P2 Writeup: advanced lane finding

reference rubric is here: **https://review.udacity.com/#!/rubrics/1966/view**

The goal of this project is write a piece of code that finds road lanes from a video feed. The video feed is taken with a single camera installed in front of a car approximately at the centre of the vehicle.

Since a video is a sequence of images, the steps of this project are given per each image, as follows:

1. Camera calibration
2. Distortion correction
3. Apply a perspective transform to rectify binary image ("birds-eye view")
4. Use color transforms, gradients, etc., to create a thresholded binary image.
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

1. Camera Calibration

All images and videos in this project are taken using the same camera. So, the camera calibration process can be done only once, using a set of chessboard images (provided as support of the project) and the result of calibration process can be used to correct the distortion of all other images. More specifically, the camera calibration process yields the camera matrix, the distortion coefficients, the rotation vectors and the translation vectors. To calibrate the camera I used the cv2.findChessboardCorners() function to find the corners of all reference images, and then by calling the cv2.calibrateCamera() that takes the corners found in the chessboard images and the grid of corners as they should appear in an undistorted image. The figure below shows the result of the calibration and distortion correction processes applied to two of the provided images (one on the left and one on the right hand side of the image). The top row of the figure shows that the original images I chose for this report have radial and tangential distortions. The second row of the figure shows the same images after applying the correction distortion process using the open cv cv2.undistort() function that takes the camera matrix and the distortion coefficients as input. The images on the bottom of the figure show the result of the perspective transformation (birds-eye-view) applied to the undistorted images using the edges of the board as source and destination points. The function run_calibration_process() in the source code, triggers the entire calibration process.

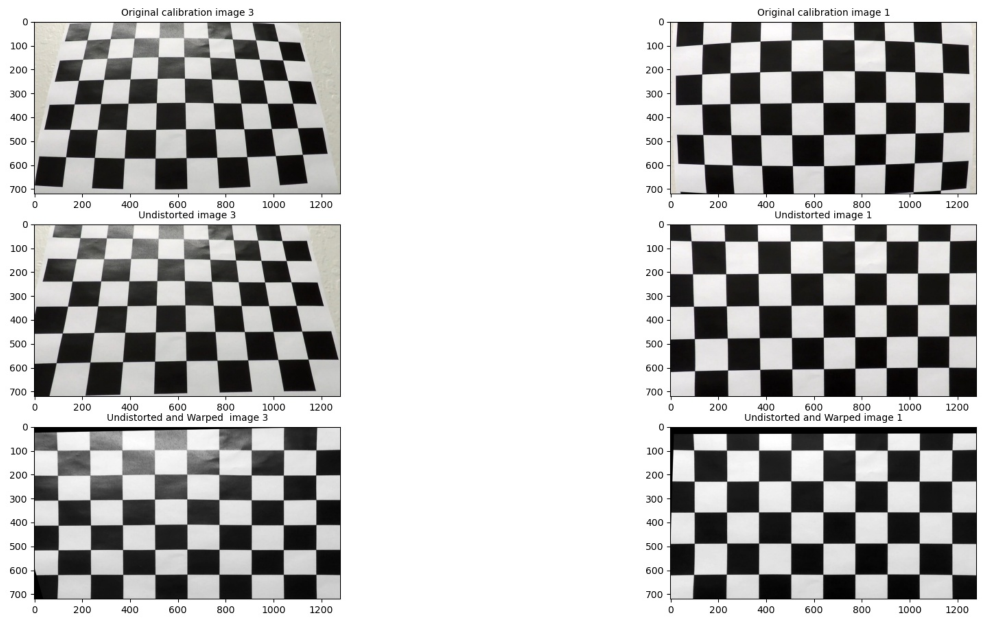
Original calibration image 3
Undistorted image 3
Undistorted and Warped image 3
Original calibration image 1
Undistorted image 1
Undistorted and Warped image 1

FIGURE 1: DISTORTION CORRECTION AND PERSPECTIVE TRANSFORM OF CHESSBOARD IMAGES TO SHOW THE RESULT OF CAMERA CALIBRATION.

