Self-Driving Car Engineer Nanodegree Program

Advanced Finding lane lines Project

François Masson
October 22th, 2018

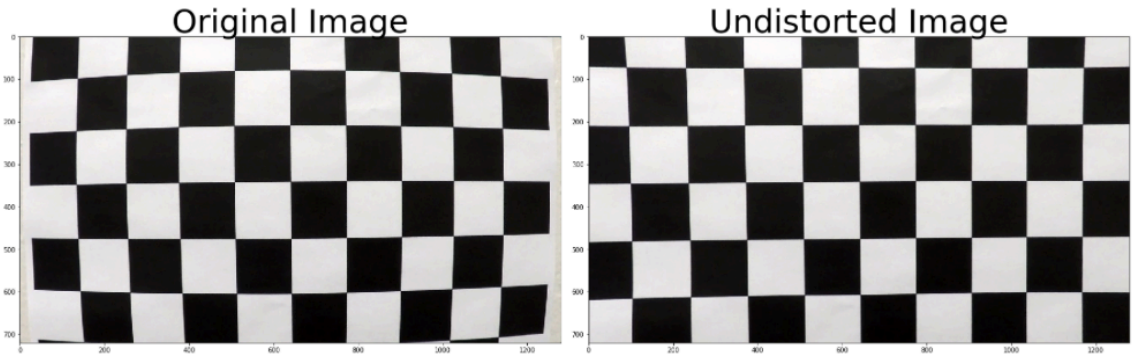
Pipeline Software

The pipeline used in this project covered the following steps:

- Camera calibration matrix and distortion coefficients given a set of chessboard images.
- Distortion correction to raw images.
- Color transforms, gradients, etc., to create a thresholded binary image.
- Perspective transform to rectify binary image ("birds-eye view").
- Detection of lane pixels and fit to find the lane boundary.
- 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 matrix and distortion coefficients given a set of chessboard images.

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



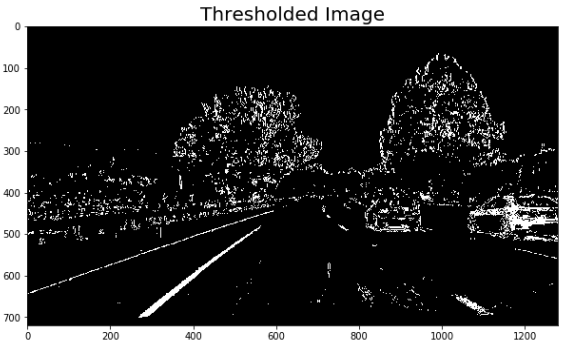
Distortion correction to raw images.

Having found the imgpoints and objpoints vectors, I compute again the camera calibration and distortion coefficients using the cv2.calibrateCamera() function but this time on sample image to remove any radial and tangential distortion from the image car such as below :

Color transforms, gradients, etc., to create a thresholded binary image.

In this step, I decided to use the Sobel in the x direction as a gradient. I also apply a conversion to LAB color space of the image and I separate the B channel before applying a threshold color channel. I did that because with the S channel I had shadow/light misdetection. The final result is actually a combination of the two binary thresholds. The hyperparameters applied were:

- Kernel of size 5x5
- Minimum threshold x gradient: 40
- Minimum threshold x gradient: 80
- Minimum threshold color channel: 150
- Maximum threshold color channel: 255



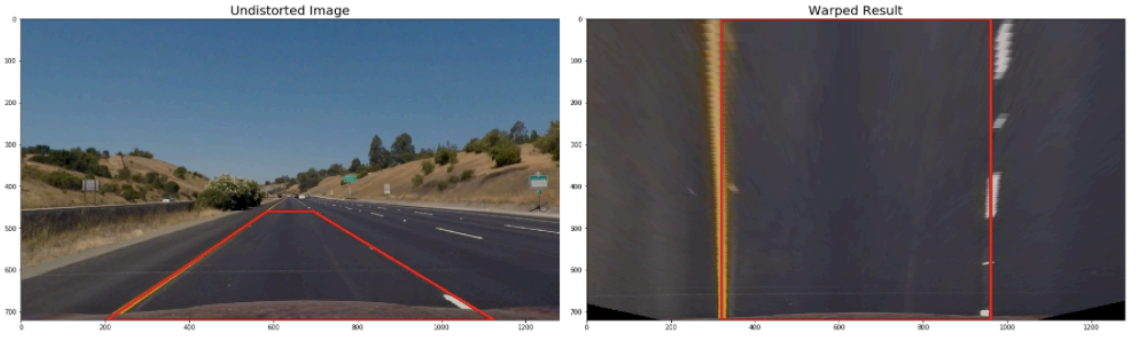
Perspective transform to rectify binary image ("birds-eye view").

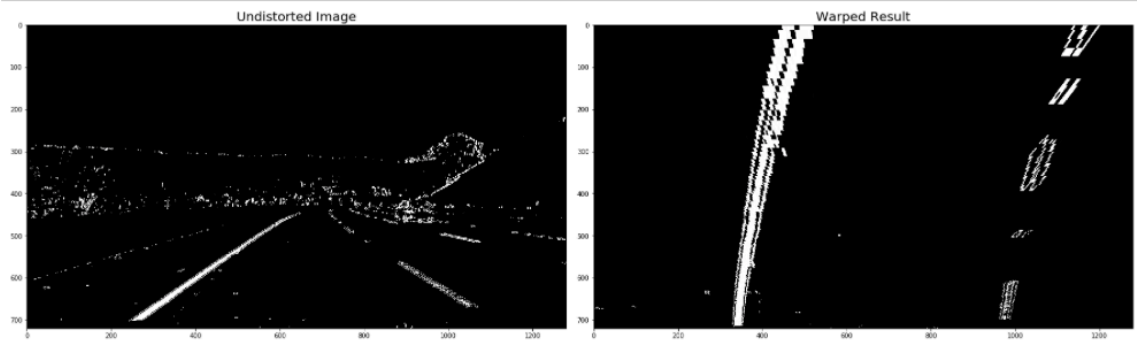
In this section, I used the warper function (function already provided) for my perspective transform. The warper() function takes as inputs an image (img), as well as source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

```
# Source point
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]])

#destination point
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]])
```

