Advance Lane Lines Project Report

Alberto Rivera
April 22, 2017

Self Driving Car Nanodegree

Domain Background

This writeup and the accompanying files are part of Project 4 from Term 1 of the Udacity Self Driving Nanodegree program. The purpose of this project is to correctly identify lanes on a road and their curvature. This is essential information for self-driving cars when calculating steering angles.

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 ("bird's-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 template for this project can be found on the following Github repository:
https://github.com/udacity/CarND-Advanced-Lane-Lines

My implementation of this project can be found on the following Github repository:
https://github.com/alberto139/CarND_Advanced_LaneLines
Note that the ‘data’ folder and ‘run1.mp4’ have been removed from the repository because of size constraints.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Camera Calibration and Distortion Correction

The purpose of calibrating the camera is to later undistort the image.

Calibrating the camera and undistorting the image is important because camera lenses capture images that are slightly different form their actual representations. Having an undistorted image is important because our calculations could be used in a real vehicle, so making those calculations according to representations which are as close as possible to the real world is important.

To calibrate the camera we will use a function provided by OpenCV.
`cv2.calibrateCamera(objp, imgp, img_size, None, None)`

This function takes in objectPoints and imagePoints as the first two parameters. Each of these is a vector of vectors of calibration pattern points in the calibration pattern coordinate space. For further information look at the OpenCV documentation

To obtain the objectPoints and imagePoints we will use the finding chessboard corners technique implemented in the next couple of cells. The image points are returned by the `findChessboardCorners` function also from OpenCV.

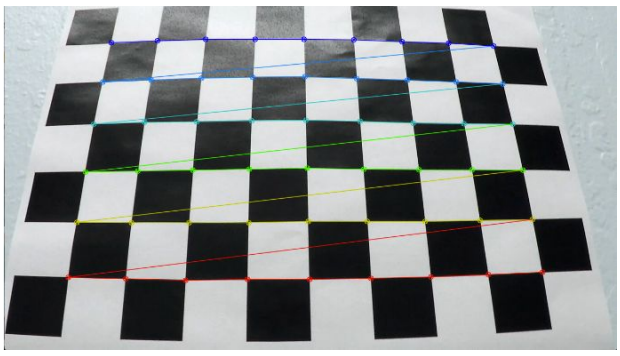

Figure 1. Result of `findChessboardCorners`

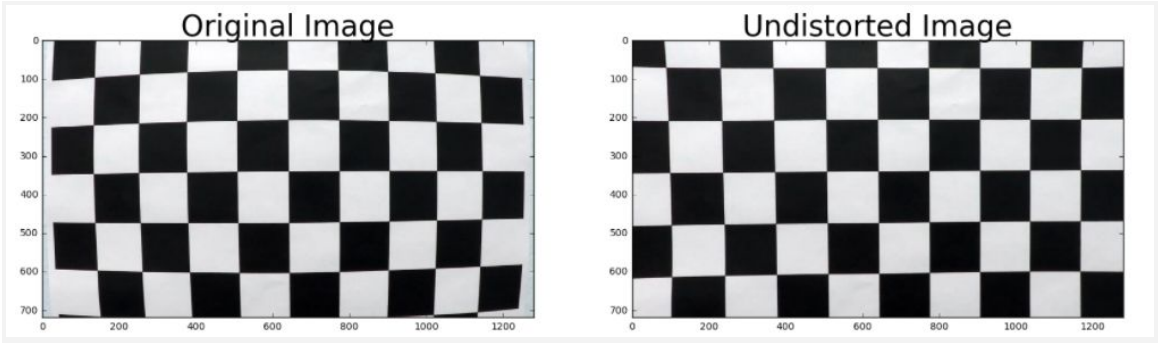

Figure 2. Image used for calibration (Original Image) compared to the result of undistorting it (Undistorted Image)

Figure 3. First frame of the project video (Original Image) and the result of undistorting the image (Undistorted Image).