2. Distortion correction

The distortion correction uses the camera matrix and the distortion coefficients to undistorted images. If a given image shows optical distortions, such as in the two top chessboard images above, the undistorted image is an image where all optical distortion are removed. With images of chessboards, it is easy to see the effects of distortion correction because the lines that seem rounded because of the radial distortion (top images) are now straight (middle images). The effects of a distortion correction might be less evident in the test images (see below) where the same distortion correction function is applied.

FIGURE 2: DISTORTION CORRECTION AND PERSPECTIVE TRANSFORM OF TWO OF THE TEST IMAGES TO SHOW THE RESULT OF CAMERA CALIBRATION.

3. Apply a perspective transform to rectify binary image ("birds-eye view")

The perspective transform process takes an image as input and warps it so that the resulting image seems seen from another perspective. In this project we focus on the top down or "birds-eye view" perspective because two road line in a birds-eye-view are parallel and therefore it is easier to to apply all line detection function on an image with parallel lines. In particular, for the purpose of this project it is convenient to focus the lane finding algorithms on a a very specific area within the original image because we know where the lane lines will be with respect to the camera view. The top layer of the picture below shows the two images with highlighted the trapezius that I have chosen as region of interest. The position of the edges of the trapezius was done “manually”; that is, I observed the effect of the choice of the trapezius edges in the original (undistorted) image space on the birds-eye view image (second row in the image below). My goal was to make sure that in the birds-eye view perspective the road lanes seem to be straight and parallel. I have added some artificial vertical lines to help my observations. After the manual inspection, I fixed the source and destination points of the perspective transformation and used them for the rest of the project. Note that the region of interest cuts a portion of the car hood.

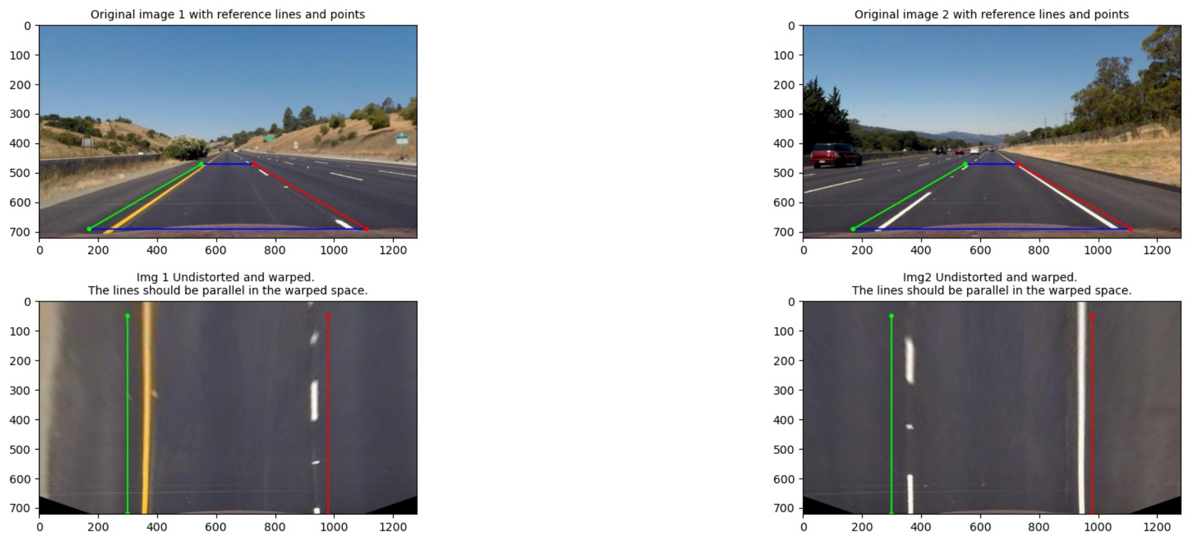
Original image 1 with reference lines and points
Original image 2 with reference lines and points
Img 1 Undistorted and warped.
The lines should be parallel in the warped space.
Img2 Undistorted and warped.
The lines should be parallel in the warped space.

