Advanced Lane Finding Project

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.

Camera Calibration

The code for this step is contained in the first two code cells of the IPython notebook located in "./P2.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 a test image using the `cv2.undistort()` function and obtained this result for the first example chessboard image:

For later usage I saved the intrinsic camera values (3x3 matrix and the distortion coefficients) in a pickle object to disk.

Pipeline (single images)

1. Provide an example of a distortion-corrected image.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

This part can be found in the second code cell in the jupyter notebook "./P2.ipynb", where the 3x3 camera matrix and the distortion coefficients are loaded from the saved pickle file. For undistorting a test image the function `cv2.undistort()` is used again. As can be seen in the result on the right side,

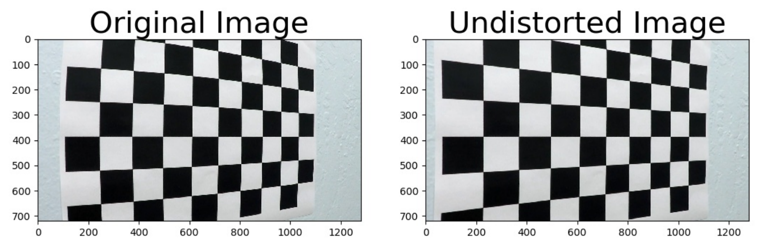


Figure 1: Original and Undistorted chessboard

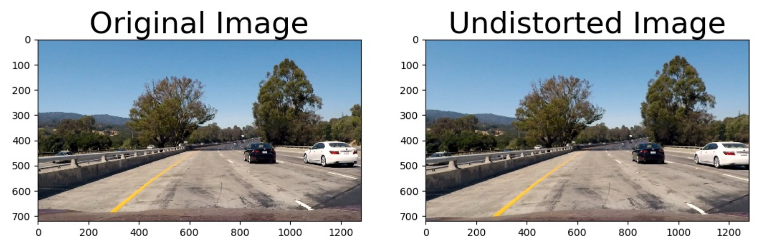


Figure 2: Original and undist test image

the white car is cut off while in the original image it is fully shown, meaning that the undistorting has successfully been applied to the test image.

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The code for this chapter can be found in the third code cell in the jupyter notebook "./P2.ipynb". The very first thing I did for this goal, was to explore color spaces and their channels to get a feeling for it.

RGB color space The RGB color model is based in the theory that all visible colors can be created out of the three primary colors, red, green and blue. The more each color is present in a pixel the whiter it gets.

HLS and HSV color spaces The HLS color space describes colors with the three properties, color itself, saturation and lightness or value. While color itself or hue is straightforward, saturation is defined by the amount of gray (0% = neutral gray, 50% = little saturation, 100% high saturation) and lightness/value is the amount of light reflection also called luma (0% no lightness = black, 100% full lightness = white).

The advantage of HLS and HSV over RGB color space is that lightness and saturation can be changed without changing the color. This is very useful for image processing, where the color of the lanes stays more or less the same, but only saturation and especially lightning changes due to shadows, sunlight, …

To visualize the different color spaces I took a rather difficult picture with shadows, light and dark road as well as white and yellow lanes. The following pictures show the different color channels of the above mentioned color spaces.

The R - RGB, S - HLS and V - HSV channel look very promising, while the G - RGB, L - HLS and S - HSV channel look kind of interesting as well.

Looking at the thresholded image one can see that the red channel from the RGB is not suitable for images where the lane is on a light colored road.

We need a color space that separates the image intensity from the color information. That is one reason why HLS and HSV are useful. In HLS color space we differentiate the color from the saturation

Looking at the same binary threshold with the S - HLS channel as a basis, we get the following:

Or with the S (HSV) - channel as a basis:

Building the pipeline For building the pipeline the approach was to make is as simple as possible in the beginning and then get more complex only if needed.

