Craig Vargas

CarND Project 4: Advanced Lane Lines – Write-Up

Camera Calibration

The camera was calibrated by using the built in opencv helper functions: findChessboardCorners and calibrateCamera. Due to the known uniform design of a chessboard, the functions can calibrate the camera by viewing a chessboard photo taken by the camera, analyzing the geometry of the chessboard corners, and calculating what type of adjustment must be made to set the chessboard corners from where they appear in the photo to where they should appear if the camera had no distortion effect. To make this process robust multiple chessboard photos were taken with the same camera at different angles and depths so that the algorithm could get a good idea of the distortion effects of the camera along most areas of the photograph.

Along with the chessboard photos, the findChessboardCorners algorithm needs to be told how many corners should be found in each photograph, and also needs instruction on how to map those corners from photo points to real-world points. The real-world points were self-defined to live in a 3D coordinate system (x, y, z) with z always equal to 0 (assuming no depth to the chessboard squares) and the x,y coordinates defined to be: (0,0), (0,1), … (0,num_cols -1), (1,0) … (1,num_cols -1), …. (num_rows -1, num_cols -1).

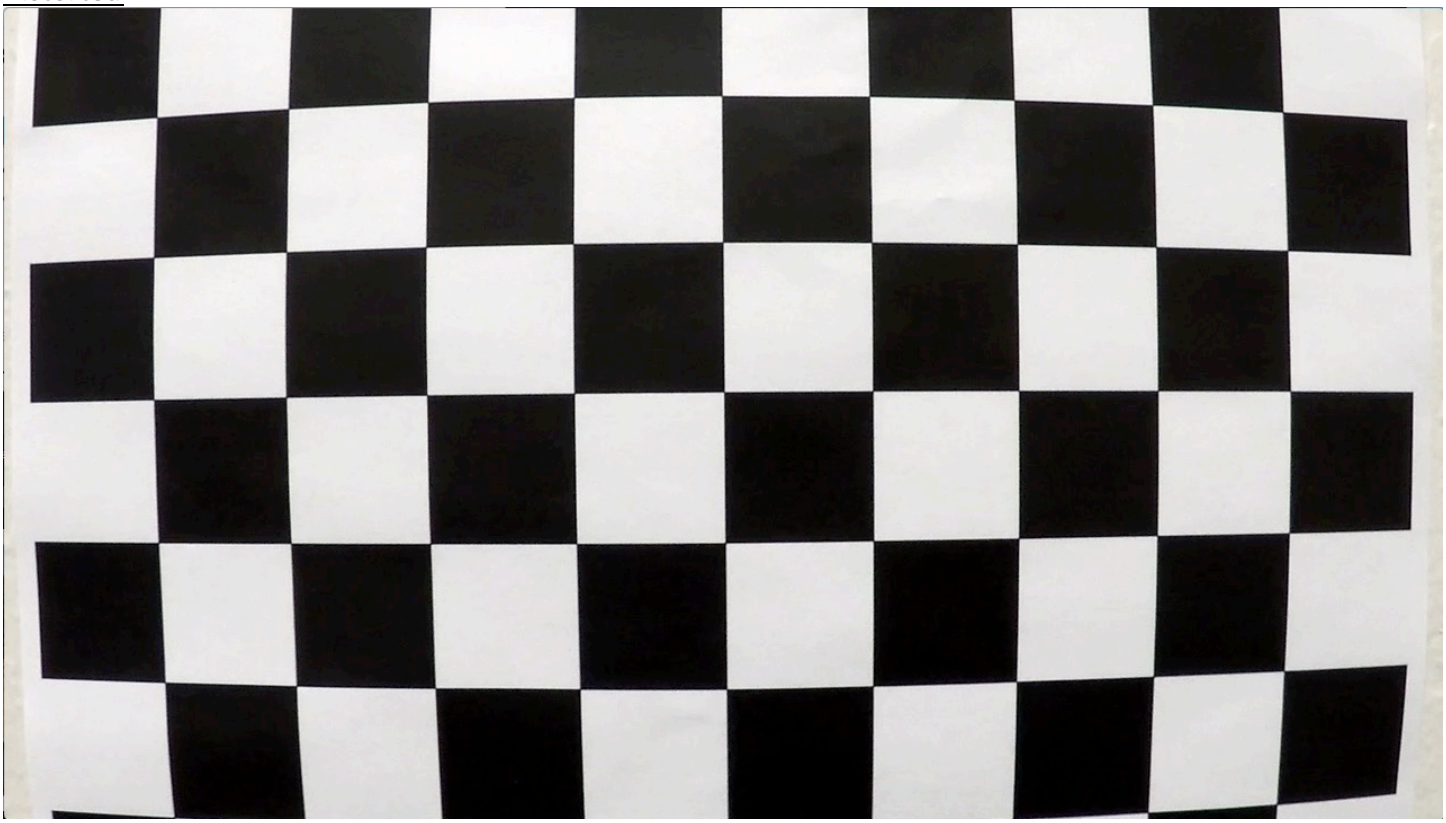
From the openCV documentation we see that the calibrateCamera function gives us information that can be used to undistort photos taken by the camera. In particular, we need the Camera Matrix, that contains intrinsic camera parameters for focal lengths expressed in pixels and an image point that is typically the center of the image. We also need the distortion coefficients which consist of both radial and tangential distortion coefficients.

