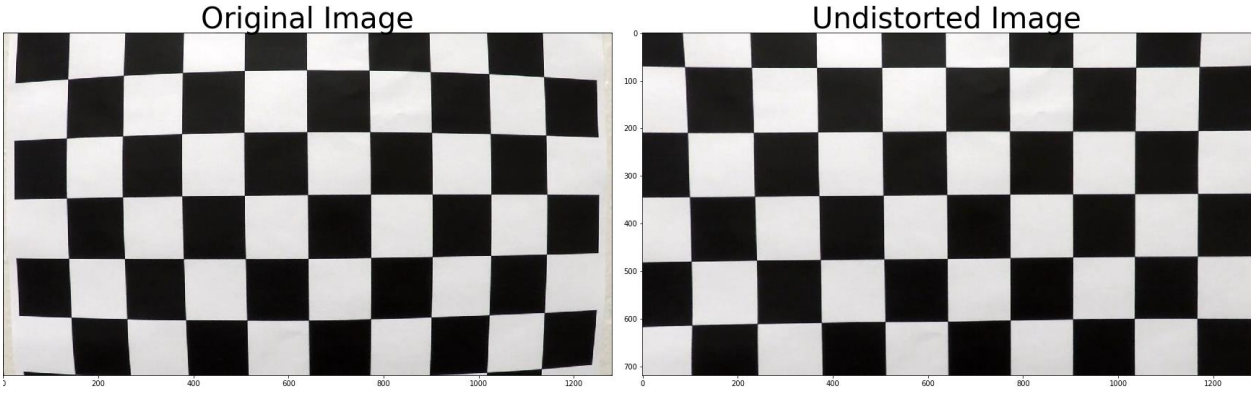
Udacity: Self Driving Car Engineer Nanodegree

Project 2 – Advanced Lane Finding

Submitted by Ayeda Sayeed

Camera Calibration

To begin the project, I had to first calibrate the camera that was used to capture the images and videos. I set up two objects to hold object points (the grid coordinates of detected corners) and image points (the pixel coordinates of detected corners). Next, for each calibration image available in the *"camera_cal"* workspace folder, I would read in the image of the chessboard, taken at a variety of angles and distances. I converted the image to grayscale so that the `cv2.findChessboardCorners()` function could detect the inside corners of the board; that is, the junctions of 2 black and 2 white squares. If corners were found in that image, the pixel coordinates of those corners are appended to the image points object, and the corresponding object points are appended to their object. Then, after all images' corners were detected and appended, I used `cv2.calibrateCamera()` on the accumulated image and object points to calculate the camera matrix and the distortion coefficients. The camera matrix and distortion coefficients are used in `cv2.undistort()` to remove any camera distortion and provide a true image. Below is an example of the original calibration image, and the undistorted version, performed on calibration image *"calibration1.jpg".*



This figure above can be found in the Jupiter workspace directory '/output_images', titled 'comparisontestjpg'.

This calibration was performed as a standalone procedure to provide the camera matrix and distortion coefficients. Once these values were found, I saved them as constants in my video pipeline so that calibration would not be performed with every frame of the video. `VideoFileClip` library expects all functions performed on videos to have only colour image inputs, so the camera matrix and distortion coefficients cannot be passed to each frame of the video.

Pipeline (test images)

I designed a pipeline for testing the advanced lane finding algorithm on individual frames/images, so that each image can be analyzed individually, allowing for troubleshooting and tuning of parameters based on the results. Thus, the pipeline handles one image at a time, instead of iterating through all of the images. This pipeline can be found in the Jupiter notebook "*Image Test Lane Finding.ipynb"*

1. Distortion correction

Having performed the camera calibration, we can use the camera matrix and distortion coefficients again with `cv2.undistort()` to perform the distortion correction of individual images. Below is an example of the distortion correction performed on *'test3.jpg'*.

This figure above (and the results from the other test images) can be found in the directory '/output_images', titled 'distortion_comparison_{original_file_name}.jpg'.

