Advanced Lane Finding Project

By Erwan Suteau

Introduction

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.

GitHub Structure

The source code can be found on GitHub, here: https://github.com/esuteau/car_advanced_lane_finding

It is organized as follows:

- **Source code**:
    - ***main.ipynb***: Contains all the python code until the video run (I also saved an HTML for reference). This jupyter notebook contains also a detailed description of the techniques used in this project
    - ***main.py***: Python version of the jupyter notebook. You can run this script to test the image processing blocks on test images
    - ***video_run.py***: Code used for the video run. Uses the functions defined in main.py as well as new ones for the video run.

- **Output Images**: In ***/output_images/test_images*** you will find examples images for each stage of the pipeline. Those were based on the initial set of test images.
    - ***binary***: contains the result from the binary thresholding
    - ***undistorted***: After running camera calibration on the binary images
    - ***warped***: After perspective transform on the undistorted images
    - ***line_detection***: Shows the points that were found during line detection
    - ***curve_fitting***: After 2nd order interpolation on the detected set of points
    - ***curvature***: calculation of the curvature on the 2nd order polynomials
    - ***final***: Overlapping the detecting lane on top of the original picture, after transforming back to the original camera perspective

- **Video**: ***output_images/output_video.mp4***

IMAGE PROCESSING

In this write-up I will briefly described the different stages leading to an appropriate lane detection on road images.

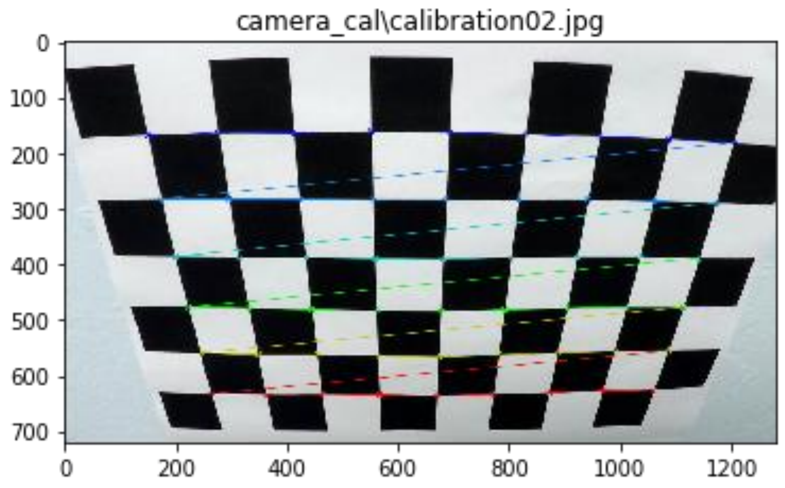
I highly recommend to open the html version of the jupyter notebook at the same time. You'll find a much more detailed description of the blocks. The notebook is well organized, and contains examples for all the different stages of the project, as well as the final video.

CAMERA CALIBRATION

Real cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses. This creates an effect that distorts the edges of images, so that lines or objects appear more or less curved than they actually are. This is called radial distortion, and it's the most common type of distortion.

For camera calibration, I just reused the block that was written in the lesson. It uses openCV's awesome library to find the corners of a chessboard. Then only necessary change was to set the dimensions to 9x6

Here is an example on a chessboard image

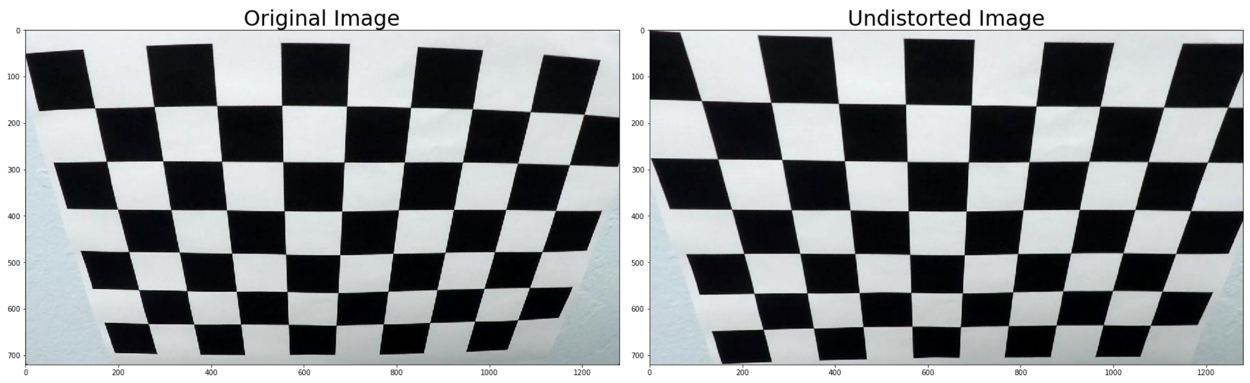


Pipeline

Distortion Correction

Again here, we use openCV's prebuilt functions (cv2.undistort)

The results for the same image is the following:



Binary Transform

After experimenting a lot with gradient thresholding, grayscale thresholds and different color spaces (RGB, HSV, HSL), I realized that using only the Hue and Lightness channels worked well for detecting Yellow and White lines, respectively.

