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.

Rubric Points

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

Writeup / README

---

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. [Here](https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

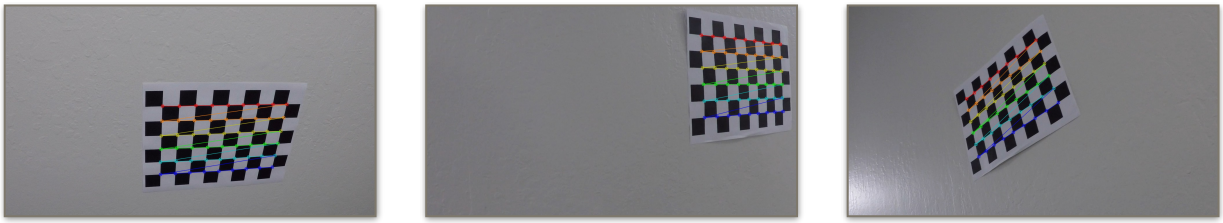
Camera Calibration

---

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

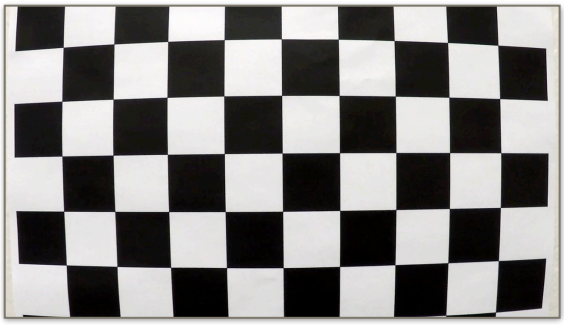
For the camera calibration I created a separated script (camera_calibration.py) that I ran on the provided calibration images and stored the parameters into a file that I later loaded to undistorted the video stream.

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.

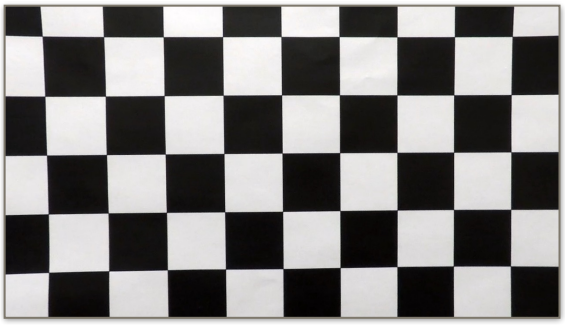


Example of the chessboard corner detection on the sequence of calibration images

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 the test image using the *cv2.undistort()* function and obtained this result:



Original image



Undistorted image

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:

Original image

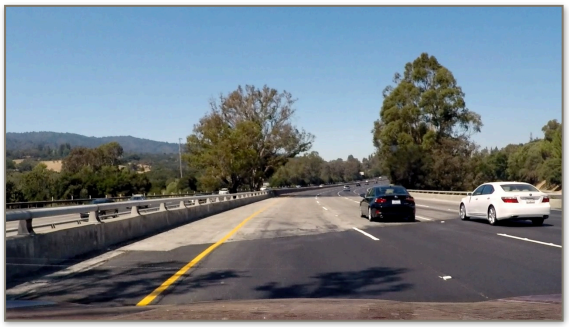
Undistorted image

To test an image processing step I created a script thresholding.py. At line 68 the calibration parameter are loaded and at line 73 they are applied.

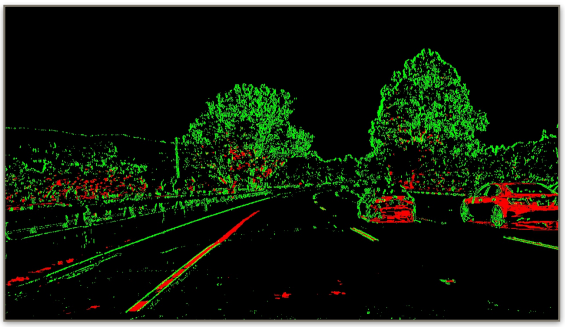
---

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.

