Advanced Lane Finding Project – Report

The purpose of this document is to provide a detailed reported about the used process to find lines on the read, based of Udacity's lessons for this project.

This projects comes as part of the Udacity's Self Driving car Nanodegree program.

It is the second project related to find lane lines. It builds upon the first project, but introduces a new way of doing the same task with different knowledge, making it more robust at the same time.

For this project, the goals 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
- Wrap up all the above steps and apply it on a video stream pipeline.

The rest of this document will be occupied with different sections, each one detailing each of the bullet points above.

Disclaimer: all code samples and references provided below can be found in this python notebook publicly available on github.

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

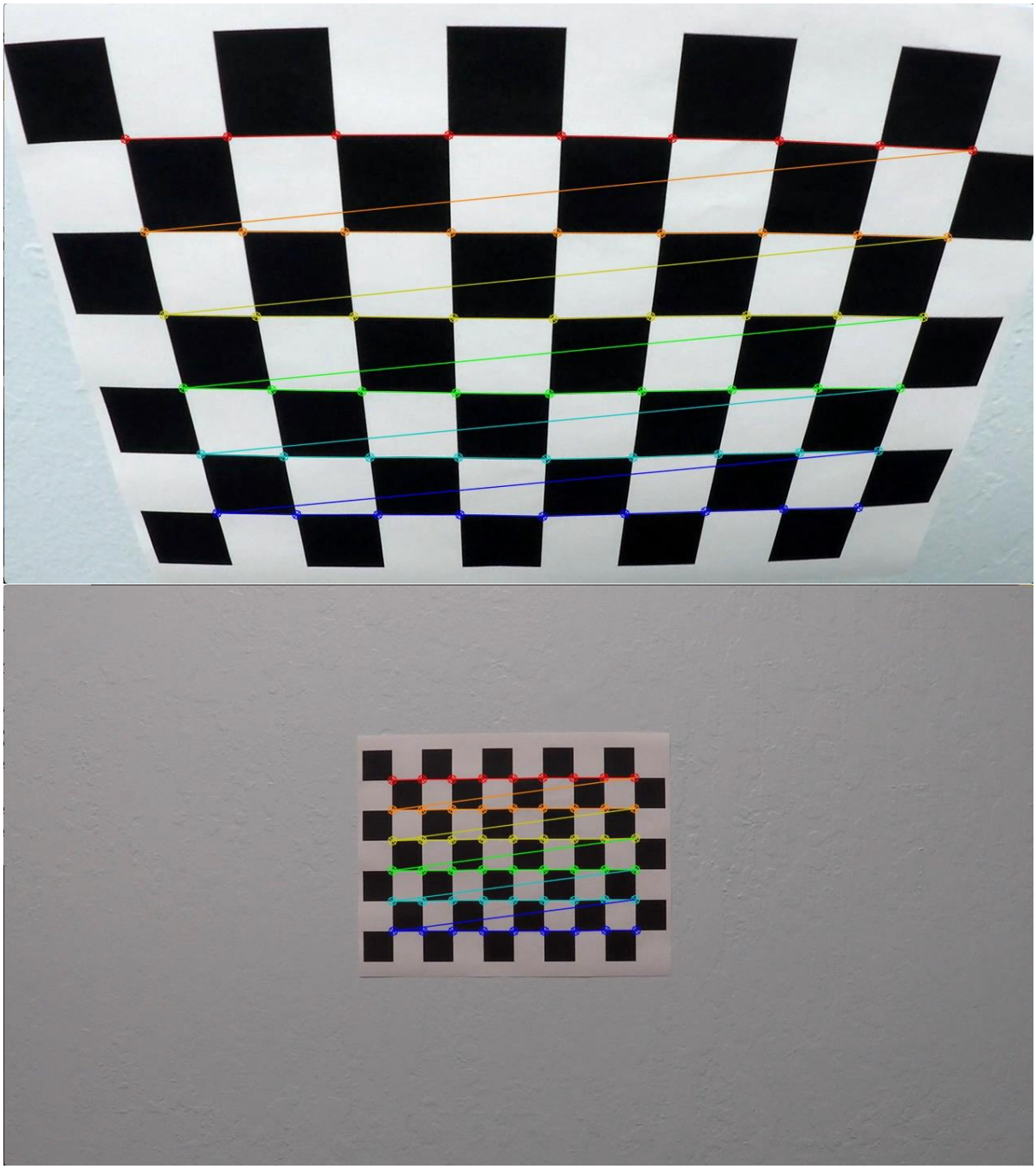
The images used in this project are all front facing images of a car. Before we can use these images, we need to be sure that the camera is capable of correctly measure distances between objects in an image.