2. Thresholded Binary image

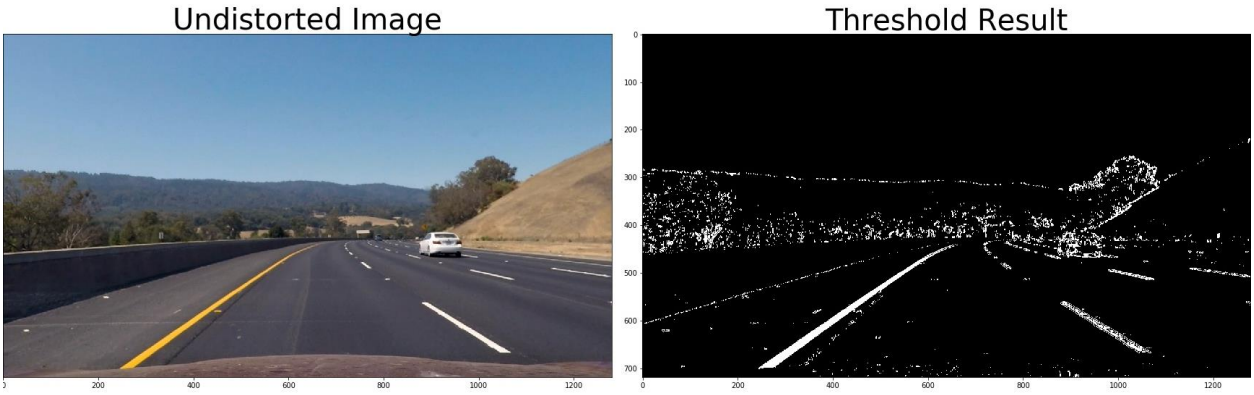
The first step to identifying lanes is to identify pixels that possibly make up the lane lines. I used saturation and gradient thresholds to identify possible pixels.

I chose saturation thresholds because of the consistent and bold colour and saturation of lane lines (bright yellow, bright white), making them have typically high saturation values. Things like trees are textured and consist of low to mid saturation pixels. By removing pixels with saturation values in the lower half (0-169), we are left with higher saturation pixels that could potentially be lane lines versus landscape features.

I chose to use gradient thresholds in addition to the saturation thresholds, to really pick out the pixels that define the outlines of high-saturation objects. By calculating the x gradient, I can identify pixels that outline vertical-leaning lines. I applied these thresholds to a lightness-channel filtered image since the lightness values do a reasonable job of highlighting distinguishable features in the image.

To implement these theoretical methods, I had to first convert the undistorted image into an HLS image, and isolate the lightness and saturation channels. I used `cv2.Sobel()` on the l-channel image, with a kernel size of 7 (to provide a smoother result). I then scaled the resulting x gradient to a range of 0-255, using the absolute value of the x gradient. To produce a binary image, I simply set all pixels that

fell within the threshold criteria to 1, and the rest to 0. I produced a binary image from the s-channel image in a similar fashion, then combined the two to produce a final image where both thresholds are true.



This figure above (and the results from the other test images) can be found in the Jupiter workspace directory '/output_images', titled 'threshold_comparison_{original_file_name}.jpg'.

3. Perspective transform

Because we are looking at the road from the front of the vehicle, the camera images and thus the lane lines are angled. It is easier to analyze and view the lane in a single, 2-dimensional plane. So, I implemented a perspective transformation to see the lane from an overhead view. First, I manually identified 4 points that outline the trapezoid that contains the majority of the lane, extending from the vehicle at the very bottom of the image, to almost the center of the image. The model for this point identification was the test image *"straight_lines1.jpg".* I then identified 4 other points that form a rectangle, centered in an image of the same size as the original, extending from the top to the bottom, and the same width as the bottom of the lane in the original view. The final points are shown in the table below.