FIGURE 3: PERSPECTIVE TRANSFORM OF TWO IMAGES WITH STRAIGHT LINES. THE BIRDS-EYE-VIEW SHOWS THAT THE ROAD LINES ARE PARALLEL

4. Use color transforms, gradients, etc., to create a thresholded binary image.

Since the project videos show many different scenes (shaded areas, change of pavement colors, lanes of different colors etc.) I decided to observe the individual effect of single color channels and gradients and then combine the best channels in a resulting binary image.
The figure below shows all the single channels I studied for one of the test images. The first row shows (from left to right)

1. the birds-eye-view of the test image
2. the binary image of the saturation channel only, filtered with some thresholds
3. the binary image of the red channel only, filtered with some thresholds

the second row (from left to right)

4. the binary image after applying a Sobel operator on the x axis, filtered with some thresholds
5. the binary image after applying a Sobel operator on both axes and taking their magnitude, filtered with some thresholds
6. the combination of all the previous filters as logical OR

The result seem to work well for this image, but the next image shows some drawbacks of this simple approach. The value of the thresholds of each channel are the result of some visual tests. Before applying the Sobel operator on the x axis I apply some Gaussian blur reduction too.

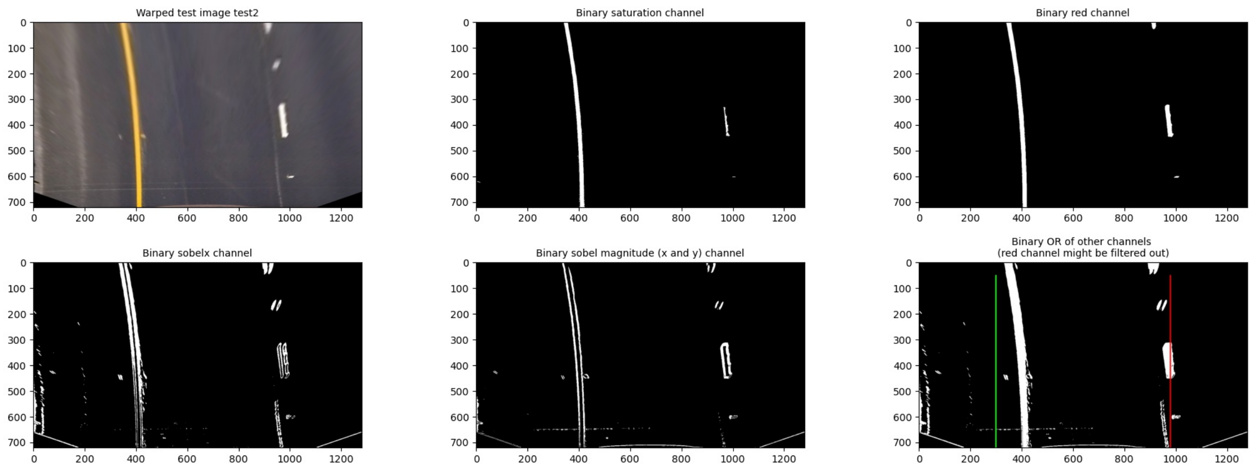

FIGURE 4: EFFECT OF 2 COLOR CHANNELS, THE SOBEL OPERATOR ON THE X AXIS AND THE GRADIENT MAGNITUDE. THE LAST PICTURE IS THE LOGICAL OR OF ALL CHANNELS

The figure below shows how for the test image 1, the red channel filter is too sensitive (top right image). The other filters seem to be still useful. To overcome the effects of the red channel, I have decided to set a threshold to the amount of activated pixels in the "red" image. I observed some images and set the threshold to "count_light_in_red_ch > 150000". The bottom right figure of the image below shows the result of the logical OR of all other filters without the red channel.