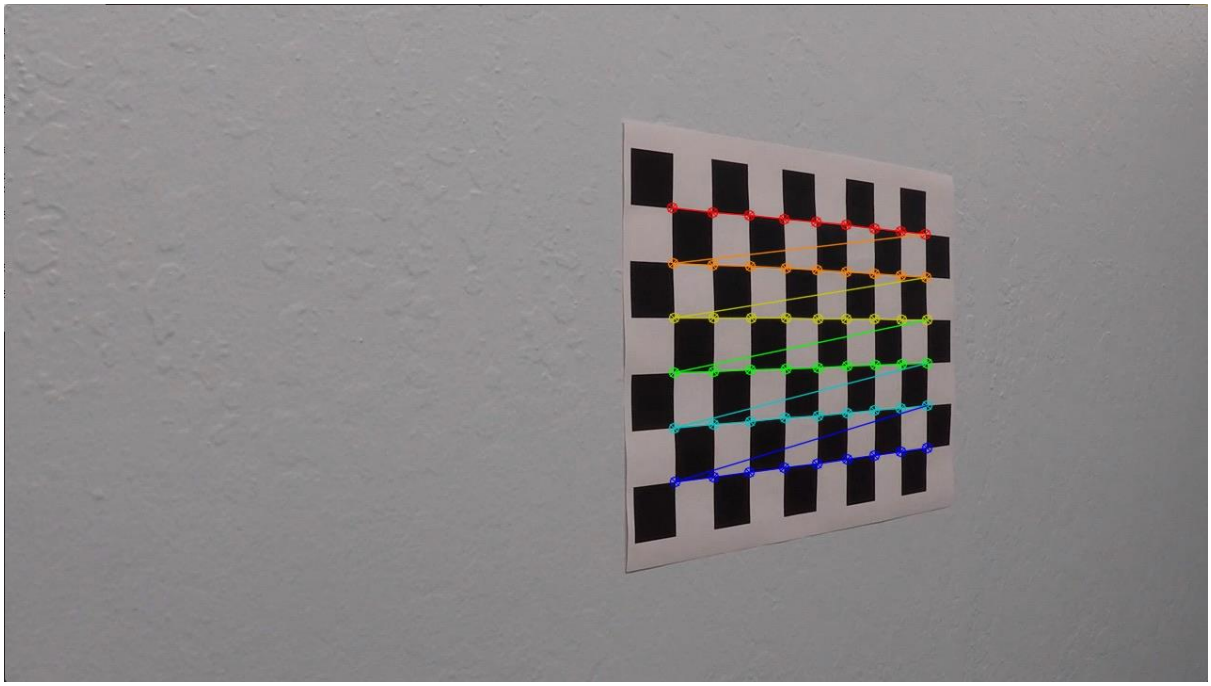
For that, as explained by Cezanne Camacho in the lessons, we need to find the calibration matrix and distortion coefficients for the camera.

One way to obtain these values is by taking advantage of python opencv library, that provides functions to do this.

In the provided notebook, this step can be found in the function "calibrate_camera". This method simply uses the provided calibration images on the root of the project.

This is the outcome result of this code:





The full images results can be found in this folder on github.

Apply a distortion correction to raw images

After calibrating the camera, something far more important for out later video streaming on the road than the calibrated chessboard pictures above, was without a doubt the calculation of the calibration matrix and distortion coefficient (mtx, dist) that now will be used to apply a distortion correction to our actual road images.

