Sebastian Nietardt

Udacity Self-Driving Car Nanodegree

Project 4

April 14, 2017

Advanced Lane Finding

Camera Calibration

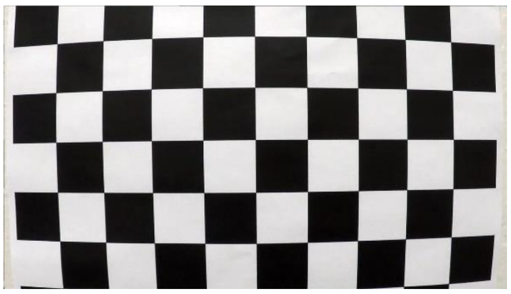
Camera Calibration

The code for finding the image points and camera calibration can be found in the second cell of the jupyter notebook. The code for the distortion correction can be found in the fourth cell of the jupyter notebook. The code of the fourth cell is called from the ninth cell of the jupyter notebook.

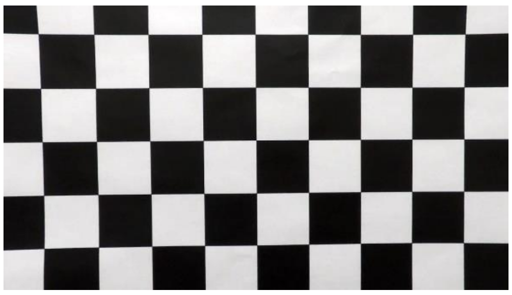
The first step in the project was to calibrate the camera. The camera matrix and distortion coefficients have been calculated using OpenCV. First, the object points needed to be defined as (x,y,z) coordinates of the chessboard corners. The z-values were assumed to be always 0. The chess boards have 9 corners in x direction and 6 corners in y direction, so the object points reached from (0,0,0) to (8,5,0). The image points have been calculated using the cv2.findChessboardCorners() function on the gray scaled images. The calibration has been done using the cv2.calibrateCamera() function.

After camera calibration, the distortion correction has been applied to the chess board images using the cv2.undistort() function. An example of the distortion correction can be seen in the following image:

Distorted chess board image:



Distortion corrected chess board image:



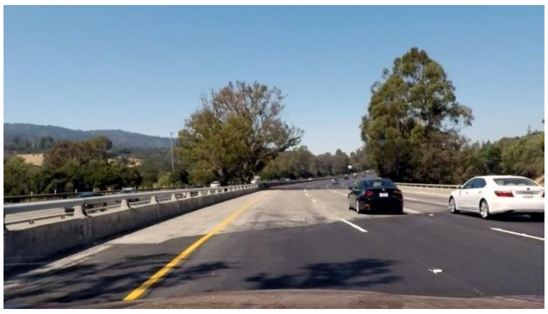
Pipeline (test images)

Provide an example of a distortion-corrected image.

The distortion correction for the test images is the same as for the chess board image by calling the cv2.undistort() function with the matrix and distortion coefficients. The code can be found the fourth cell of the jupyter notebook. This time it is called from the findLaneLines() function in the eleventh cell, which is called from the twelfth cell for test images. The following images show an example:

Distorted test image:

Distortion corrected test image:



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.

The code for the binarized image is located seventh cell of the jupyter notebook. It is called from the findLaneLines() function in the eleventh cell, which is called from the twelfth cell for test images.

To binarize the images, a mixture of the lightness value (for daylight images), saturation value and a Sobel operator value has been used. The image has been converted to HLS values and the lightness and saturation values were extracted. The lightness value has been used to filter out shadows during daylight and has been disregarded if the image was not taken during daylight. If the average lightness value of the image was higher than 60, it has been assumed that the image is a daylight image.