1. Grayscale + Threshold –> Get the white lanes
2. Red channel –> Get the yellow lanes

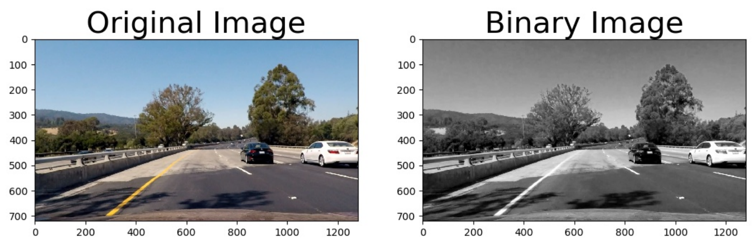


Figure 3: RGB - R Channel

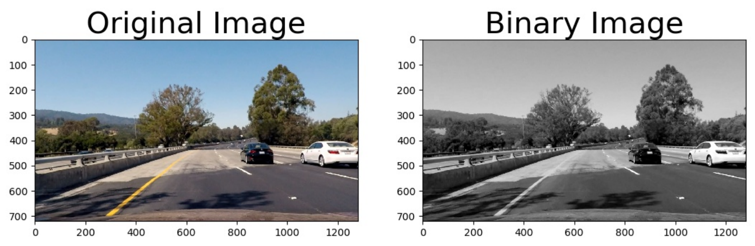


Figure 4: RGB - G Channel

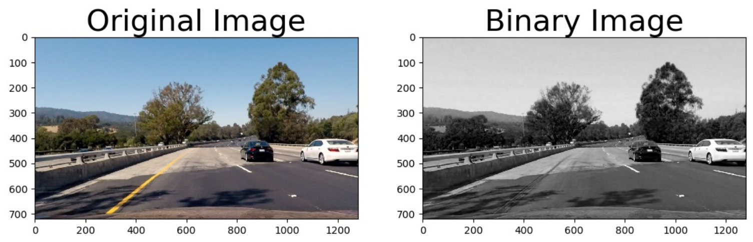


Figure 5: RGB - B Channel

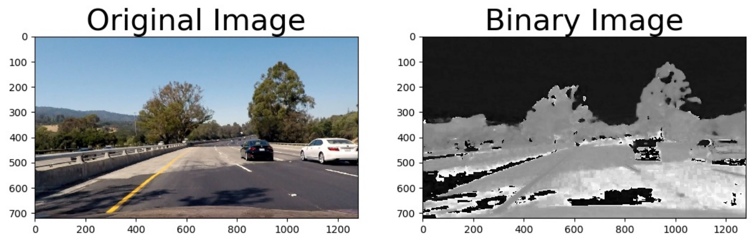


Figure 6: HLS - H Channel

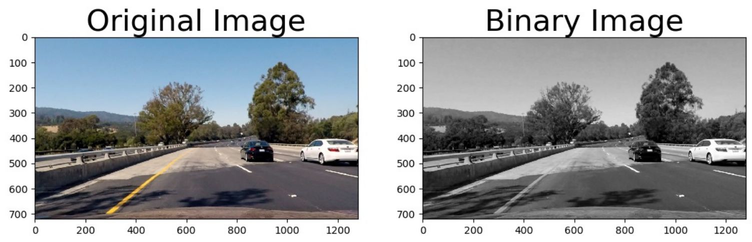


Figure 7: HLS - L Channel

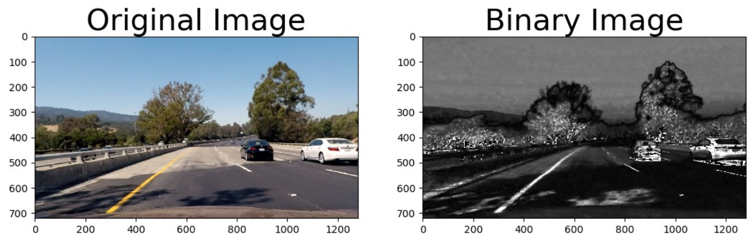


Figure 8: HLS - S Channel

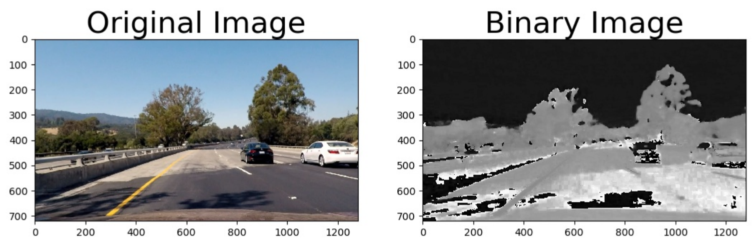


Figure 9: HSV - H Channel

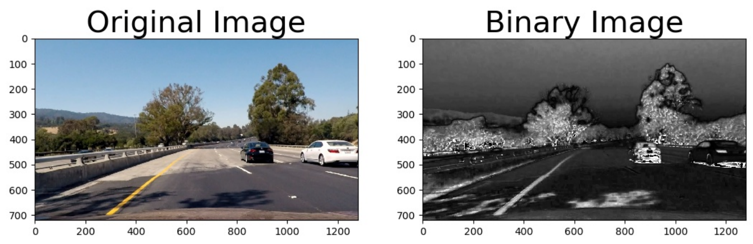


Figure 10: HSV - S Channel

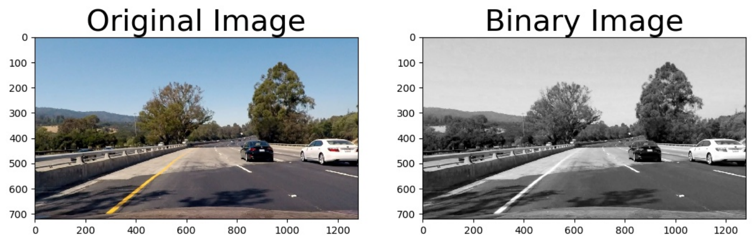