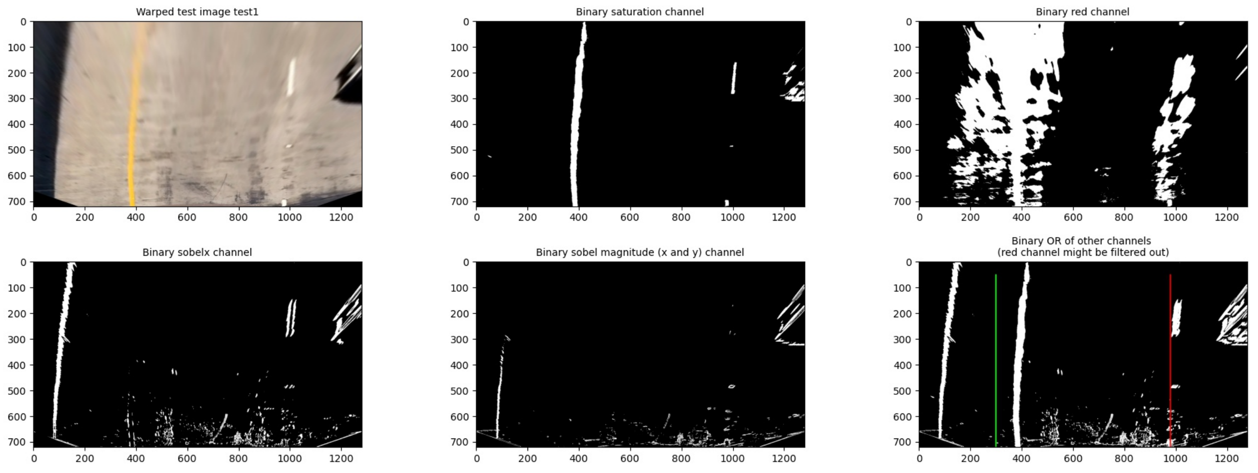

FIGURE 5: EFFECT OF 2 COLOR CHANNELS, THE SOBEL OPERATOR ON THE X AXIS AND THE GRADIENT MAGNITUDE. THE LAST PICTURE IS THE LOGICAL OR FILTERS OUT THE RED CHANNEL

7. Detect lane pixels and fit to find the lane boundary.

The lane detection pipeline (function called “process_frame()”) uses two different lane finding strategies, one based on the histogram and vertical sliding windows and one based on the search around prior poly line. My goal is to use the search around prior poly method as much as possible because it is less computational intensive than the histogram based approach.

The histogram and vertical sliding windows approach is implemented in the function called lane_search_with_windows() while the second approach, search around prior polynomial is implemented in the function called lane_search_around_poly(). Both functions return a dictionary with relevant line parameters such as the 2nd degree polynomial coefficients calculated with cv2.polyfit() and the list of x and y coordinates of each line.

At the beginning of the video I use the histogram approach to find the first few lines, every time some lines are found, they are saved into a buffer of lines until the buffer is full of lines. I used a ring buffer with size 5 (it seems to work well enough) to store the lines. When the buffer is full, I average all the lines in the buffer (left lines and right lines are treated separately). From the moment when the buffer is full, ideally I will only search for lines using the second method, unless there is some issue with the new lines.
If the detected lines with the prior search method are “good enough”, I store them in the buffer (that is, the latest detected lines replace the oldest lines in the buffer), I use them for the rest of the pipeline, and move to the next frame.

My definition of “good enough” is as follows:
- The lane with is between 3.3m and 4.2m
- The car offset is less than (in absolute value) 45cm
- The difference between the poly fit coefficient of the new line and the averaged poly fit coefficients in the buffer is less than 30 (which is a value I chose looking at some observations)

If the detected lines with the search around prior are not good enough, then, I clear the buffer and try the histogram approach once again. The pipeline continues to use the histogram approach until the buffer is full.

My current implementation of the pipeline works well (it is not perfect) only for the project video. Thanks to the smoothing process and the red filtering, I can discard the effects of shadows and the effect of changing color of driving surface. Improving the criteria for good line detection, I should be able to remove the few glitches that are still visible.

Nevertheless, the pipeline does not work reliably neither with the challenge video nor for the harder challenge video.

The challenge video is characterized by many black lines that overlap the road lanes for some time and then diverge from them. My current definition of good enough lines is not robust to discard such lines. Possible improvements could be checking the curvature radius of the new lines and discard them if they diverge too much (I tried to implement such system but I did not manage to build a complete solution). Another improvement could be to explore additional color and gradient filters to remove the black lines from the binary image.

The harder challenge video shows very windy curves and my current implementation of the pipeline performed very bad. First improvement I can think of is to modify the histogram detection approach to detect if one of the sliding windows “touches” the side of the image. This probably happens very often because of the very small curvature radius of the curves in the video. The shining light in front of the camera is also a challenge for the detection, therefore a smarter color and gradients filter should probably be used. The motorbike cutting the line covers the left lane for some time, a clever algorithm should probably reuse old lines in this part of the video.