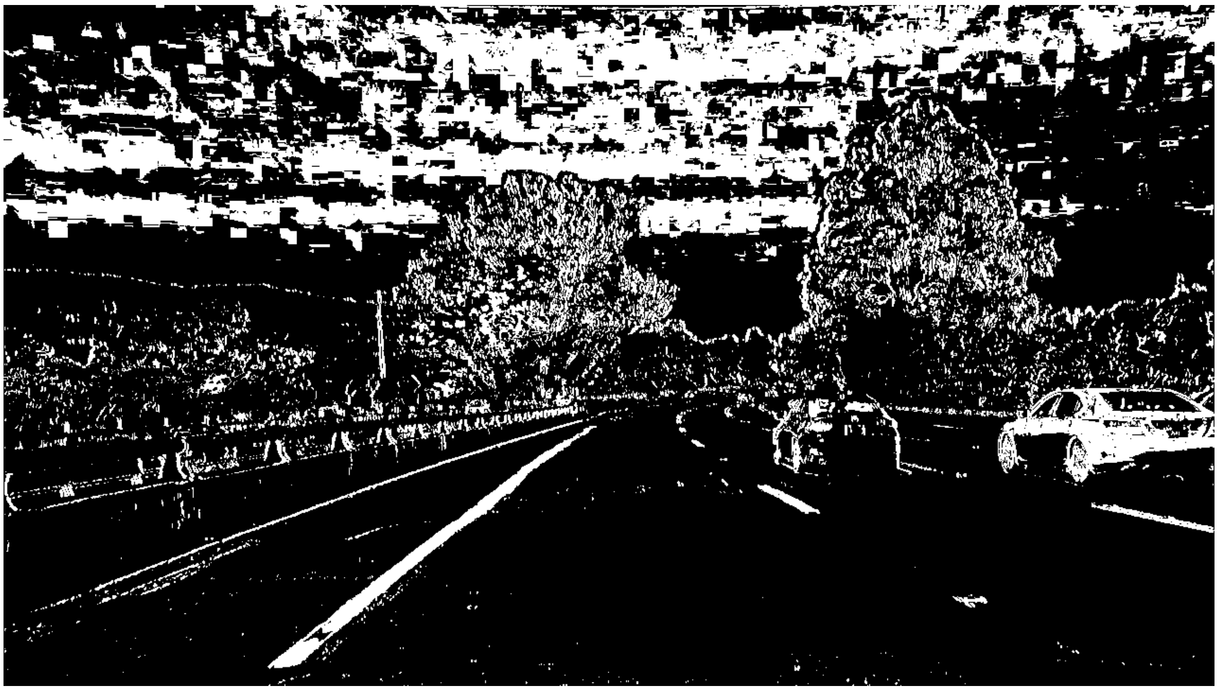
The thresholds for lightness have been set to a minimum of 0 and a maximum of 30. Using these thresholds, a binarized image with very low lightness values has been created.

For the saturation values the minimum threshold has been set to 120 and the maximum threshold has been set to 255. Using these thresholds, a binarized image of saturation values has been created.

The minimum threshold for the Sobel operator has been set to 20 and the maximum threshold was set to 255. Using these thresholds, a binarized image of the derivative values has been created.

To generate the final binarized image, the binarized saturation image and the binarized Sobel operator image where combined. Each pixel where at least one of the binarized images had a value greater than 0, the value of the final binarized image has been set to 255. In order to filter out shadows in daylight images, the pixel value of the final binarized image had been set back to 0 if the binarized low level lightness images had a value greater than 0 for this pixel.

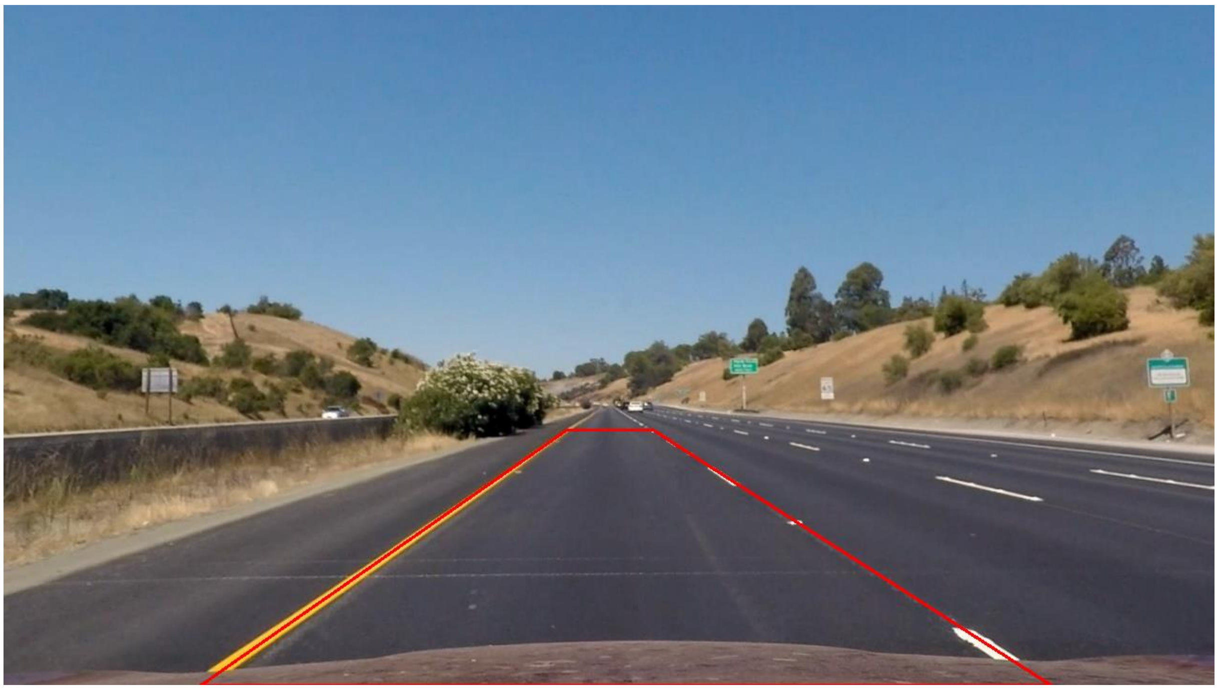
The following example shows a final binarized image.



Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

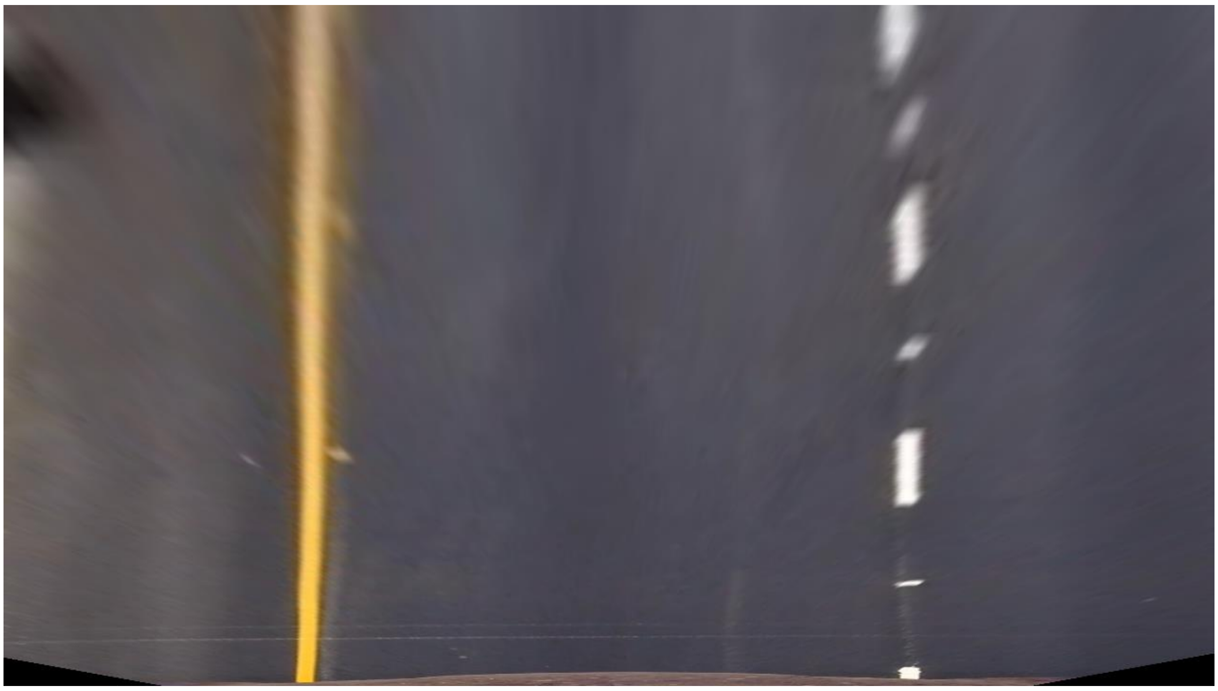
The code for the perspective transform is located in the fifth cell of the jupyter notebook. It is called from the findLaneLines() function in the eleventh cell, which is called from the twelfth cell for test images. The values for the polygon can be found in the tenth cell of the jupyter notebook.

The perspective transform of the images has been done by calculating the perspective transform matrix (using the cv2.getPerspectiveTransform() function with source and destination values) and handing it over to the cv2.warpPerspective() function. For the source and destination values, a polygon had been created. The values for the polygon have been generated manually, by measuring the pixels of a lane in an image where the lane goes nearly straight. The following image shows the polygon on a straight lane.