Gradient and Color Thresholds

Using thresholds is useful in identifying edges in a image and therefore objects and lane lines. In our case we used a combination a gradient and color thresholds to transform the image into a version of itself that will make it easier to identify lane lines later on. Figure 4 details the function used to perform this transform and Figure 5 shows the result of of running the first frame of the source video through the function.

```
def thresholded_binary(undistorted_img):
    # Magnitude Threshold
    mag_binary = mag_thresh(undistorted_img, sobel_kernel=5, mag_thresh=(75, 255))

    # Convert to HLS color space and separate the S channel
    s_binary, s_channel = color_thresh(undistorted_img, color_space = 'hls',
channel = 's', vis = True, thresh=(170,255))

    # Stack each channel to view their individual contributions in green and blue
respectively
    # This returns a stack of the two binary images, whose components you can see
as different colors
    color_binary = np.dstack((np.zeros_like(mag_binary), mag_binary, s_binary))

    # Combine the two binary thresholds
```

```
    combined_binary = np.zeros_like(mag_binary)
    combined_binary[(s_binary == 1) | (mag_binary == 1)] = 1

    return color_binary, combined_binary
```

Figure 4. Combined Threshold Function

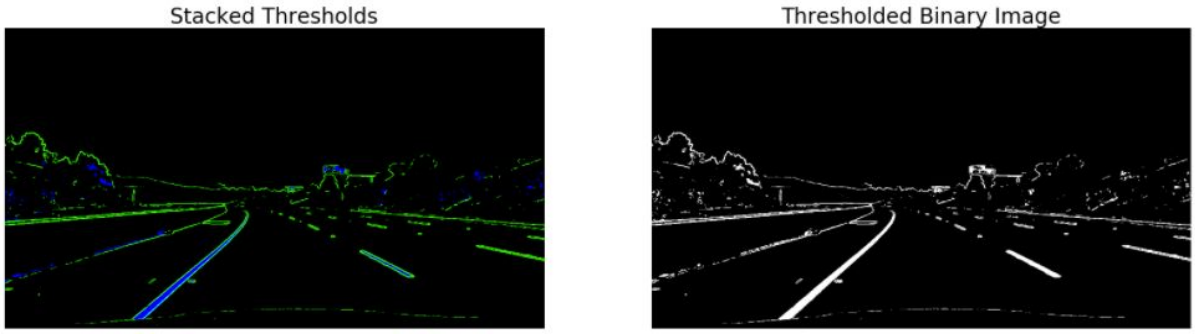

Figure 5. Threshold Function Output

To get the binary thresholded binary image a few thresholding techniques were used in conjunction. A magnitude gradient threshold was used, because it takes into consideration both x and y sobel orientations, which are two other gradient methods. It also results in a very low noise image. The technique used was to apply a color threshold. This was done by converting the image into HLS (hue, lightness, and saturation) space and segregating the saturation channel. We then combine the magnitude gradient with the saturation channel to get a binary image that shows the lane lines very clearly. In the left image of Figure 5 we can see the individual contributions to the binary image from each threshold. The gradient contribution is represented in green and the color contribution is represented in blue.

Perspective Transform

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform we'll be most interested in is a bird's-eye view transform that will let us view a lane from above; this will be useful for calculating the lane curvature later on.

Our images from the source video are a 2D representation of our 3D world. Since the is no z direction in a 2D representation objects that are farther away are represented as being smaller. We can approximate a top-down representation of the same image by bringing the farther away points closer by making them larger. Figure 6 shows the

location of some source points and where they would appear after the transform. Function `warp_image(img, src, dst, image_size)` is defined in the 12th code cell in the notebook. It makes use of other functions provided by OpenCV to do the transform, mainly `warpPerspective(img, M, image_size, flags=cv2.INTER_LINEAR).`

Figure 6. Source points and their destination position after the perspective transformation.

Figure 7. Original image and its top down view after a perspective transform.