8. Determine the curvature of the lane and vehicle position with respect to center.

The drawing below shows how I calculated the car offset (function car_lanes_offset())with respect to the lane middle (the length of the segment D in the drawing). A and B are the x coordinate of the left and right detected lines. The drawing shows that A and B are calculated for different y values, but I used the same value of y, that is ymax. The car offset to the centre of the lanes is calculated as D= [(A+B)/2+A] - xmax/2 that is, D=lane middle-Image middle . If D>0 then the is on the left of the lane center.

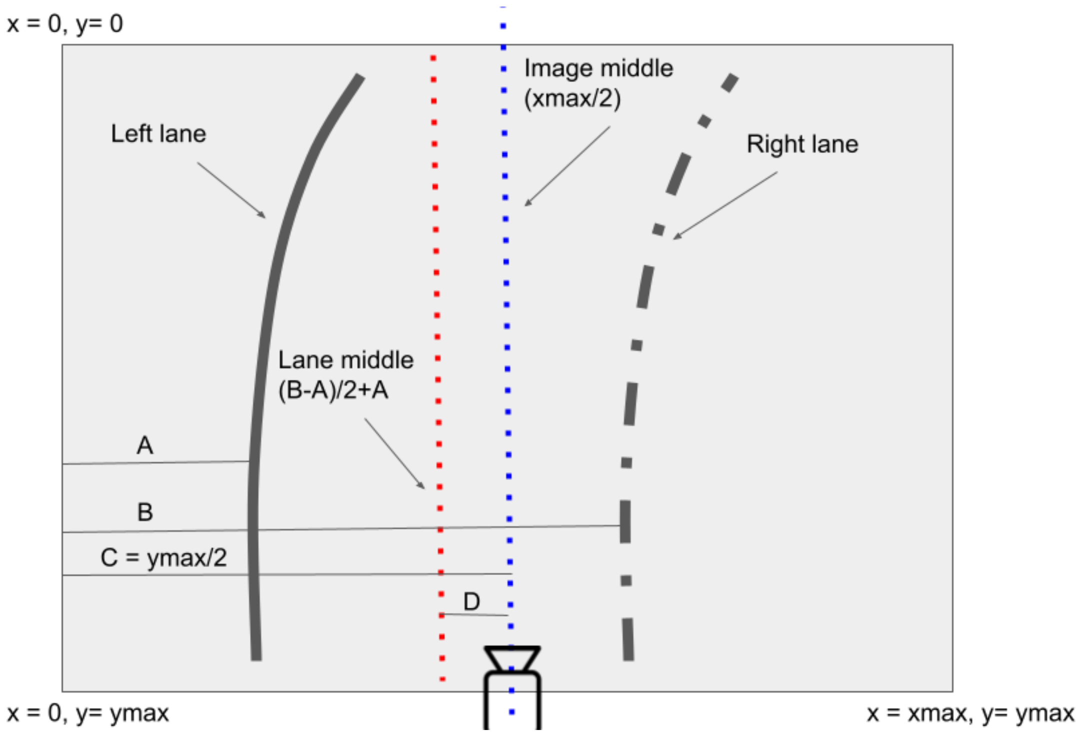

FIGURE 6: DRAWING THAT EXPLAINS HOW TO MEASURE THE CAR OFFSET FROM THE CENTRE OF THE LANES

Calculating the curvature and Conversion from pixels to meters

The curvature radius of each line is calculated using the formula below,

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

where A and B are the polynomial coefficient s of y2 and y, respectively. The evaluation of the curvature radius already converted into meters is implemented in the function measure_curvature_in_meters().

Following the same thinking as described in the class, I converted the curvature radius and the car offset into meters. The conversion rate I used is based on the visual observation of the different images and on the fact that the legal lane width in the us is 3.7m. Since I observed that in the pixel space two lanes (in the warped/birds-eye-view space) have a distance of about 600 pixels, the conversion rate for the x axis is 3.7/600. For the y axis I based my observation on the fact that dashed lines have a nominal distance of 3m. So the conversion rate I used is 15/720 where 720 is the entire height of the image.

9. Warp the detected lane boundaries back onto the original image.

Figure 7 below shows one frame of the project video that was edited using the the entire lane finding pipeline. The image shows the detected lane boundaries and the corresponding "driving space" between them. The lane detection and driving space coloring was done in the birds-eye-view space and warped back into the image space using the inverse perspective transformation matrix.