Since the lines are not always exactly on the polygon, an offset for the left and right side of the lane needed to be defined as well. The following table shows the coordinates of the polygon and the following image shows the transformed version of the image above.

Source	Destination
(595,450)	(320,0)
(208,720)	(320,720)
(1107,720)	(960,720)
(686,450)	(960,0)



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

The code for the lane line identification is located in the eighth cell of the jupyter notebook. It is called from the findLaneLines() function in the eleventh cell, which is called from the twelfth cell for test images.

The first step to finding the lane lines was to define a sliding window, which can slide up along the lane line starting at the bottom of the image. The width of that window has been set to 50 and the height has been set to 90. Since the images have 720 pixels in height, the 90-pixel high window slices the image into 8 vertical layers. A margin for the sliding window has also been defined and has been set to 100. The window can use the margin to slide within a maximum of 100 pixels to the left or the right side to track the lane line.

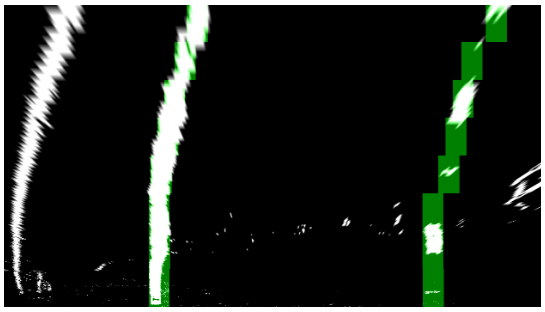
The search for the lines starts at the bottom of the image. It is assumed, that the left lane line can be found in the left half of the image and the right lane line can be found in the right half of the image. In addition, it is assumed that the lane line does not start at the left or right edge of the bottom. That is why just the inner half of the lane offset has been taken in account to find the start window. The x value of the start window has been defined as the highest vertical sum of the pixel values within the lowest quarter of the image.

After the start window has been found, the window starts sliding up and looks for the lane line pixels in the next higher level. But just within the defined left and right margin. If it does not find any lane line pixels on the next level, it either uses the x value from the former level or calculates a new x value taking the current curvature of the lane line in account.

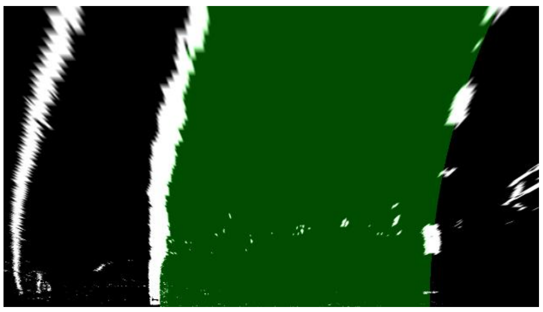
After all the lane line has been detected by the sliding windows, a polyline has been fitted through the pixels. The left and the right polyline has been used to create a filled polygon, which represents the lane line.

The following images show an example lane detection result. The left image shows the sliding windows, which detected the lane lines. The right image shows the detected lane, highlighted in green.

Perspective transformed image with sliding windows:



Perspective transformed image with highlighted lane:



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.

The code for the perspective transform is located in the findLaneLines() function in the eleventh cell (line 102 to 143) of the jupyter notebook. It is called from the twelfth cell for test images.

First, the pixel values have to be transformed to metric values. It has been assumed, that the lane is 3.7 meters wide and that the manually defined polygon covers about 26 meters along the lane.

That means the following formulas has been applied:
meter_per_pixel_in_y_direction = 26 / number_of_image_pixels_in_y_direction
meter_per_pixel_in_x_direction = 3.7 / (number_of_image_pixels_in_x_direction - offset_in_x_direction)

Now that the meter per pixel are known, a polyline $f(y) = Ay^2 + By + C$ can be fit through the metric values of the left line and the right line. With the new polylines, the curvature can be calculated using the formula $R_{curve} = \frac{(1+(2Ay+B)^2)^{\frac{3}{2}}}{|2A|}$.