Figure 11: HSV - V Channel

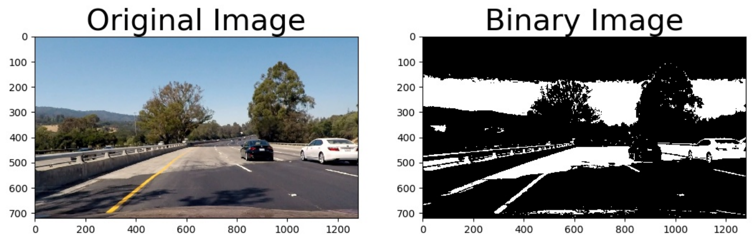


Figure 12: Binary image from R - channel

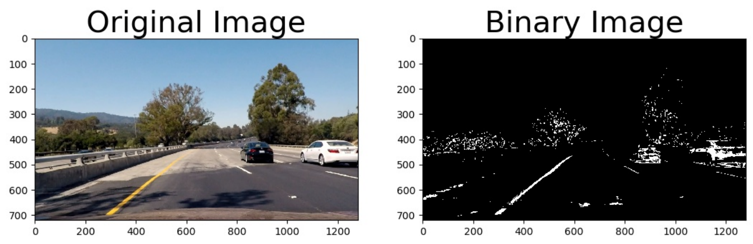


Figure 13: Binary image from S (HLS) - channel

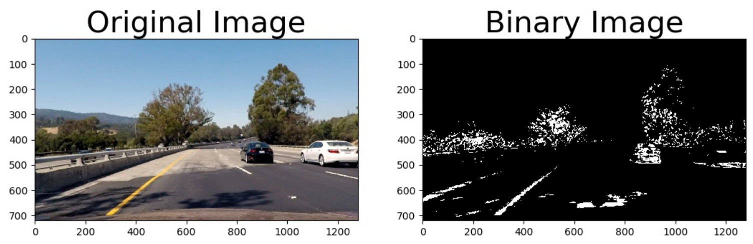


Figure 14: Binary image from S (HLS) - channel

3. HLS –> Saturation channel mostly, since there is a high sensibility
4. Gradient –> not for now
5. Combine the channels

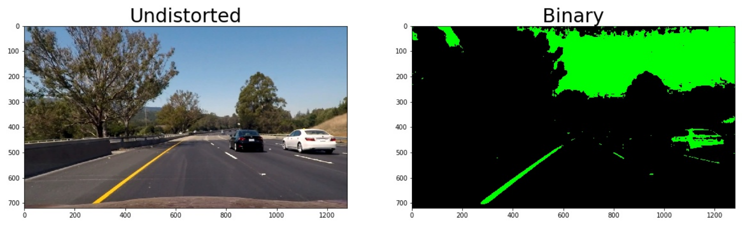


Figure 15: Combines binary - Image 1

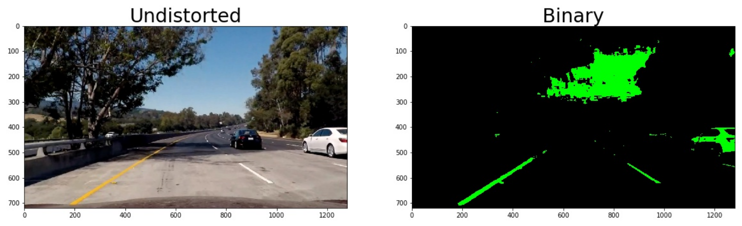


Figure 16: Combines binary - Image 2

As it can be seen the pipeline chosen is sufficient for the given test images.