The code for camera calibration exits in:

1. Identifying chessboard corners: calibrate.py – lines 38-54
2. Defining a set of real world points: calibrate.py – lines 32-33
3. Calibrating the camera: calibrate.py – line 66

Examples of the camera calibration photos:

Distorted



Undistorted

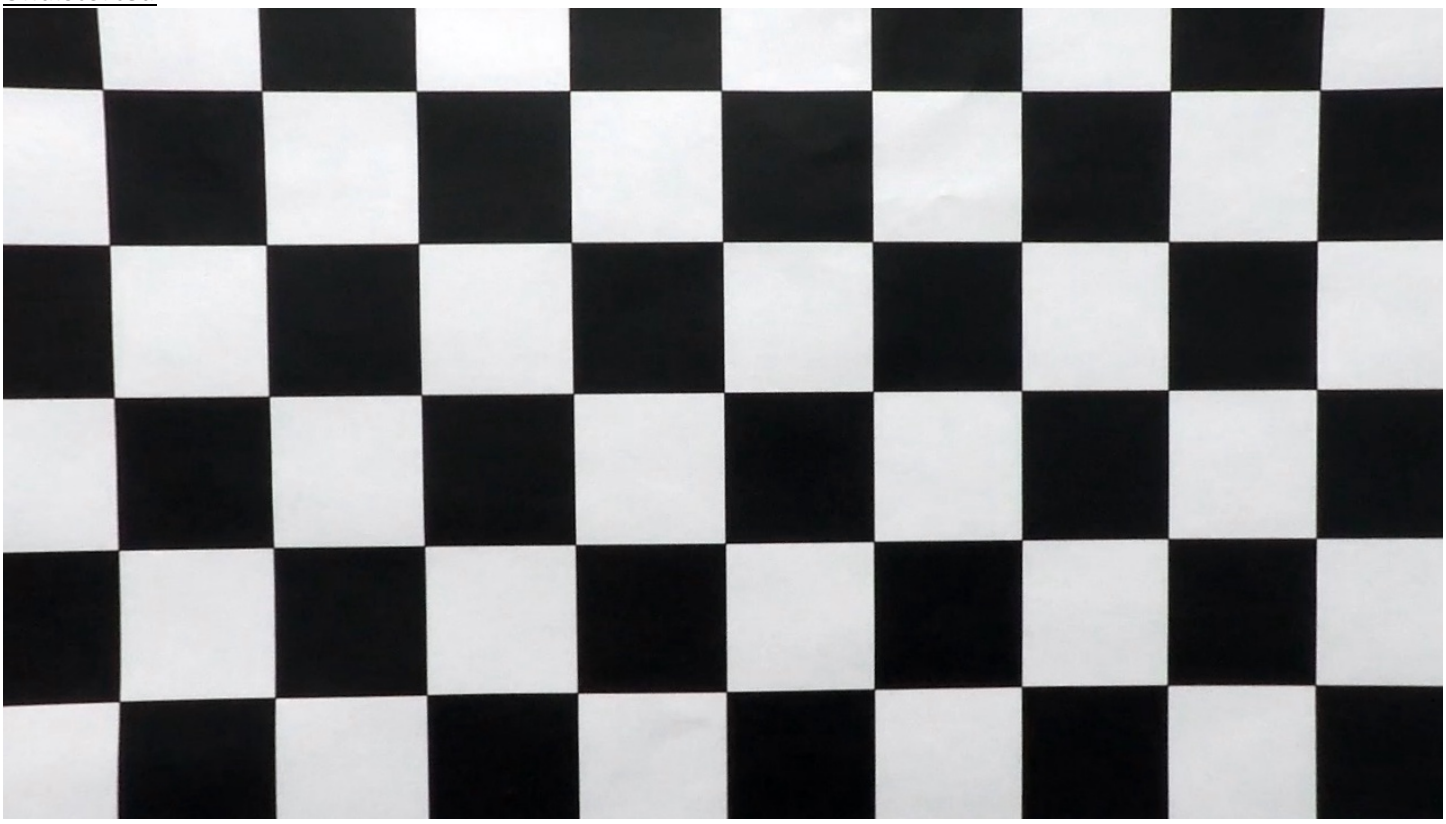


Image Processing Pipeline

Step 1: Undistort road image:

Using the camera calibration information collected above we first undistort the road image (gen-vid.py: lines 273-284)

Undistorted Image

Step 2: Create a binary image that removes most information aside from the lane lines:

Several image processing techniques were used to create the binary image:

1. Isolation of the 'S' channel in an HLS image (cv_utils.py – lines 62-70):
    a. After much experimentation with color spaces the 'S' channel in the HLS representation of an image showed a lot of promise in capturing the most of the yellow lane lines information as well as a good amount of the white lane lines. This method alone had its drawbacks of losing too much white lane line information when the threshold was high and adding too much road lighting noise when the threshold was low. I decided to keep the threshold high, remove the road lighting noise, and use other methods to get the white lane information.
2. Isolation of the 'V' channel in an HSV image (cv_utils.py – lines 75-83):
    a. The 'V' channel in HSV was helpful in gaining more information about the yellow and white lines

3. Isolation of the ‘b’ channel in the Lab color space (cv_utils.py – lines 100-108):
    a. The ‘b’ channel in the Lab color space was helpful in gaining a lot of information about the yellow lines on the road
4. Isolation of the ‘L’ channel in the Luv color space (cv_utils.py – lines 113-121):
    a. The ‘L’ channel in the Luv color space was helpful in gaining information about the white lines that were located further away from the car on the far end of the image.
5. Sobel operations (cv_utils.py - lines 11-25):
    a. The Sobel operator can find rate of change in pixels analogous to a differentiation operator. OpenCV allows you to apply the Sobel operator in either the x or y directions. Lane information showed up in both directions since lanes appear diagonal in camera images.
    b. It was also observed that keeping the filter size smaller removed a lot of noise around the rate of change in pixels.
    c. Operating on the image in the x and y direction alone added good lane info but also added too much other road noise. Throwing out information that was not contained in both the x orientation and the y orientation versions of the Sobel processing proved to remove a decent amount of the road noise that could distract or lane finding algorithm. (gen_vid.py - line 146)
6. Combining information (gen_vid.py - line 259):
    a. The information from the HLS color channel isolation as well as the Sobel operations seemed to provide some overlapping data but still a good amount of non-overlapping data. I decided to keep the information from both to retain as much lane information as possible.