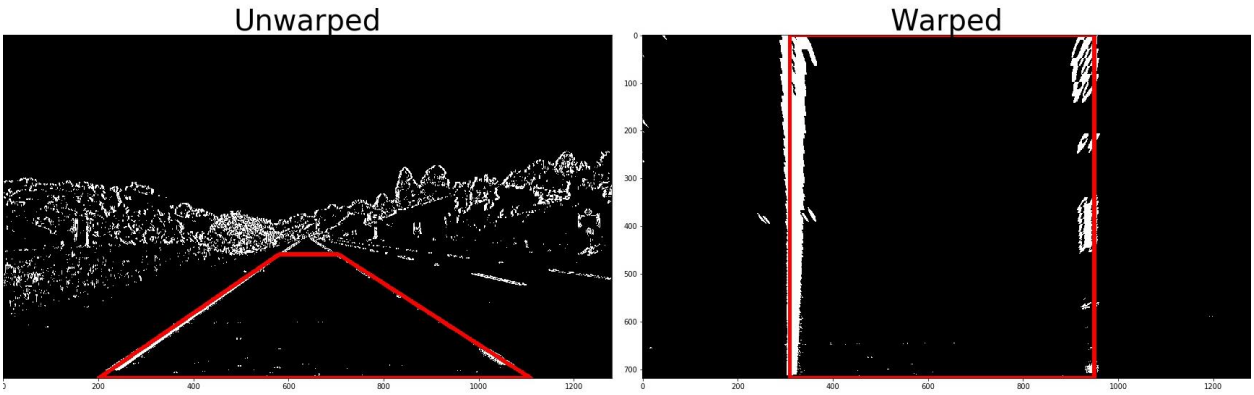
Source	**Destination**
200, 720	310, 720
580, 460	310, 0
705, 460	950, 0
1110, 720	950, 720

It's important to keep the points in a certain order, so that the corresponding source and destination pairs are matched when performing the perspective transformation.

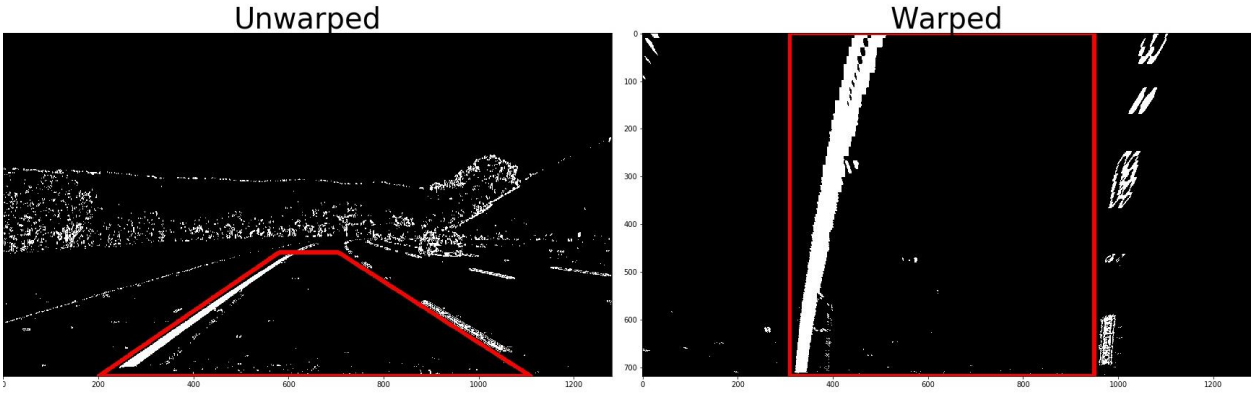
I used `cv2.getPerspectveTransform()` to find the matrix transform, `M`, that would convert the source points to the destination points, thus converting a front-view image to an overhead view image that contains only a small window around the region of interest. I also calculated the inverse transform, M_{inv}, that would convert pixels in an overhead image back to front-view space.

Finally, I used `cv2.warpPerspective()` with the perspective transform, `M`, to produce the overhead view of the region of interest of the lane. To ensure that my perspective transform was correct, I checked that the resulting lane lines were parallel.