The provided results for a rectified image and a rectify binary image are given below:

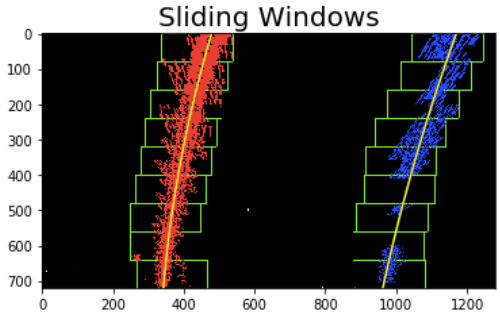




Detection of lane pixels and fit to find the lane boundary.

In this part, I implement the sliding window techniques. This technique may be described as:

1. Loop through each window in nwindows
2. Find the boundaries of our current window.
3. Use cv2.rectangle to draw these window boundaries onto our visualization image
4. Find out which activated pixels from nonzeroy and nonzerox above actually fall into the window.
5. Append these to our lists left_lane_inds and right_lane_inds.
6. If the number of pixels you found in Step 4 are greater than your hyperparameter minpix, re-center our window based on the mean position of these pixels.

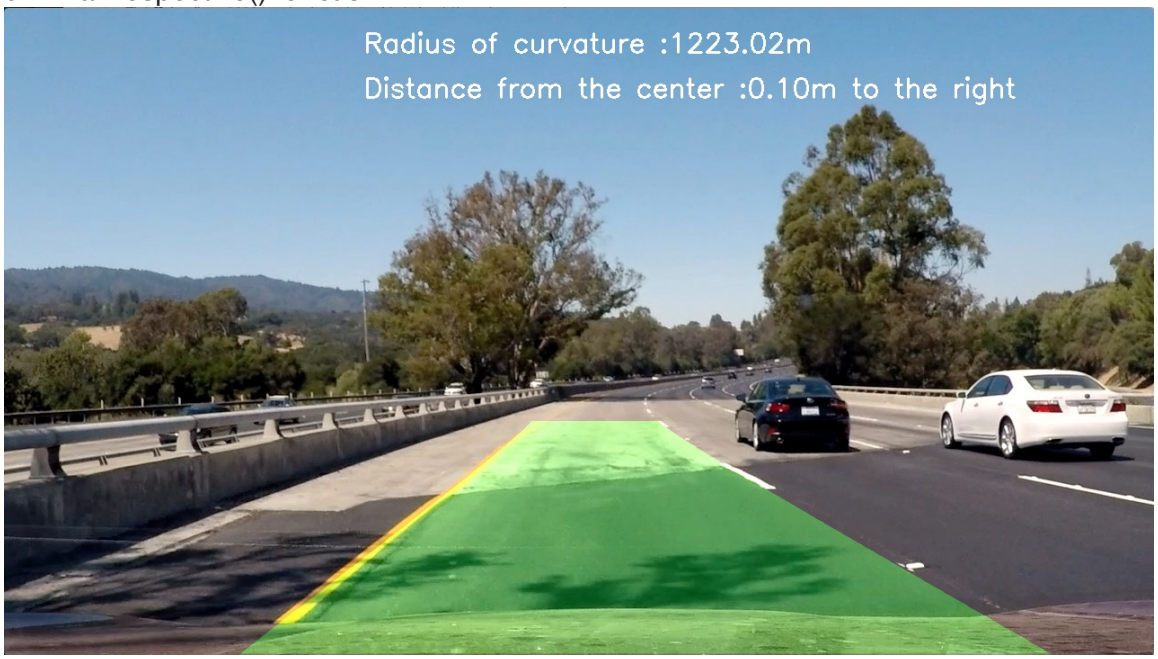


Curvature of the lane and vehicle position with respect to center.

Once I had polynomial fits, the radius of curvature was found thanks to the formula provided in the lesson. Using the appropriate resolution, the conversion from pixels to meters was possible. In order to determine the distance from the center, I took the left and right coefficients and calculate their average. The offset was just the result of the center of the image in the x axis minus the average. A negative value means an offset to the left. A positive one, an offset to the right.

Warp the detected lane boundaries back onto the original image.

Finally, in order to project those lines onto the original image, I warped the blank back to original image space using inverse perspective matrix (Minv) by using the cv2.warPespective() function



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

Here's a link to my video result:
https://github.com/FrancoisMasson1990/Self_Driving_Car_Engineer_Nanodegree_Program/blob/master/Car-Advanced-Lane-Lines/Test_Video/project_video.mp4

Discussion

In this project, I clearly see the importance of using previous frame and detection points to obtain a more robust algorithm. Only by finding the lanes from prior instead of using every time the sliding window, results were more accurate. Using a higher kernel size also smooths the detection.
However, some improvements have to be made: The calibrated points and the perspective transform don't fit anymore when there are up and downs roads. The software can not follow the road correctly (Harder_challenge.mp4). Finally, the Perspective transform technique using only 4 points to reshape a polygon to a rectangle provided some weakness.