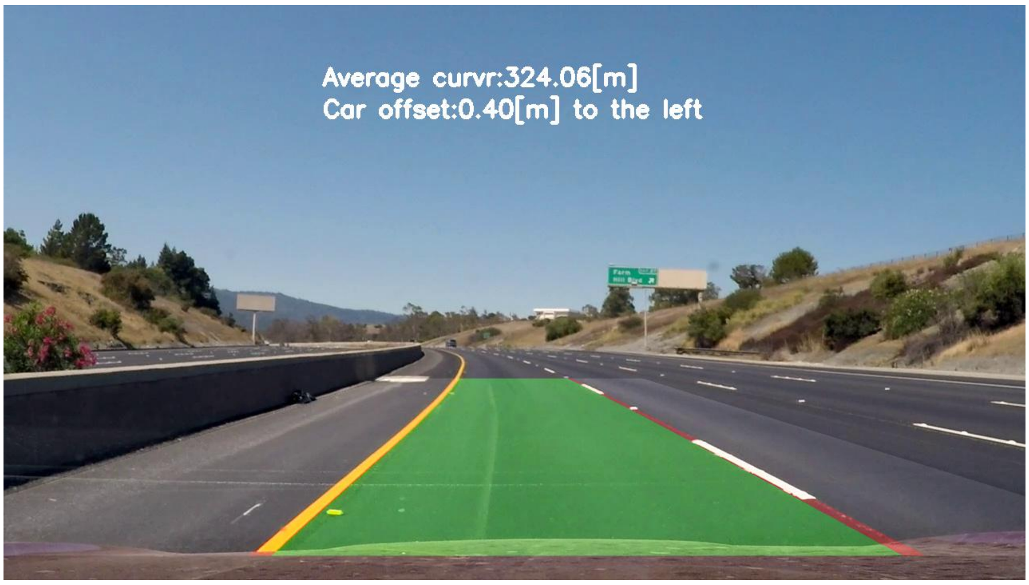

FIGURE 7: EXAMPLE OF DETECTED LANE BOUNDARIES WARPED BACK ONTO THE ORIGINAL IMAGE

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

Figure 8 shows the pipeline applied to the first frame of the project video. The original picture is shown on the top left corner. On the top right corner you can see the result of the pipeline including the curvature radius (in the expected range) and the car offset with respect to the lane center. The figures on the bottom layer show the birds-eye-view of the region of interest and the detected polynomial lines with the vertical sliding windows method.

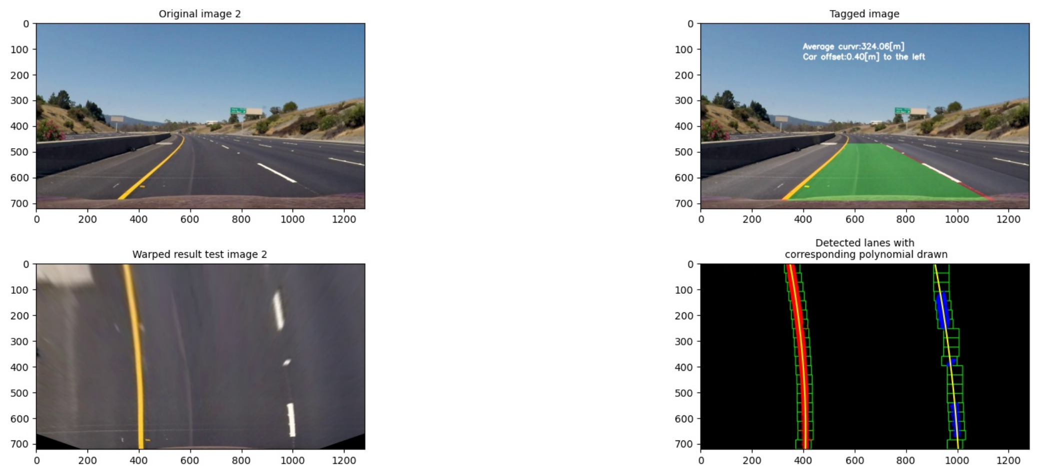

FIGURE 8: EXAMPLE OF DETECTED LANE BOUNDARIES WARPED BACK ONTO THE ORIGINAL IMAGE

The processed project video result of the pipeline is provided as part of this submission and is called “project_video_processed.mp4”