The model for finding the points for the perspective transform, *"straight_lines1.jpg",* and the results of the transformation:



The transformation performed on *"test3jpg":*



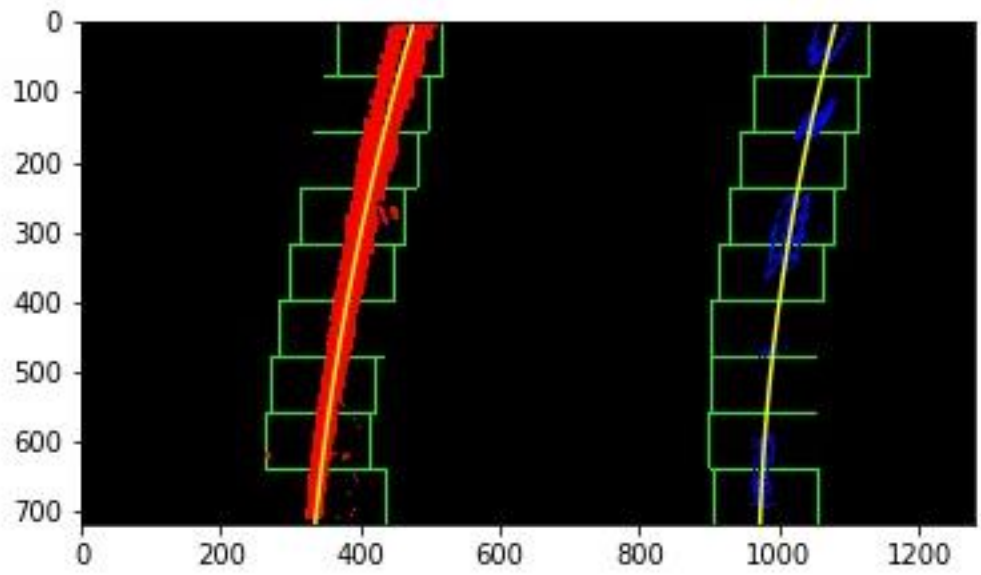
These figures above (and the results from the other test images) can be found in the Jupiter workspace directory '/output_images', titled 'warp_comparison_{original_file_name}.jpg'.

4. Detecting lane pixels and fitting polynomials

Now that I had an overhead view of the lane, it was time to detect the lane lines. To begin this process, I used the sliding window approach. First, the centreline of the overhead image was found, acting as a barrier to find the column in each half of the image that contained the most "activated" (or white) pixels. To find this column, a histogram was taken across the x axis of the image, and the bin or column with the highest values corresponded to the x-values of the image where the most white pixels reside. These two x-values were the basepoints to continue finding all of the left and right lane line pixels. Around this basepoint at the bottom of the image, I found all activated pixels within a prescribed window and appended their coordinates to a list of good indices. If at least 50 pixels were found, I found the mean x position of these pixels. This became the basepoint (and centreline) for the next window,