The saturation channel, which seemed to work well at first for detecting white lines, is actually doing really poorly in the presence of shadows, which is the reason why I'm not using it.

After doing binary thresholding, we define a mask area and remove everything outside of it. This allows us to remove the sky and the left and right sections that could be bothering us for line detection.

The code can be found under the section Apply Binary Transform in the jupyter notebook. I'm actually not using the gradient functions at all, but kept them in the notebook.

All the following stages will be demonstrated on the test images called test2.jpg:

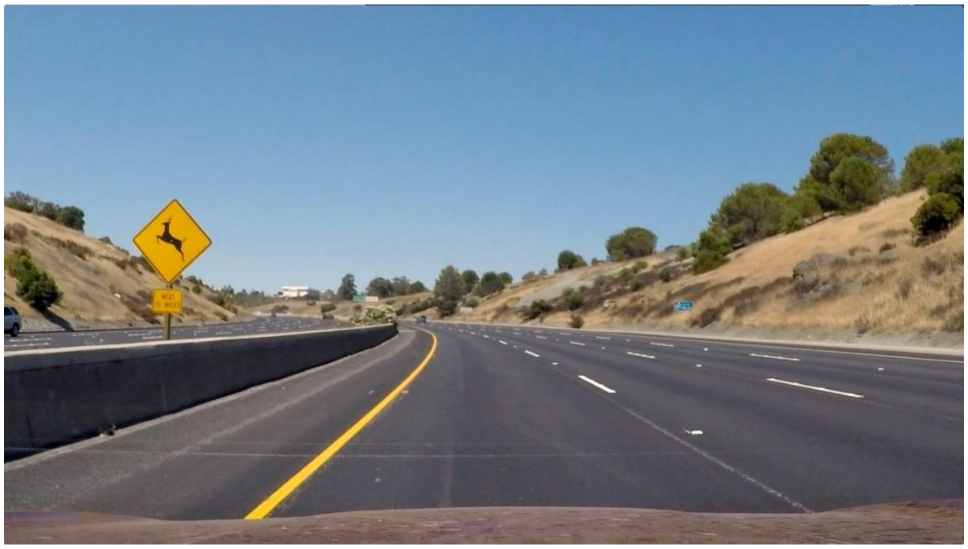


Figure 1: Our test image (test2.jpg)

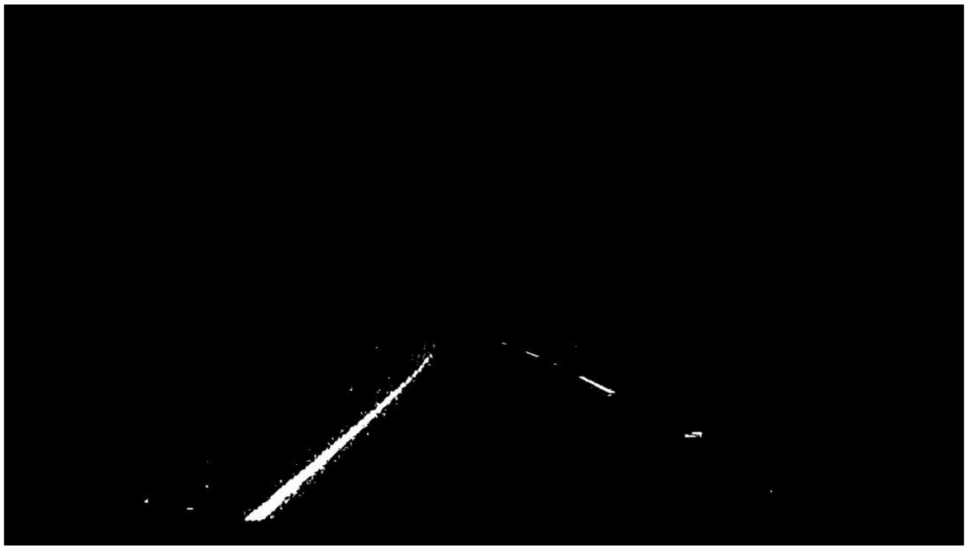


Figure 2: Binary Transform and Masking on test2.jpg

Perspective Transform

The goal here is to produce a top-down view of the image, so that we can later on calculate the curvature of the lane.