Example of a binary image

Step 3: Performing a perspective transform to obtain a “bird’s eye view”

Using openCV functions getPerspectiveTransform and warpPerspective I was able to transform the image to obtain a birds eye view. The Function getPerspectiveTransform takes a set of source points (points on the original image) and a set of destination points (points on a new image where the source points should be placed) and returns a mapping matrix that is used to “warp” the image into a different point of view.

The purpose of this stage in the processing is to obtain a view of the lanes where the lines appear parallel to each other. The source points were calculated by manually eyeballing points on the top and bottom of the left lane line as well as points on the top and bottom of the right lane line. The destination points were selected so that the lane would take up a majority of the “warped” image with some margin on the left and right side that would be helpful for later identifying lane lines.

In this part of the pipeline you have to make sure that your lane lines in the warped images are roughly parallel or that they do not intersect and would not intersect anywhere in your image if the image were extended further. Code for this part of the pipeline exits in gen_vid.py lines 288-340.

Chosen source points:	SRC	DST
Top Left:	(575, 460)	(250, 0)
Top Right:	(705, 460)	(1030, 0)
Bottom Left:	(255, 660)	(250, 720)
Bottom Right:	(1040, 660)	(1030, 720)

Example of a warped perspective

Step 4: Identifying lane pixels and fitting a polynomial (gen_vid.py lines 343-499)

Lane pixels were identified by creating and algorithm that searched in an area most likely to contain the lane pixels and marked any pixels in that area that were “on” or that displayed white since we were working with a binary image at this point in the pipeline.

First I had to find the area that was most likely going to contain the lane pixels. This process started off by taking a portion of the lower end of the binary image and summing up all of the “white” pixels found in each pixel column of the image. The purpose of this was to take a portion of the image where the lanes would appear roughly straight and accumulate as many white pixels that could be found vertically in each position. The assumption here is that pixel column with the most “white” pixels was likely to contain a lane line. I then started in the middle of the image because that was an easy way to separate the left and right lanes, assuming the car is in the middle of a lane to begin with, and established the left pixel column with the most “white” pixels to be the start of the left lane line and vice versa for the beginning of the right lane line. Summing up the pixels was done using the numpy.sum function (gen_vid.py line 356). Finding the beginning of the left and right lanes was done using the numpy.argmax function (gen_vid.py lines 363-364).