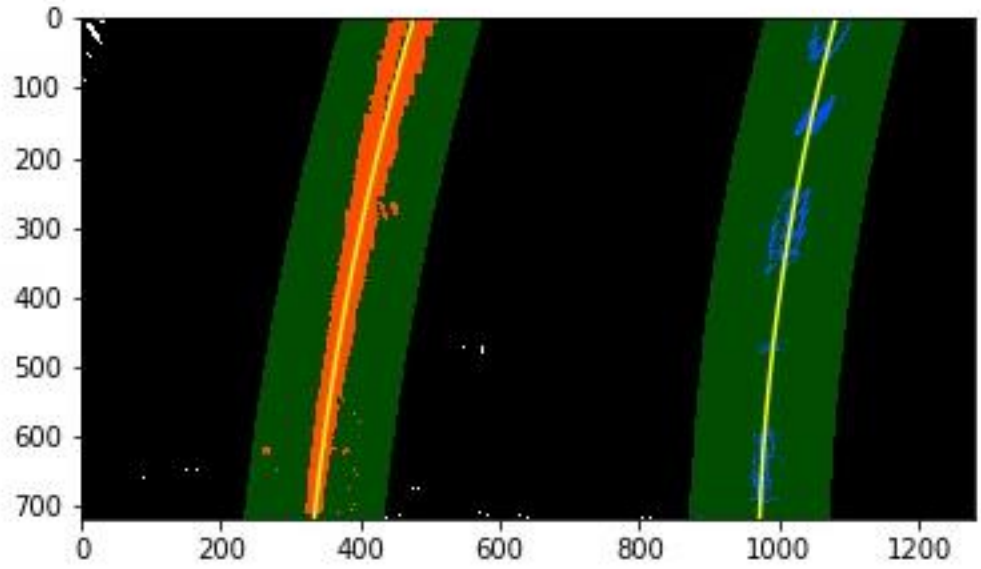
that is stacked on top of the previous window. The process continues until the windows have traversed from bottom to top of image. The final result is a list of the x and y values of all of the activated pixels found in the windows, for each the left and right lane lines. This process is defined in the function `find_lane_pixels()`.

With the final lists of the possible lane line pixels' x and y values, I used `np.polyfit()` to fit a 2nd order polynomial line to the pixels and form a solid line for each the left and right lane lines, where y is the independent value since we know the lane must extend from the very top to the very bottom of the image. Using the coefficients of these polynomial, I calculated the single point pixels that would belong to said polynomials, thus resulting in a line pixel for each y pixel in the image. This polynomial and its pixels can be drawn on the warped image to show where the lane lines lie. The process is defined in the function `fit_polynomial()`. Below is the image showing the windows used to find the lane line pixels, and the resulting polynomial lines.



This figure above (and the results from the other test images) can be found in the Jupiter workspace directory '/output_images', titled 'sliding_window_{original_file_name}.jpg'

In this standalone image test, I applied a margin around these polynomials and found the corresponding pixels in a similar fashion to the sliding window approach, thus refining the search for lane pixels and producing more accurate line polynomials. This process is defined in `search_around_poly()`. The image below shows in green the margin around the previously found polynomial, and plots the newly found polynomial.



This figure above (and the results from the other test images) can be found in the Jupiter workspace directory '/output_images', titled 'searchfromprior_{original_file_name}.jpg'

In the video pipeline, the sliding window approach is used in the first frame only. The resulting polynomials are used in `search_around_poly()` on the next frame, and every subsequent frame from then on uses the previous frame's polynomials to refine its search for polynomial pixels.

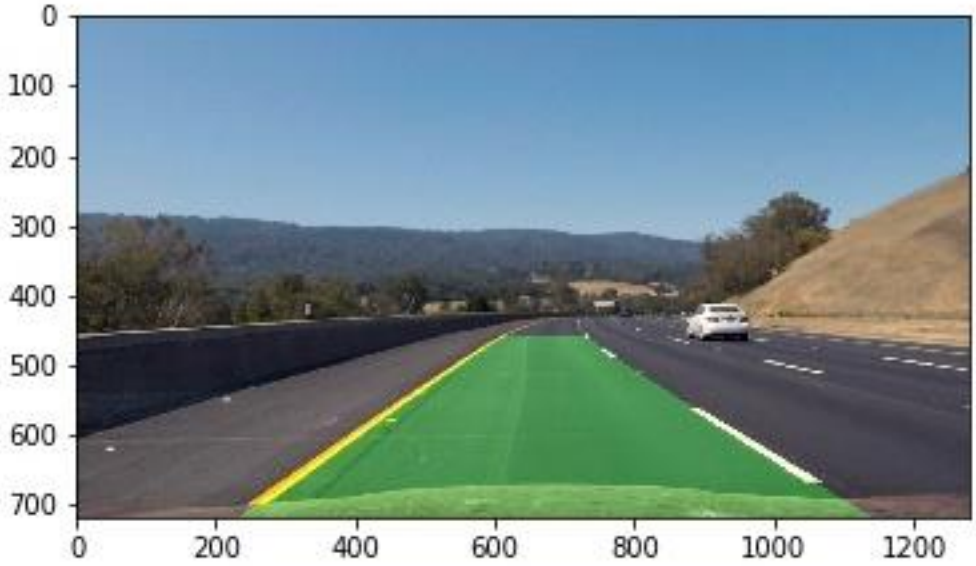
5. Radius of curvature and vehicle position