As mentioned, now we can apply a distortion. The way this was done for this project was to also use an opencv function, in this case, cv2.undistort() function.

In the provided notebook, a method called “undistort_images” is used for notebook training and perception purposes.

During the pipeline video stream, the method “undistort_image_computed” is the one actually used.

Below there is a comparison between the original image, and the same image after distortion correction.

Figure 1 - Original image

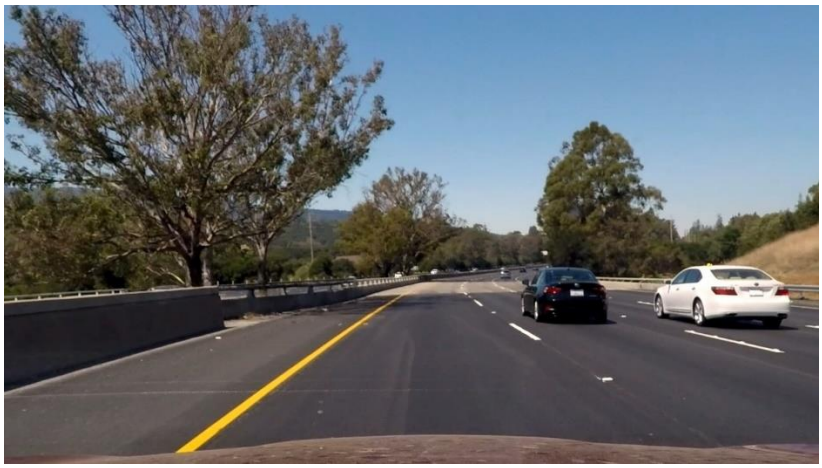


Figure 2 - Image after distortion correction

A full set of samples images after distortion correction can be found is this github folder.

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

This step is a great step to highlight lane lines in an image. A better way to put it, it's a great way to isolate the line pixels in one image.

This step is in my opinion one of the most important improvements regarding the first project, where to detect the lines, we basically used canny edge detection.

The problem with that is that in different lighting conditions, the canny edge dectection loses a lot of it's value.

Several color transformations and gradients were tried.

Below it's possible to find a sample of images manipulations through color and gradients transformations.

H channel from HLS

Figure 3 - A sample image containing the H channel form HLS color gradient

More images from the H channel available on github.

L channel form HLS

Figure 4 - A sample image from the L channel from the HLS gradient

More images from the L channel available on github.

S channel from HLS

Figure 5 - A sample image from the S channel from HLS gradient

More images from the S channel available on github.

Sobel X filter (Excellent for detecting vertical lines)

Figure 6 - A sample image with the sobel x filter applied

More images of the sobel x filter available on github.

For the provided image, a combination between the sobel x filter and the S channel from the HLS color space seemed appropriate. Other combinations would probably be as efficient or even better than this one, but for the purpose of the project, this combination looked ok.

A region of interest was also applied.

After all these step, the same sample image as the shown above looks like this:

As it's possible to observe, the majority of the noise was eliminated and we basically have isolated a lot of the lanes without much more information on the image.

More images of the combined color spaces and gradient available on github.

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

After manipulating the images through color space and gradients, the next step in the pipeline was to apply a perspective transformation on the image.

Once again, python opencv library was used to perform this task.

For this task, some steps were necessary:

- Find the source points
- Find the destination points
- Compute the getPerspectiveTransform(dst, src)
- Compute the warped image warpPerspective(img, M, img_size, flag=cv2.INTER_LINEAR)

Where:

Dst – destination points

Src – original points

Img – The image to be warped

M – the original matrix of the perspective transform

Something to have in consideration here is that the points are manually calculated here, and although that is fine for the project video, where the road is almost always straight and the images will have more or less the same formats, that becomes a real problem for the challenge video, where the road is far from straight, and when the road is curving to much, the algorithm gets very lost.