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines 6 through 33 in thresholding.py). Here's an example of my output for this step:

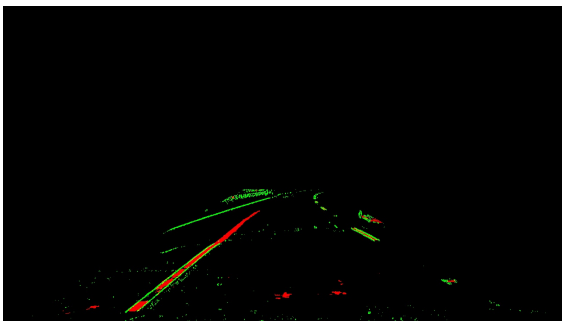


Original image



Thresholded image

I then applied a Region of Interest to mask the area I am not interested in (function *region_of_interest* in thresholding.py):



ROI applied

---

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 is a part of the long pipeline function *process_image()* in pipeline.py, which appears in lines 54 through 65. The code includes definition of source (*src*) and destination (*dst*) points and enabling visualisation for debug purpose.

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:

src *points connected with lines (red)*

dst *points connected with lines (red)*

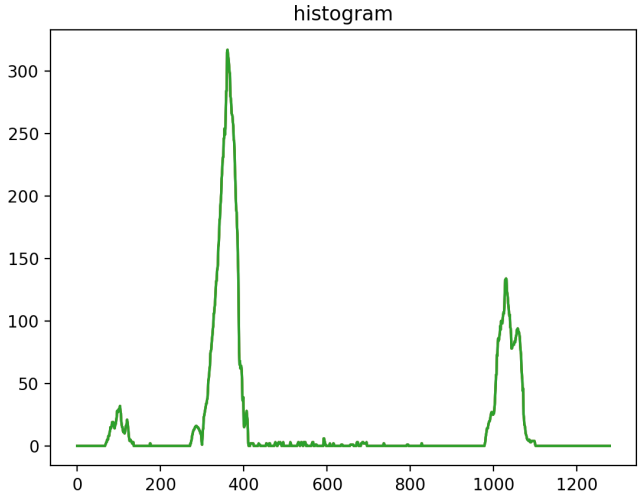
Another example of curved lane lines after the perspective transform:

Perspective transform on a binary image

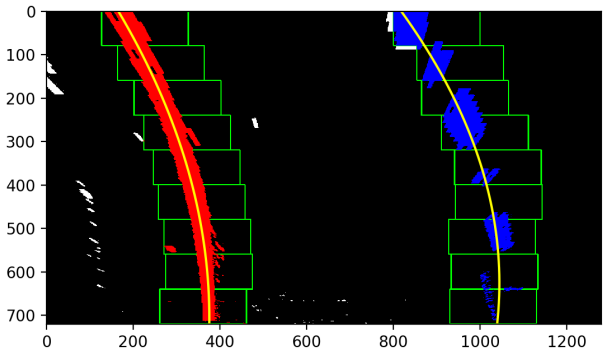
---

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

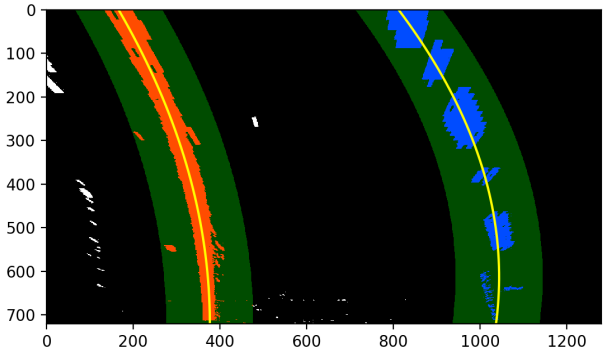
To identify line pixels I created a histogram of the lower half of the image to have a good starting point for the search. The peaks in the histogram are the lines positions:



Then I used the sliding window search method described in the lectures. I used 9 windows of 100 pixels width. Once at least 7 segments per left and right lines were detected I concluded it is enough to fit the polynomial. Then the next frame could be searched based on previous results within a 100 pixels margin without window search. If we cannot fit polynomial at some point there we return to the window search approach again in the next frame.



Window search initial frame



Next frame result

---

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.