We are basically trying to move the camera at the vertical position, pointing down to the road (like a bird view of the road)

To do this we'll use opencv's *getPerspectiveTransform* and *warpPerspective*, with a source polygon and destination polygon. The source and destination polygons represent the same physical coordinates, but for 2 different positions of the camera.

We defined the coordinates of the source and destination polygon based on the first image of the dataset. Here are the coordinates that we defined:

```
height=720.0
width=1280.0
dst_margin=350.0

# Coordinates of Source polygon
src = np.float32([
    [206, 720],
    [583, 460],
    [703, 460],
    [1100, 720] ])

# Coordinates of destination polygon
dst = np.float32([
    [dst_margin, height],
    [dst_margin, 0.0],
    [width-dst_margin, 0.0],
    [width-dst_margin, height]])
```

Here is the result on the very first image, with straight lines.

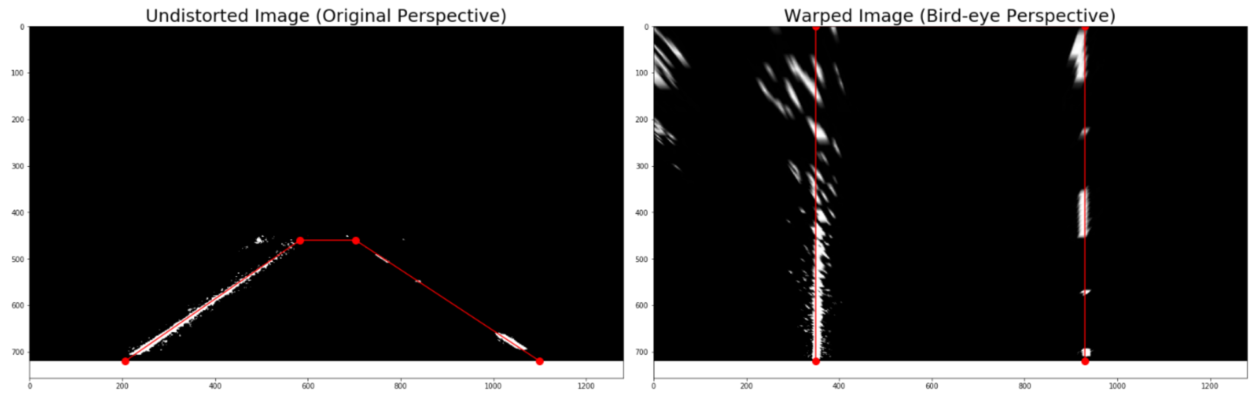


Figure 3: Perspective Transform (straight1.jpg)

And now on our test image (test2.jpg)

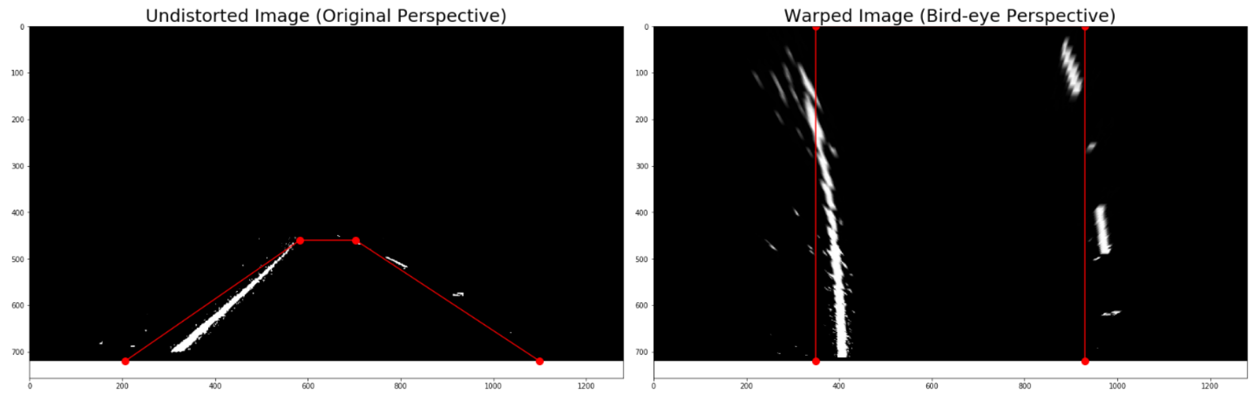


Figure 4: Perspective Transform (test2.jpg)

Line Detection

After get the top-down view, we perform line detection.