Below a sample image after the warped transformation:

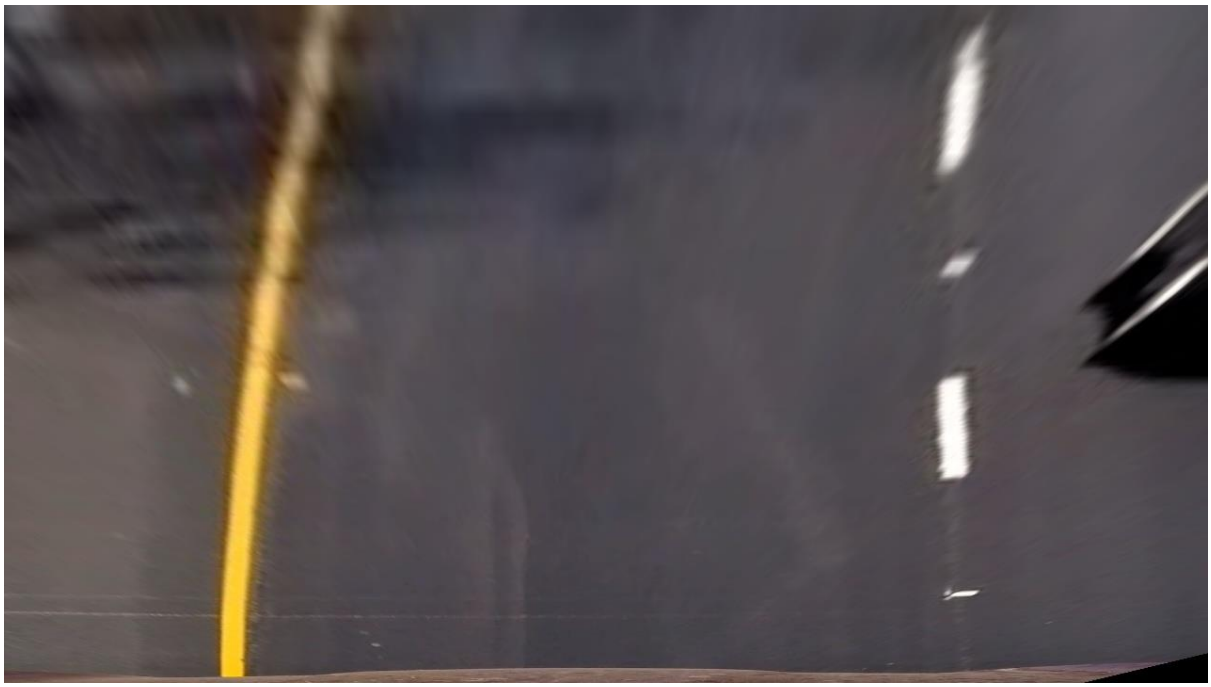


Figure 7 - Original warped image

More original warped images available on github.

Figure 8 - Thresholded warped image

More thresholded warped images available on [github](#).

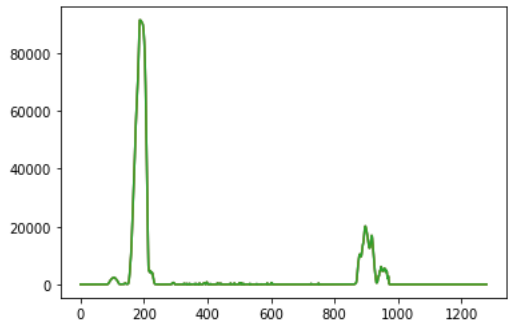
In the pipeline, the thresholded warped image is the one be used. The other image is just here for reference and explanation purposes.

Detect lane pixels and fit to find the lane boundary

Next step on the pipeline is to use the thresholded warped image and find the lane lines on the images.

For this, the first step is to find peaks in one histogram of the image. These peaks would represent the white pixels on the image, and therefore, identifying the images.

Here is one example of the histogram:



The left peak represents the left line, and the right peak represents the right line.

In this example, the left peak is higher because the line was not segmented, while the right line was.

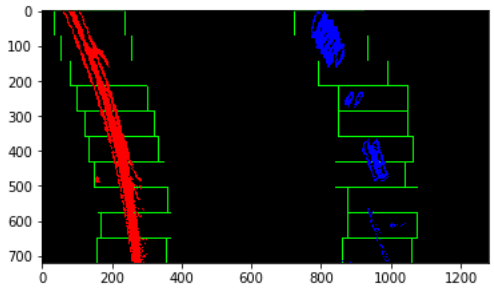
Next, the sliding window technique was applied.