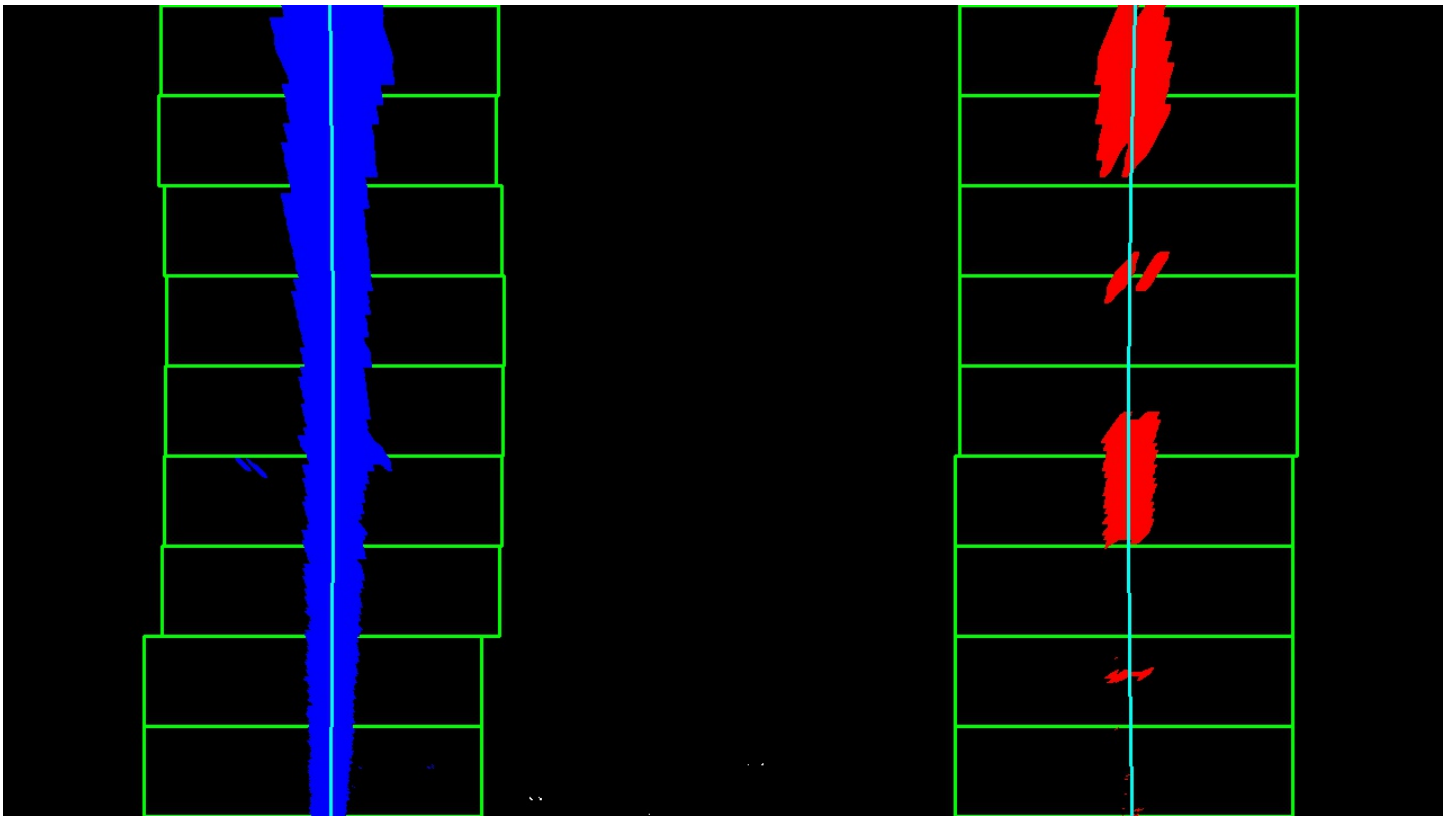
Next I divided the image into nine windows, used the numpy.nonzero function to obtain indices for all non-zero pixels, and iteratively searched localized areas of the image for lane pixels and saved those pixel locations to an array. The iterative process started at the base of the left and right lane lines found above and created a window to search for lane pixels that was 80 pixels high (720 image height/ 9 windows) and 300 pixels wide which was defined by a margin parameter that was decided on using trial and error. The smaller the window the less likely it was to pick up noise from other white markings on the image but also the more likely it was to miss lane pixel markings as the lane moved during a curved portion of the road. Once lane pixels were found, if the amount of pixels found were equal to or greater than a selected threshold (6000 pixels) then the base of

the lane lines were updated to be the average of the 'x' position of those recently found pixels. Those pixels that were found were saved to an array and the iterative process began again with the new left and right lane positions and a new set of search windows that was defined to be just above the last set of search windows. (gen_vid.py lines 388-410)

Once the lane pixels were located two types of polynomials were fit to those pixels using the numpy.polyfit function. The first polynomial was fit using the exact image pixel locations so that the points on this polynomial could be used to draw lane lines (gen_vid.py 416-424). The second polynomial was fit using points that were scaled from image pixel locations to real-world distances (gen_vid.py lines 426-439). The scaling factor was chosen by using known distances between lanes and the length of lane lines within the view of an image. The former value comes from a U.S. standard while the latter value was measured and given to us in a Udacity lecture. The mapping of pixels to distances was done manually using an "eyeballing" method. This second polynomial is used at a later stage to measure curvature.

