center of the lane, are displayed on the output image.

3.6 Result

To visualize the results we use the function `project_lanes` to mark the identified lane lines on to the road and mark the area of the detected lane. Therefor the lines are drawn into an empty image. Afterwards we use the `perspective_transform` function to warp the rectified image back to its original perspective and combine it with the original image data (see Figure 9 and 10 below). This way we can use the output for real-time verification and measurement with a video stream for example: output of `project_video.mp4` can be found in folder `output_images`.

Figure 9: Output with identified lane, curvature and distance from center, `test2.jpg`

Figure 10: Output with identified lane, curvature and distance from center, `test5.jpg`

4 Discussion

The first challenge was to define appropriate points for the trapezoidal shape which is used for the perspective transform as not every approach turned out to be feasible for every test image. Small changes have huge effects on the curvature of the lines in the rectified version of the image and therefore compromise the precision of the output very easily. This choice could be made less relevant by introducing respective filters after perspective transform i.e. additional regional masking. I implemented a function from Project no. 1, but still have to experiment with the integration into the pipeline.
Testing the challenge videos showed there is still fine tuning to be done in terms of creating the binary image via thresholding. Especially experimenting with different color spaces and channels seems to be promising in order to be able to deal better with shadows. Those produce changes in brightness and color which are causing the most detection errors while testing the challenge videos.
Furthermore the code can be optimized i.e. to properly use the `fit_polynomial` function: the `sliding_windows` function should only be used for the first frame, afterwards the results from the previous line detection can be used with `fit_polynomial` to get changes of the lines much more effectively. Other than that there are various functions to be integrated to enhance the performance and robustness of the lane line detection: Filtering noise in the binary image, applying a “sanity check” to separate out false detections with huge deviation compared to previous frames and introducing a smoothing method to avoid wobbling of the visualization of the lane area.
The applied methods demonstrate impressively the crucial information we can get only from putting a single camera’s input through a software pipeline. It also shows how easily those methods are influenced by small decisions in software design, which can cause huge effects on very important control decisions like steering. Therefor these techniques have to be part of a redundant system of cameras and sensors and maybe even cross-check with output provided by models trained with advanced machine learning methods as explored in the previous project.

References

[1] Udacity Self-Driving Car Engineer Nanodegree Program. *Term 1, Lesson No. 18: Advanced Lane Finding.*
URL: https://www.udacity.com/drive

[2] M. Bourne. *Interactive Mathematics: Applications of Differentiation/8. Radius of Curvature.*
URL: http://www.intmath.com/applications-differentiation/8-radius-curvature.php

[3] Texas Department of Transportation. *Roadway Design Manual: 2. Basic Design Criteria/4.Horizontal Alignment/Curve Radius.*
URL: http://onlinemanuals.txdot.gov/txdotmanuals/rdw/horizontal_alignment.htm#BGBHGEGC