Calculating the radii of curvature of the left and right lane lines was defined in the function `measure_curvature_real()`. First, the pixel values were converted to real-life distances, in meters, and the corresponding new polynomials were calculated. Using these polynomials and the equation for the radius of curvature, the radius of each lane line was calculated.

To find the vehicle position from the center of the lane, I simply took the mean of the x values of the fitted polynomial as the position of each lane line, calculated the center between these x's, took the difference from the image center, and multiplied by the pixel-to-meters conversion factor. This process was defined in `find_vehicle_pos()`.

6. Result with identified lane

Finally, I had to draw the identified lane onto the original image, in the vehicle view. I created an empty image, `color_warp`, to hold the pixels that occupy the entirety of the lane in the warped view. I created an array of the left and right lane lines pixel coordinates, and used `cv2.fillPoly()` to fill the pixels between the left and right lane pixels with the colour green. I then used cv2.warpPerspective and the inverse transform, M_{inv}, to "unwarp" the green lane, and overlay it onto the undistorted vehicle view image using `cv2.addWeighted()`. The final result is shown below, with the identified lane coloured in green:



I also tested the text overlay, writing the vehicle position and the average of the two radii of curvature to the top of the image, shown below:

Pipeline (video)

The procedure for advanced lane finding on a video can be found in the Jupiter notebook *"Video.ipynb",* the resulting videos for which can be found in the directory *"/output_videos"*. The video *"project_video_output.mp4"* is the required submission.

In order to use the prior frames' information, I had to create a class, `Line()`, which I could update with current lane line information for each the left and right lane line, and access prior lines' data. With at least 10 frames of data, I average the best fits of each lane line and use these averages as the polynomials to search around for the next frame. The lane object is only updated if the newest polynomial found is within some reasonable margin of the previous polynomial; this way, unprecedented errors are discarded. Please see class def `Line.populate_line()` for commented code explaining the iterative process.

Discussion

Through completing this project and analyzing my results, I have several topics to discuss in terms of improvement:

- **First frame needs to be a straight lined section of lane** in order to be able to use the sliding window approach as an initial line detection method, because the sliding window approach's fundamental strategy is finding the column of the image that holds the most activated pixels. This limits the versatility of the pipeline, so that videos where the vehicle starts on a curved section of lane cannot be processed accurately.
- **Sliding window approach could be further refined** by using a second histogram within each window to re-center the next window instead of taking the mean.
- **In the final seconds of *project_video_output.mp4*, right line does not update and thus is too straight and the detected lane does not cover the full width of the lane.** The line identification begins to look wild in the end of the video, resulting in the averaged line (without any further updates) being displayed instead of the true line. I tried tuning the thresholds for acceptable differences between polynomials until this line was accurately identified, but was unable to find the sweet spot between too straight and too wobbly. The best wobbly identification is saved to *"project_video_output_wobbly.mp4".*
- **Pipeline seems to be inefficient.** It takes almost a full 5 minutes to process a 50 second video. I think I would need to investigate the object types and sizes that I use so that the pipeline uses less memory.
- **My pipeline does not work for the challenge videos.** It is clear that the *"harder_challenge_video.mp4"* is quite curvy, so my averaging and difference thresholding between frames would have to be more forgiving. As well, an identified object (the motorcycle driver), interrupted the lane. This is something my pipeline is not built for. Additional processing is required to detect objects entering the lane and ignore them when trying to identify the lane lines.