To do that, I process different stripes of road, of 50 px each. I sum the number of white pixels on each column, then do a moving average to remove noise, and then find the maximum.

If the maximum is below a certain threshold I consider that nothing was detected.

On this test image, here are the points we actually detect:

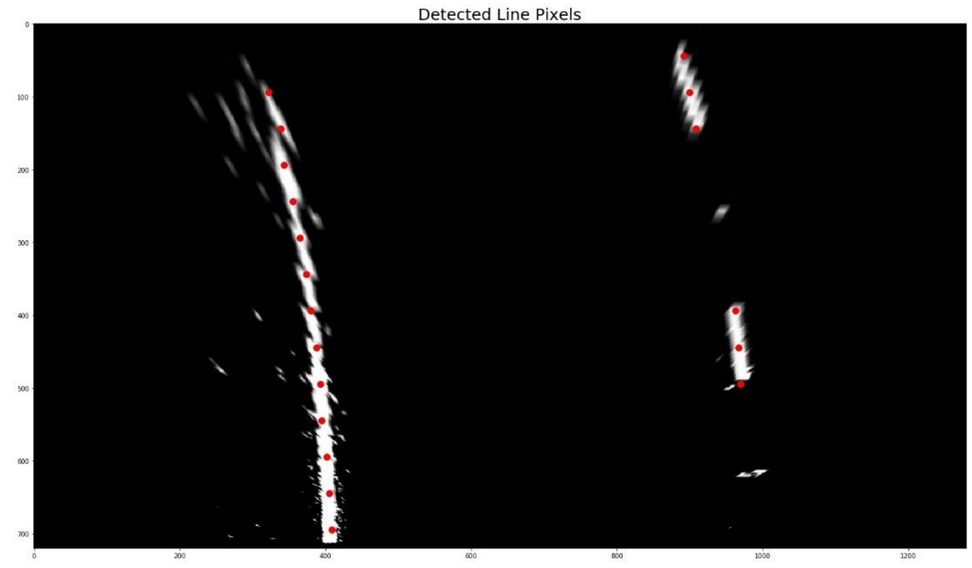


Figure 5: Line Detection on test2.jpg

Curve Fitting

Now we can fit those points with a 2nd order polynomial to extrapolate the position of the lines.

And we also calculate the curvature of the line.

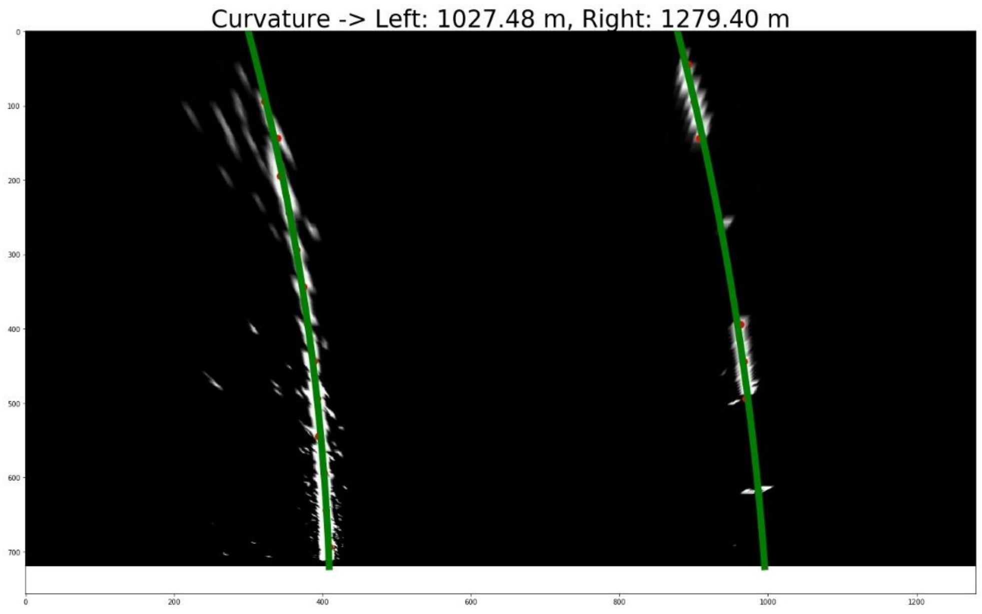


Figure 6: Curve Fitting on test2.jpg

Final Output

With both lines detected, we have the coordinates of the lane. We transform it back to the original camera perspective, draw the curvature and offset from the center and we get this:

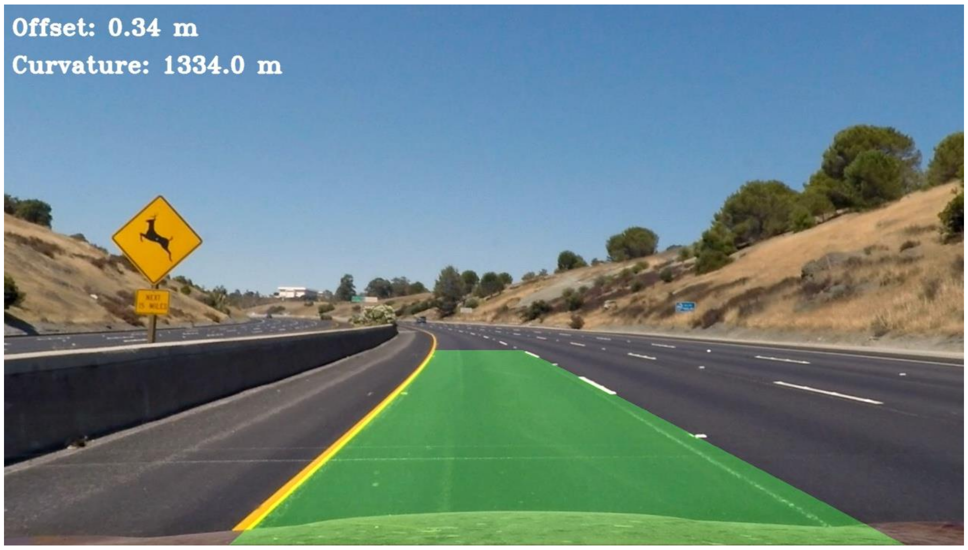


Figure 7: Detected Lane on original image

Note that a positive offset means that the car is to the right of the center, and a negative offset means that the car is to the left.

VIDEO

Now that we have our entire line detection toolbox ready, we could think that running it on a video would be straightforward. That's what I thought at least, but it is unfortunately not entirely true...

For once, we end up testing our system on a much larger dataset, so it wouldn't be surprising if the line detection went wrong on a few frames, because of shadows or other imperfections that we did not test for in our initial set of 8 images.

On top of that, running on a video offers some advantages: consecutive frames share some information. We should not expect them to have for example a curvature that is too different. We can use this to detect invalid frames and discard them, hoping that this will not happen too often. But it will, because we already saw in our test images that detecting the right lane is not trivial, somes images don't really have that much information on it...

The code for the video run is located in video_run.py. It uses the functions we have defined above, which are located in main.py

In addition to what we have seen so far, I have added a few more techniques to improve the line detection for the video.

1. Detection of anomalies: did we correctly find the lines on the current image? This is done the following way:
    a. Detect if we actually found a single point of the line.
    b. Detect extreme curvature, which should not be physically possible on a highway
    c. Detect if the curvature is really different from the previous frames (we actually use a moving average)
2. Smoothing: We combine together the points detected from the previous X frames to calculate the current line curvature and do interpolation. This prevents the system from jumping back and forth between different curvatures and solves the issue of not detecting many points. Smoothing is only done if the no anomalies were detected.

When we detect anomalies, we do a simple thing: we reuse the previously correct line detection. If too many anomalies are detected in a row, we reset the line detection and start again.

We keep count of consecutive detection anomalies for the left and right lines separately

IMPROVEMENTS

The current solution works well enough with the video example that was provided. But this is not enough data to really prove that this is a robust solution.

It seems for example that the current pipeline does have a little bit of trouble in the presence of shadows, so I would not be surprised to see it fail on a longer segment where shadows are very frequent. Also situations where the quality of the road degrades, or when lines fades away for more than a few seconds of data would cause some issues to the current algorithm (more that 100 frames.

And that's without talking about different lightning conditions, rain or even night conditions. None of those were tested, so it is very likely that the algorithm would have to become a lot smarter in those environments.

The most important section of the pipeline is the binary thresholding. Right now I'm only using Hue and Lightness dimensions, and I'm sure this could be improved.

Also, I know my algorithm for detecting the center of the line on different stripes of image would fail in the event of large patches of white pixel that do not represent an actual line, because my algorithm only finds the maximum, after some averaging. I do not take into account the width of the white patch that was found.

Here a better solution would be to use a cross correlation with a reference signal that represents the target line to be found, of correct width. This could help in those situations, but I did not have time to implement it. (Already spent 50+ hours on this project…)