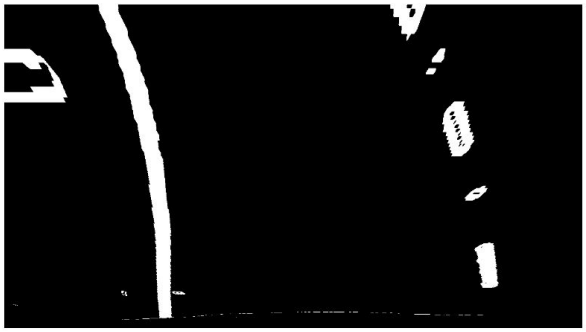

Figure 8. Binary image from applying thresholds to the top down image

Finding Lanes

With the top-down binary image we can now decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line. Do do this we use a histogram of pixel values.

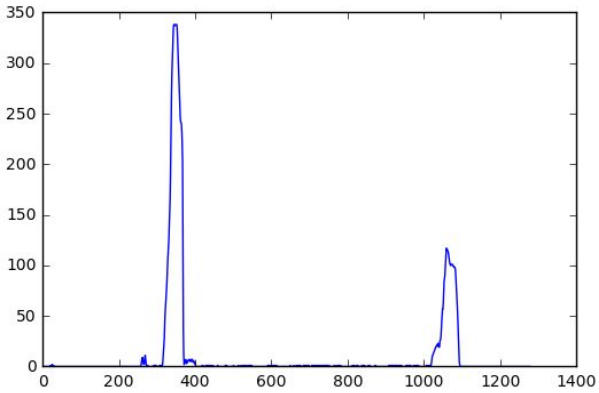

Figure 9. Histogram of Pixels .

With this histogram we can add up the pixel values along each column in the image. Since the image is binary (meaning that all the pixels are either 0, or 1) we can take the two main columns of the histogram to indicate the location of the lane lines. We combine this with a sliding window approach to identify the lines at different segments of the image and build up our lane lines. We also consider the center of each column in the histogram to identify the coordinates of the points that we will use to plot the lane lines.

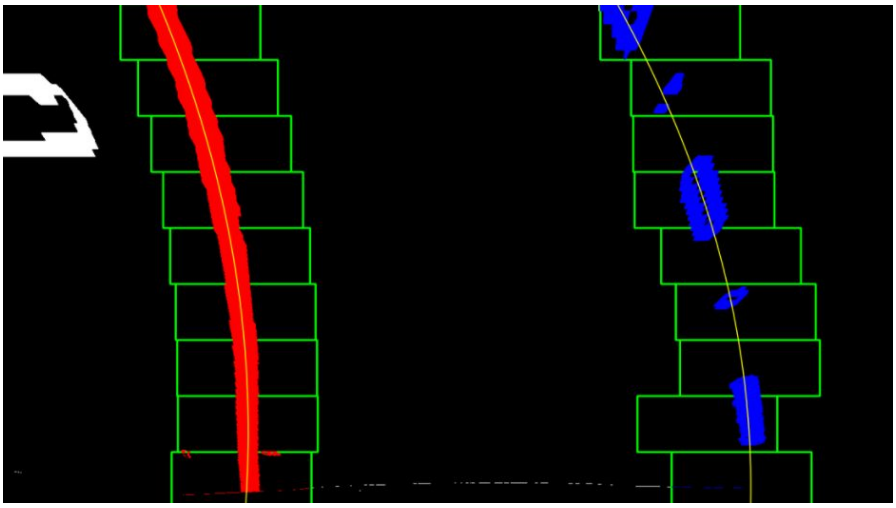

Figure 10. Lane Lines Identified by Sliding Window Technique.

This in done in the 18th code cell in the notebook by the function `find_lane_lines(binary_warped, debug=False).`

Now that we have plotted the lane lines on a top down representation of our original image we can reward the identified lane lines to accurately represent the lane lines in the original image. We get the curvature of the lines and the car's deviation from the center by performing some math given the location of the lanes.

Figure 11. Identified Lane Lines

Video

The pipeline described over the previous sections of this report has been applied to every frame on the source video to produce a new output_video with the lanes lines, and some information about them, overlaid on the original video. The output_video is included in the same folder as this report.