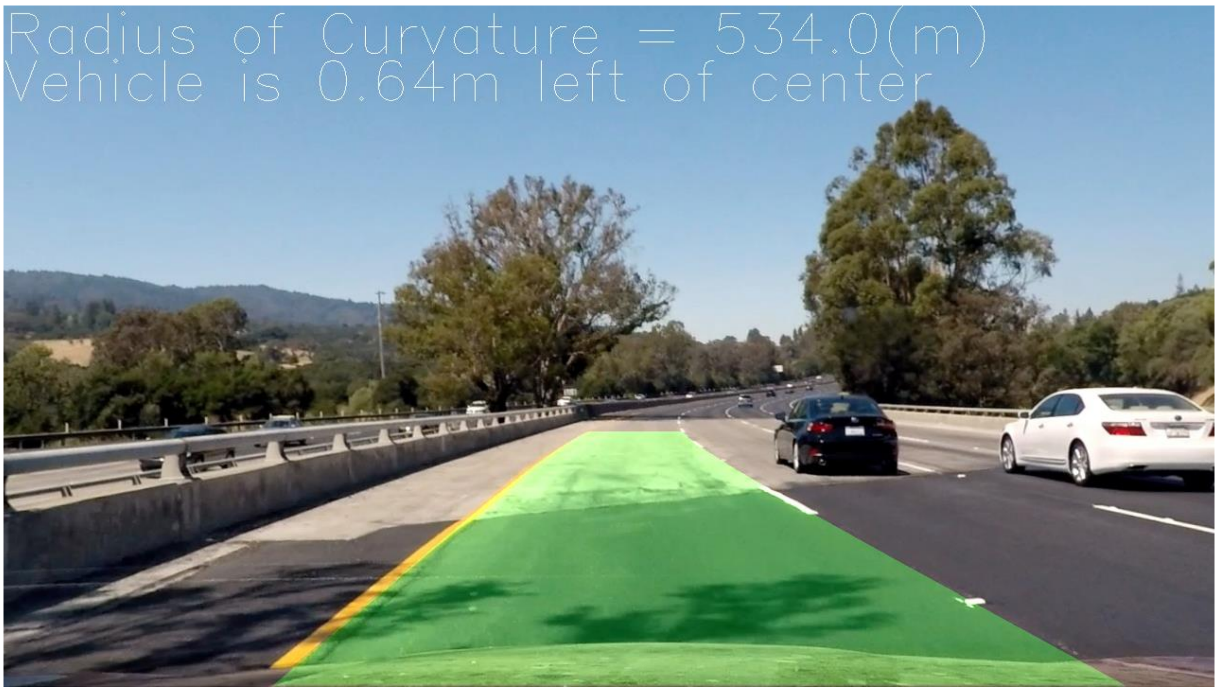
In order to calculate the position of the car on the lane, the start position of the left lane line and right lane line can be used. A positive value means the car is x meters of left and a negative value means the car is x meters of right.

$$Car_{Pos} = \frac{((right\_lane\_x\_value - \frac{image\_width}{2}) - (\frac{image\_width}{2} - left\_lane\_x\_value)) * 3.7}{(right\_lane\_x\_value - left\_lane\_x\_value)}$$

Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

The code for the transform back to the original perspective is located in the sixth cell of the jupyter notebook. It is called from the findLaneLines() function in the eleventh cell, which is called from the twelfth cell for test images. The green highlighted lane as well as the text for the curvature and the car positon are added in the function findLaneLines() directly.

The following image shows the final result of a test image.


Radius of Curvature = 534.0(m)
Vehicle is 0.64m left of center

Pipeline (video)

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!)

The code for creating the video is located in the fourteenth cell of the jupyter notebook. The video is located in the *output_images* folder of the project.

Link to the video

Discussion

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?

The first problem appeared by finding the right thresholds for the Sobel operator and the saturation values to binarize the image. It needed some calibrations to find a configuration which worked a on all the images. But the shadows still were a problem. I faced this problem by taking the lightness for daylight images in account. An image, taken during the night, should hardly have any shadows and the issue should disappear. I cannot say if the daylight image detection still works, as soon as some special places like bad illuminated tunnels appear where there is still a lot of daylight on the horizon.

Another problem was the lane detection. The start position could drift off when a car was passing by or the left road limit was mistaken for a line sometimes. I faced this issue by limiting the space of the possible start position to an area more in the middle of the image disregarding the edges. For the higher levels, I needed to guess the position of the line by its former direction, if no lane line pixel where found on that level.

The pipeline works fine for the project video, but starts failing with the more difficult videos. As soon as the road has a huge crack or color difference on the lane or the sun is reflecting in the camera, the pipeline is likely to fail. The algorithms and thresholds to generate binarized images should be improved to filter out more objects and detect the lane line in more cases. In addition to that, the curvature and the car position should be memories over several images in order to make better guesses when the lane line detection fails.