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform can be found in code cell 4 of the jupyter

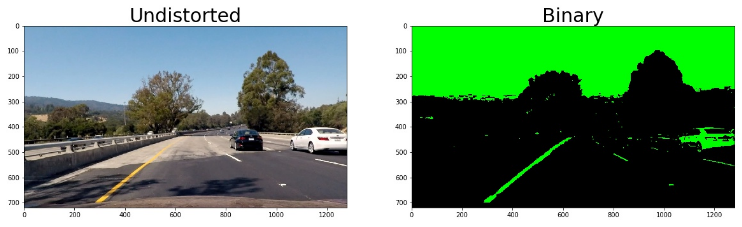


Figure 17: Combines binary - Image 3

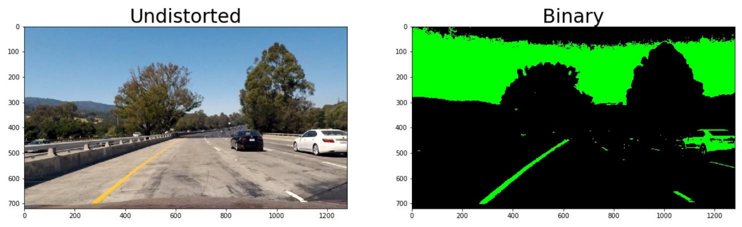


Figure 18: Combines binary - Image 4

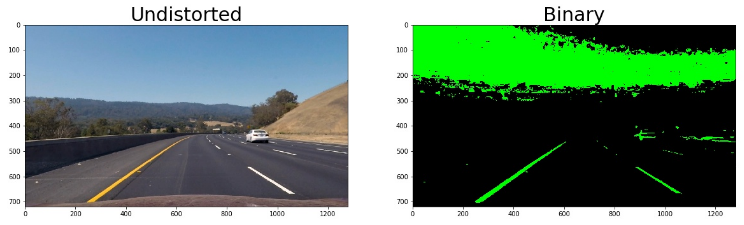


Figure 19: Combines binary - Image 5

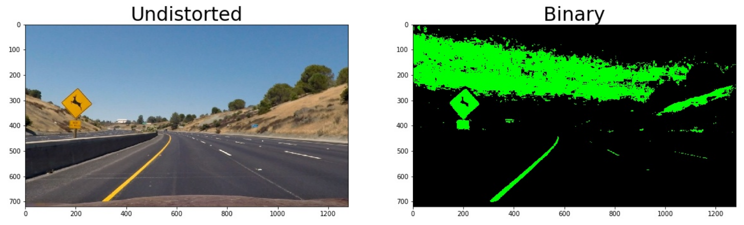


Figure 20: Combines binary - Image 6

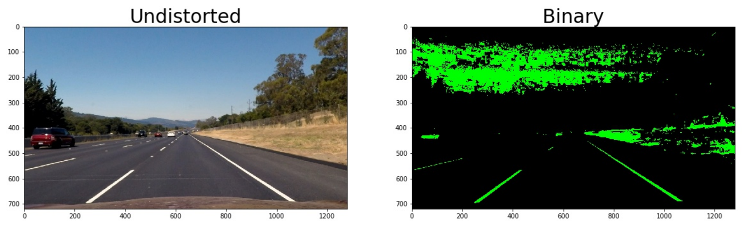


Figure 21: Combines binary - Image 7

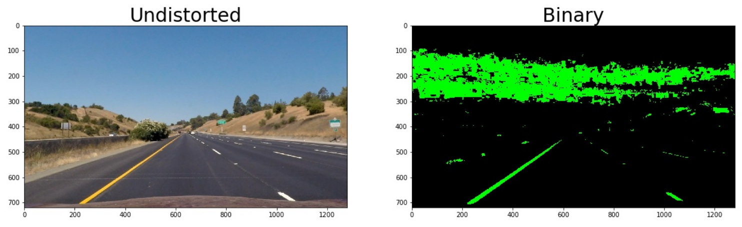


Figure 22: Combines binary - Image 8

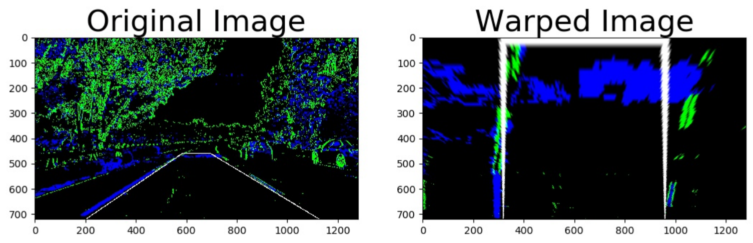
notebook. In this cell a function called `warper()`, takes an input image as well as source and destination points which are hardcoded as follows:

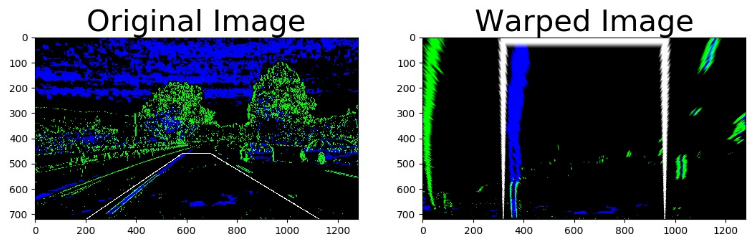
```
src = np.float32(
    [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
    [((img_size[0] / 6) - 10), img_size[1]],
    [(img_size[0] * 5 / 6) + 60, img_size[1]],
    [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
    [[(img_size[0] / 4), 0],
    [(img_size[0] / 4), img_size[1]],
    [(img_size[0] * 3 / 4), img_size[1]],
    [(img_size[0] * 3 / 4), 0]])
```

This resulted in the following source and destination points:

Source	Destination
585, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 460	960, 0

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.





4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The section about finding lane pixels can be found in code cell 5 of the jupyter notebook. The idea of the pixel search is to slide a window over the two lanes, each starting at the bottom of the window, counting the pixels in the window. If a specific amount of pixels is exceeded, the window is recentered to the mean x-value of these pixels. If we start at a good location (where the lane actually starts) the search automatically finds its way through the window. If the input images are good enough, then this path will also be along the lanes. The loop ends if each of the two windows reached the top of the image. All indices of the nonzero pixels inside the sliding windows are appended to a list. After having found all these indices we should have all pixels belonging to the two lanes. Finally a polynomial regression with any order (here we start with order 2) can be applied and drawn onto the image.

I started with taking the histogram of the lower half of the image:

```
lower_half_img = binary_warped[binary_warped.shape[0]//2:,:]
histogram = np.sum(lower_half_img, axis=0)
```

With the histogram the amount of pixel in the y-axis is counted and plotted on the x-axis. I expect to see two peaks indicating the two lanes. This is shown in the histogram plot, where the right peak is higher due to the continuous lane.

After splitting the histogram in half in the middle and taking the maximum of each side, we have two starting points for the sliding window loop. For identiyfing the non black pixels (rgb = 0,0,0) we use the `nonzero()` function from numpy.

With these two functions we apply the polynomial regression with second order to all pixels found in the sliding window loop.

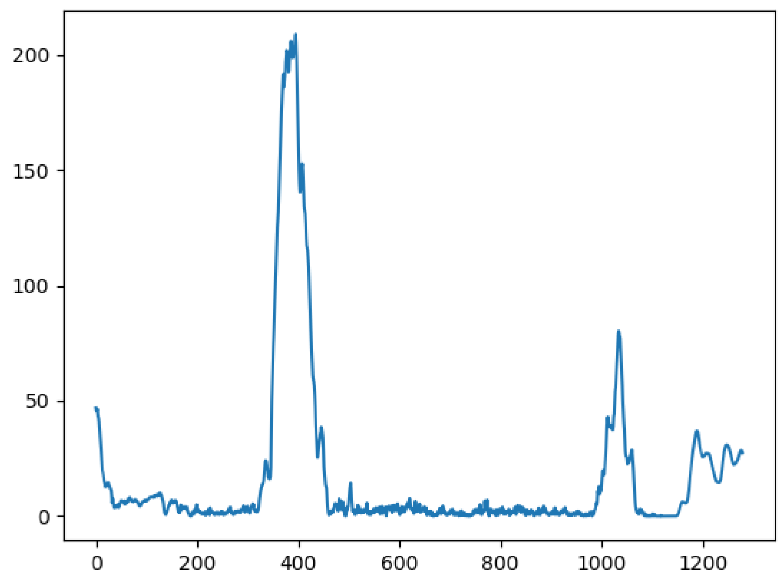


Figure 23: Histogram indicating two lanes

```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

The return values are the coefficients of a general second order polynom. To plot these functions we need to write:

```
left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
plt.plot(left_fitx, ploty, color='green')
plt.plot(right_fitx, ploty, color='green')
```

After having found two polynoms for the two lanes we do not need restart the pixel search for the new image from scratch, but can start in a range with a given margin around the two polynoms. This will speed up the pixel/polynom search for all frames of the video but the first one.

5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The calculation of the curvature can be found in code cell 6 of the jupyter notebook. The radius of curvature is given as:

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

Figure 24: Radius of curvature

In python this equals to:

```
((1+(2*left_fit[0]*y_eval+left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
```

for one lane. The `y_eval` can be any point along the y-axis of the image. In the implementation the maximum of y is chosen, which equals to the bottom of the image.

Additionally I calculated the offset of the car/camera to the lanes by subtracting the middle point between the two lanes with the middle point of the image:

```
xm_per_pix = 3.7/700 # meters per pixel in x dimension
leftx_base = left_fitx[-1]
rightx_base = right_fitx[-1]
```

```
offset_pixel = np.abs((leftx_base + (rightx_base - leftx_base) / 2) - img_size[1] / 2)
offset = np.around(offset_pixel * xm_per_pix, 3)
```

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I implemented this step in code cell 7 of the jupyter notebook. Here are a few examples of my result on the test images:

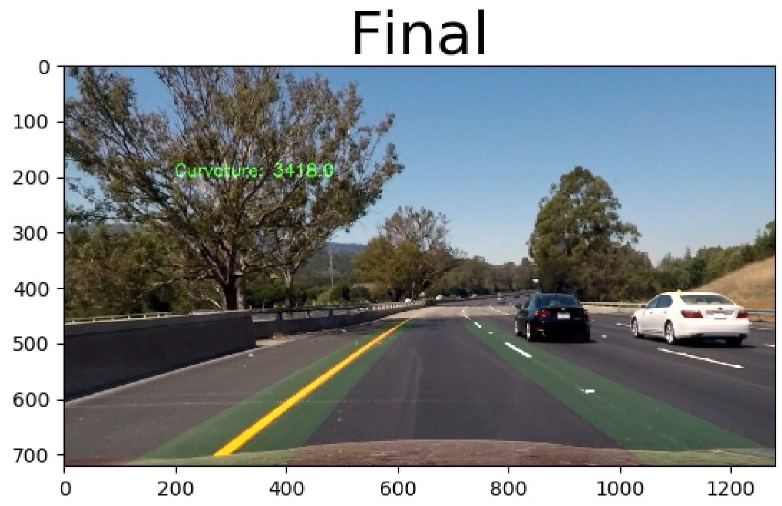


Figure 25: Warped back 1

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!).

The link for the video is actually five links, since on my local machine the ram is limited, which is why I could not solve the whole video inside the Docker container:

Project video Part 1/5

Project video Part 2/5

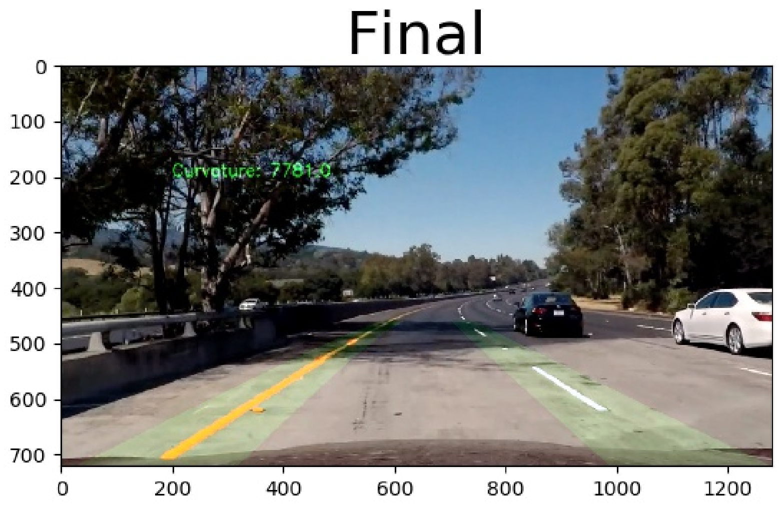


Figure 26: Warped back 2

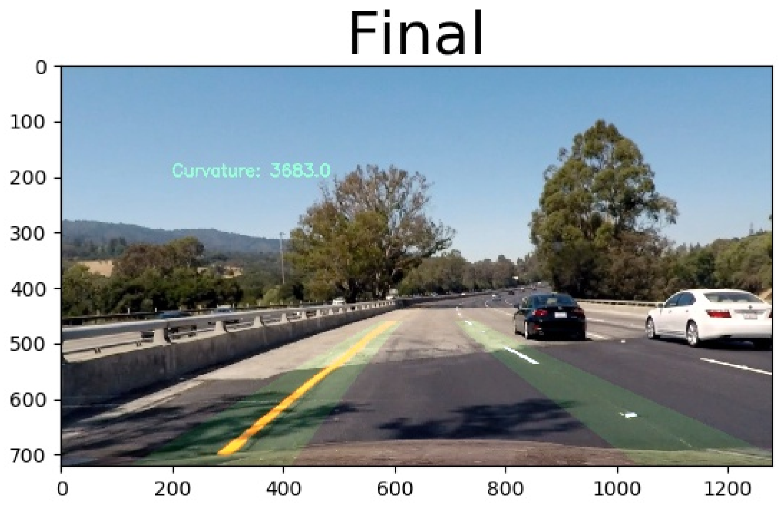


Figure 27: Warped back 3

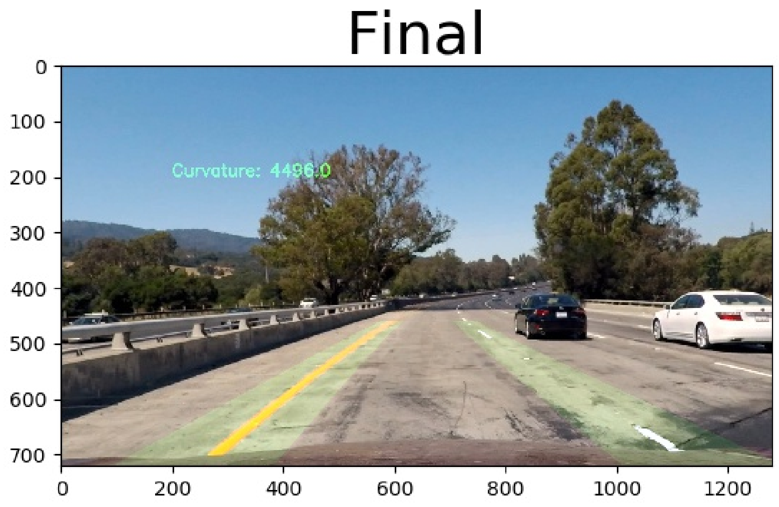


Figure 28: Warped back 4

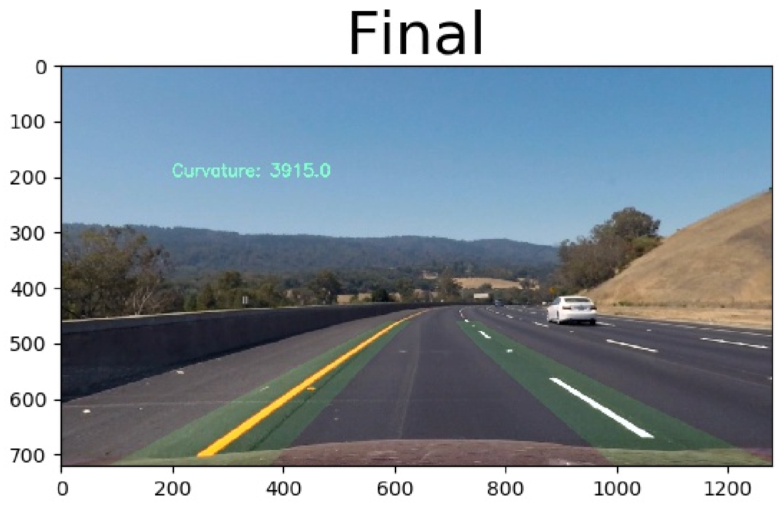


Figure 29: Warped back 5

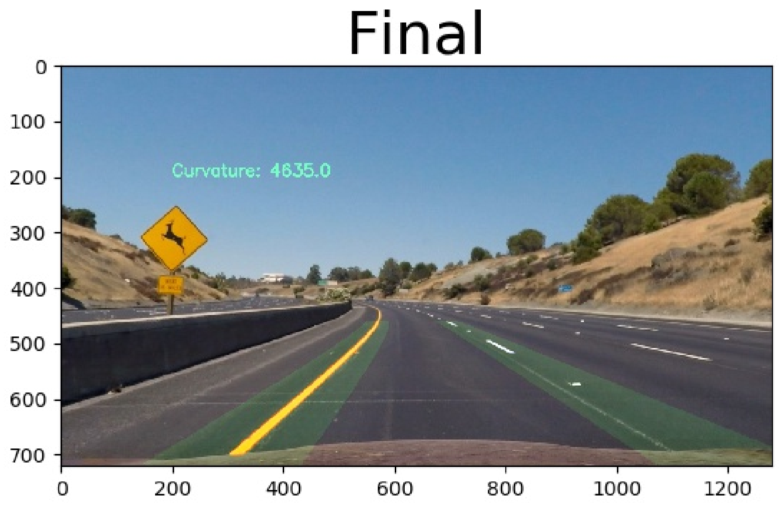


Figure 30: Warped back 6

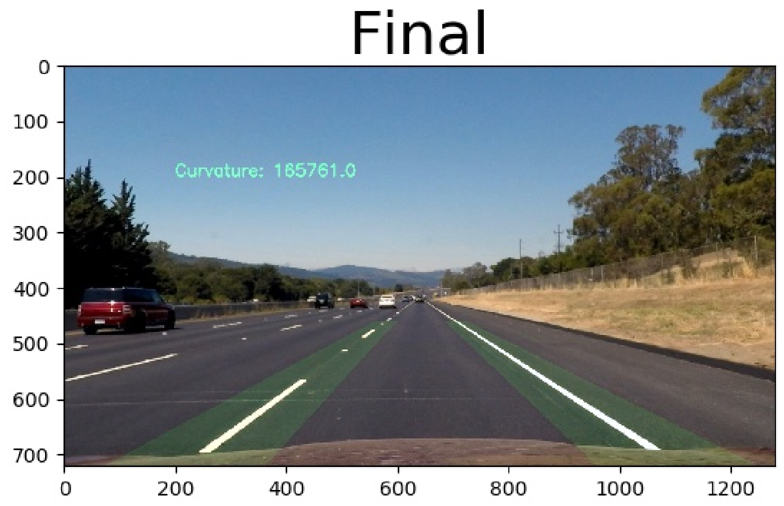


Figure 31: Warped back 7

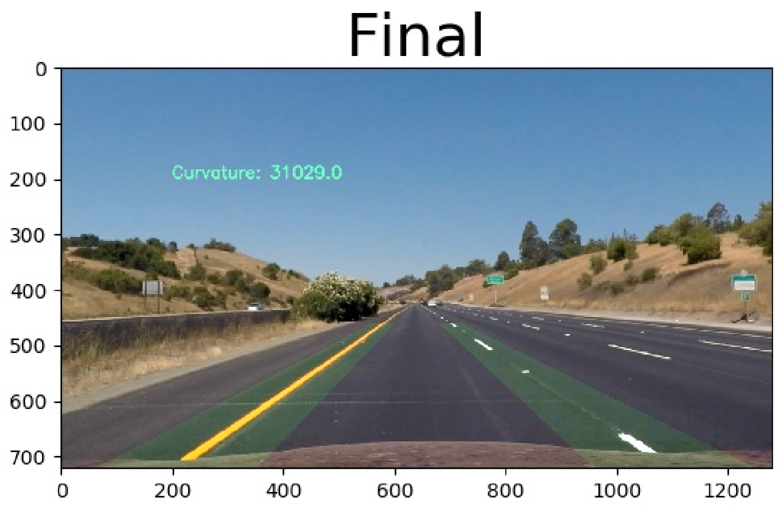


Figure 32: Warped back 8

Project video Part 3/5

Project video Part 4/5

Project video Part 5/5

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?

The approach that was taken here, was to correct the intrinsic camera failure, by calibrating the camera with given chessboard images. This step is rather straightforward.

The next and biggest step in this project is the exploration of the color channels and defining the pipeline. The approach that was taken here, was to start simple and only look at the test images provided. The found solution (=a rather simple combination of the thresholded gray, red and saturation channels) is sufficient for the test images provided, but might be insufficient for more complicated images/videos like the challenge video.

One could improve the pipeline by making a combination of color channel thresholds and gradient thresholds and really fine-tuning the lower and upper threshold parameters. These might even be adjusted depending on certain input parameters from the image. Especially the quick change of bright street color to dark color and the color of the lanes in combination. Further, some patterns on the road, that look like lanes, but are not need to be addressed in the section of creating a binary image.

In the next part of the project the pixel finding algorithm was provided. This algorithm takes into account that the lanes start at the bottom of the image and end at the top of the image, which is not necessarily always the case, especially in sharp corners. This would need to be addressed as well if the project is developed further to make it more robust.

At last some helpful calculations like the curvature and the offset calculation can be made. This is rather straightforward as well.