The histogram is important in here because they are used as a starting point to find where the lane line pixels are.

Because we have nwindows to iterate, the lane line pixel finding gets retouched and improved in each step.

Disclaimer: here the code complexity started to get harder, and the majority of the code was directly extracted from the lessons provided by Udacity.

One sample image after sliding window can be found below.

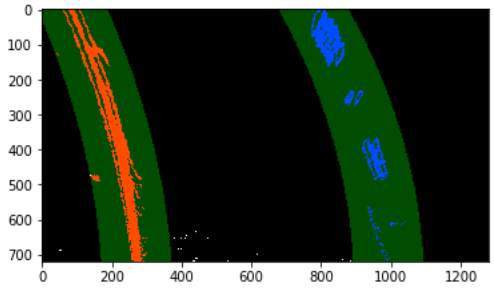


Optimization:

For the next frame, because we know there is no radical change of pixel positions, we can use the previous left and line pixels as a starting point.

In the github notebook, this optimization function is called “search_around_poly”.

The final result of the detect lane line pixels look something like this:



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

This step will be important to add metrics to the final video output.

Here it is important to mention, as explained in the lessons, that the original curvature values are obtained in pixels, but those values do not bring are real value for the real world human to use.

That's why in the "measure_curvature_real" method on the github notebook, ym_per_pix and xm_per_pix are declared, to provided a conversion between pixel values and actual meter values that make sense to the real world.

This method return a left and right curvature, and will be added to an weighted imaged later on.

Warp the detected lane boundaries back onto the original image

At this stage, we already calibrated the camera, applied the distortion correction step, masked the lane line by applying color gradients and manipulating color spaces, and even applied a perspective transformation to see the lane lines like the eyes of a bird.

All these steps made possible to identify the lane lines on the road.

Now we just need to grab the original image and "paint" that original image with the green zone obtained from the previous calculations.

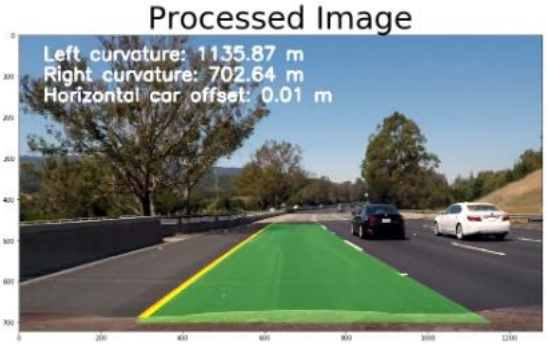
Once again, I used code provided by Udacity to perform this step, with very small changes.

This method on the github notebook is called "compose_image_with_lines".

Below is the result:

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

In this step we simply take the step above and add the curvature information as explained in the steps above, the result looks like this:



Wrap up all the above steps and apply it on a video stream pipeline.

This step should be self explanatory. It is the same step as above but applied to a video stream

The link for the generated video can be found here.

Challenge video result

Additionally, the output for the challenge video can be found here.

Discussion and future improvements

The outcome of this project looks like a much more robust method to identify lines, in comparison to the first project.

I really enjoyed manipulating color spaces to highlight and isolate the lane lines, and the math behind the pixel finding is also fascinating.

The first video works almost in perfection, even when the color on the road floor changes during the video, and there is a difference in the shades because of the tree that shows up at the middle of the video.

This was a real problem during the first project.

However, on the challenge video the performance drops a lot, and I strongly believe this is happening because the points of the perspective transformation are being manually calculated.

In the first video, the car is driving almost always in a straight way, and there are no drastic direction changes between frames.

That is a different scenario of what we can observe in the challenge video, where there are drastic changes in direction.

I already did a previous Udacity Nanodegree program on Computer Vision, and I think a viable solution would be to use a Convolutional neural network with a regression classification outcome to find the 4 destination points of our perspective transformation.