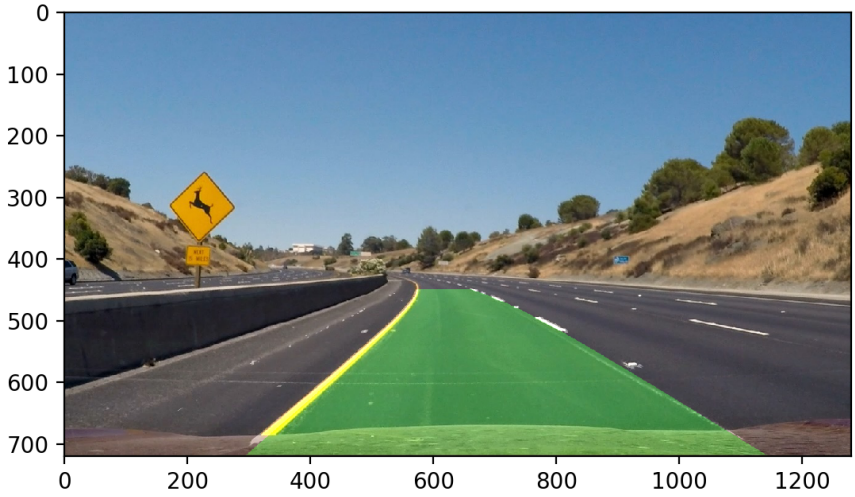
I did this in lines 216 through 237 in my code in pipeline.py. It is computed according the method described in the class material. Since we perform the polynomial fit in pixels and whereas the curvature has to be calculated in real world meters, we have to use a pixel to meter transformation and recompute the fit again. Additionally I shrank a bit the perspective transform image to better fit the lines into image and therefore a correction factor is used.

The mean of the lane pixels closest to the car gives us the center of the lane. The center of the image gives us the position of the car. The difference between them is the offset from the center.

---

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 lines 240 through 256 in my code in pipeline.py. Also the fine-tuning of the pipeline was done in finding_lines_and_radii.py. Here is an example of my result on a test image:



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!).

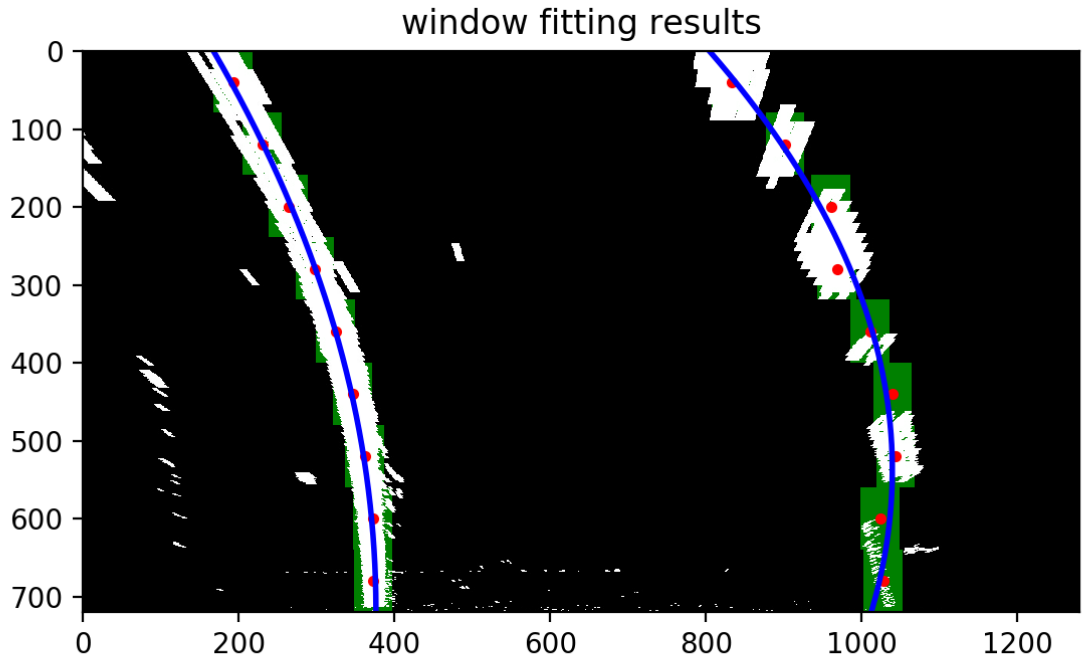
Here's a link to my video result (./project_video_processed.mp4)

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?

Most of the issues I faced were my bad Python programming skills related issues. Unfortunately I haven't invested yet time on making the line detection more robust and haven't included yet even some basic sanity checks and average filtering (running out of time). Although I tried an alternative method of line detection (convolution-based) that is also described in the lecture material (finding_curvature_convol_sl_wndw_method.py). The result of the window search and line fitting looks like that:



Further improvements are suggested in Project Rubric:

For an additional improvement you should implement outlier rejection and use a low-pass filter to smooth the lane detection over frames, meaning add each new detection to a weighted mean of the position of the lines to avoid jitter.

Such a need in improvement can be clearly seen after trying the challenge video. The current pipeline epically fails on 27% and the line detection is far from ideal. Here is the link to that video (challenge_video_processed.mp4).