Lastly the points given by the polynomial fit to the image pixel locations were formatted into an array that described the coordinates of the left and right lanes in a manner that would allow someone to "connect-the-dots" and draw a polynomial that defines the driving lane (gen_vid.py lines 455-465). The coordinates were then fed into the openCV function fillPoly in order to draw a shaded region that enclosed the car's driving lane.

Example of lane pixels and the polynomial fit to them



Step 5: Calculating the radius of curvature and car position

Radius of curvature was calculated using the real-world points polynomial that was fit in the previous step along with help from a known formula for calculating the radius of curvature from a point on a polynomial:

Radius = [[1 + (dy/dx)^2]^(3/2)] / [abs(d2y/dx2)]

Where d2y/dx2 is the second derivative of the polynomial

Polynomial form: Ay^2 + By + C

dy/dx = 2Ay + B

d2y/dx2 = 2A

The curvature of each lane line is calculated in gen_vid.py lines 431-439

The location of the car was calculated by assuming the center of the image's width was the car's horizontal location. The car's location was compared to the location of the center of the lane. The location of the lane's center was defined by averaging the values given by the polynomial, fit to the image pixels, valued at the bottom of the image (gen_vid.py lines 447-453). In my implementation negative(positive) values mean the car is located to the left(right) of the center of the lane.

Step 6: Displaying the lane and location information.

Drawing the lane on top of the original image was done with help from openCV's add weighted function which stacked the filled polynomial created in step 4 on top of the original image. I then used openCV's putText function to write the curvature and location values that were calculated in step 5. Code for this portion of the pipeline exists in gen_vid.py lines 102-132.

Example of final result

Pipeline video

Link to my video result:
https://youtu.be/a_UVNSed4rk

File: tracked.mp4

Discussion

There were a couple of areas in my pipeline that caused problems early on when testing my line finding algorithm.

Binary Image Creation:

The first big challenge was creating the binary image. This stage of the pipeline is very important for later stages. The key here is to make sure your binary image does include as much information as possible about the lane lines while also excluding as much noise as possible that exists inside of the lane and just outside of the lane as well. Any other noise further away from the lane is not a worry for the latter stages of the pipeline. A lot of trial and error was needed here to get a robust amount of information.

Experimenting with the color spaces was great early on and seemed to find most of what I needed for the lane lines but when testing with other images it was clear that variations in lighting on the road caused the color space filtering to pick up a lot of noise in the middle of the lane and this threw off the lane finding mechanism significantly. I was forced to increase the threshold of the color space filter which reduced the noise but also reduced a lot of the dashed lane line information. To compensate for this loss of information I had to add in the Sobel filter to find where pixels were changing in value. Similar problems occurred here where low thresholds included too much noise inside of the lane but higher thresholds took away too much of the lane information. Since the lane lines appear diagonal in the image they should be picked up by the Sobel operator applied in both the x and y directions so I was able to reduce some of the noise inside of the lane by requiring information to exist in both the Sobel-X and Sobel-Y filters.

Furthermore, it seemed that most of the information obtained by the Sobel operator was independent of the information obtained via the color space filtering so combining the two sources of data gave me a binary image that was robust enough.

Since this step is so important I would like to go back and try to combine more methods and different thresholds to see if I can get any more information.

Lane Averaging

When analyzing my video, it appears that the lane lines get a little wobbly in some portions of the video. In the next iteration I would like to experiment with averaging a few of the most recent lanes found to output smoother lane transitions. A lot of that wobbly noise was caused by loss of information in the pipeline and if values were averaged a small change in the lane shape caused by noise would not influence the output as much. However, a larger change cause by a real shift in the lane curvature would have a bigger impact in the averaging and if it is a real change in lane curvature then, as more of that similar lane structure was fed into the system in later frames, the average would model the new structure more accurately.