Advanced Lane Finding

Project Submission Writeup Report

For the Udacity Self-Driving Car Program – Term 1

Submitted by:

Stewart Teaze

July 14, 2017

Project Goals

The goals / steps of this project are:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the 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.

Rubric Points

The following sections of this document relate to each of the project's rubric points (project requirements), and provide details of how each requirement was addressed in the project implementation.

---

1. Submitted Files

1. writeup.pdf - This document/file is the writeup itself.
2. P4.ipynb – Jupyter Notebook with Python code segments, capturing each development step of the evolution of the computer vision processing and curve-fitting steps, contributing to (and including), the full implementation of the lane finding image processing pipeline.
3. test_images_output – Directory containing image output files, containing one lane-line-demarked/overlayed image, corresponding to the test images found in the provided directory of test files(test_images), produced/output by the code in P4.ipynb.
4. test_video_output – Directory containing lane-line-demarked/overlayed video output file (project_video_out), produced/output by the code in P4.ipynb, corresponding to the provided test video "project_video".

All of the files/directories listed above, can be found at:
https://github.com/blunderbuss9/CarND-P4-AdvLL-StewartTeaze

2. Camera Calibration

a. Initialize Camera Calibration Parameters, using provided set of Camera Calibration Images

The vehicle-mounted driver-view video camera used for creating the test images and videos for this project, as all cameras, has distortion qualities that need to be accounted for, when processing a video stream of images from that camera (such as the lane line detection processing performed by the code submitted for this project). This accounting is accomplished by calibrating the camera to allow for elimination of the distortions.

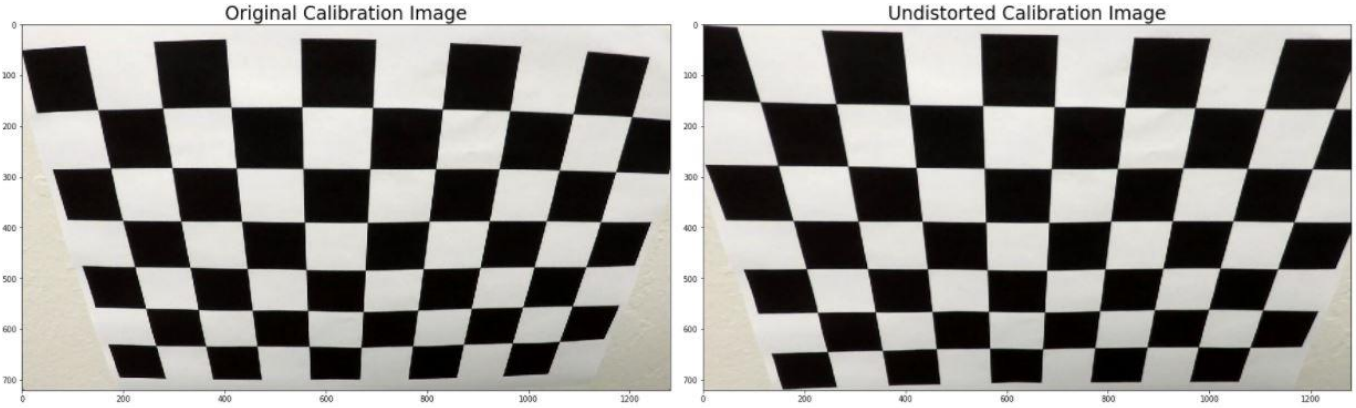
As part of the initialization process, found in the 3rd Python code cell in the P4.ipynb Jupyter Notebook submitted with this project, the objpoints, and imgpoints Python list structures, containing the camera calibration parameters, are built up by calling the OpenCV cv2.findChessboardCorners() function, once for each of the chessboard camera calibration images. Specifically, for each chessboard calibration image that is processable (some aren't, due to orientation or size of the chessboard in the image, or other factors), a set of "image points" (detected corner coordinates in each distorted test image) and "object points" (x, y, & z coordinates of the corners of the actual chessboard used as the source object in all the calibration images) is added to the imgpoints and objpoints list structures, for latest use by the cal_undistort() utility function, described next.

b. Define Camera Undistortion Function cal_undistort()

A utility function cal_undistort(), defined in the 2nd code cell "Utility Functions", is used to produce undistorted images, which it will return when invoked and passed a pre-distorted image, along with the set of objpoints and imgpoints camera calibration parameters Python list structures (described in the previous paragraph). cal_undist() itself calls cv2.cvtColor(), cv2.calibrateCamera(), and cv2.undistort().

c. Undistorted Calibration Image

Following is a side-by-side comparison, generated in the 4th code cell, of one of the original pre-undistorted calibration images from the set of provided calibration images, along with the undistorted version produced by cal_undistort().



d. Undistorted Test Image

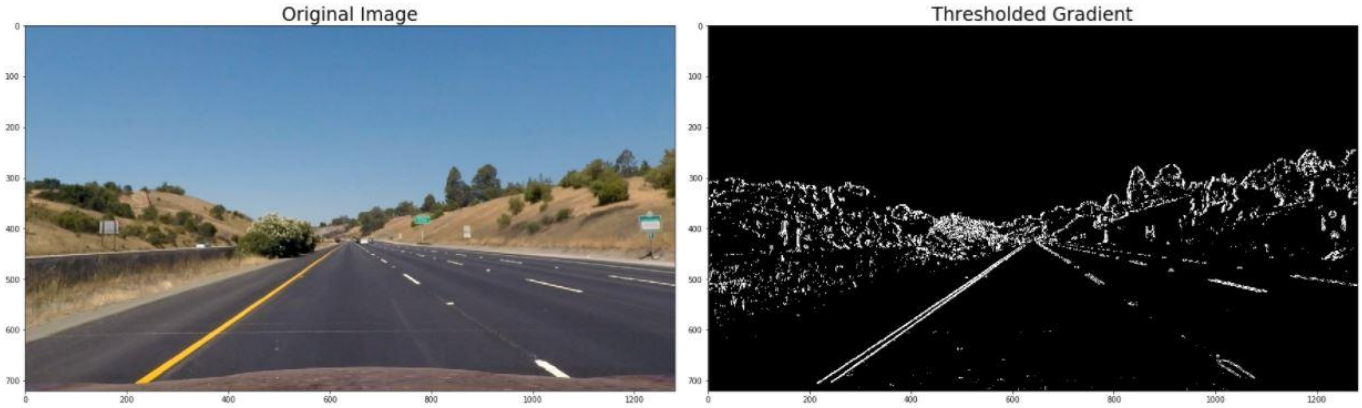
Following is a side-by-side comparison, generated in the 5th code cell of one of the original pre-undistorted test frame images, from the set of provided test images, along with the undistorted version produced by the cal_undistort() function.

3. Developing/Assembling Components of the Lane-Finding Video Processing Pipeline

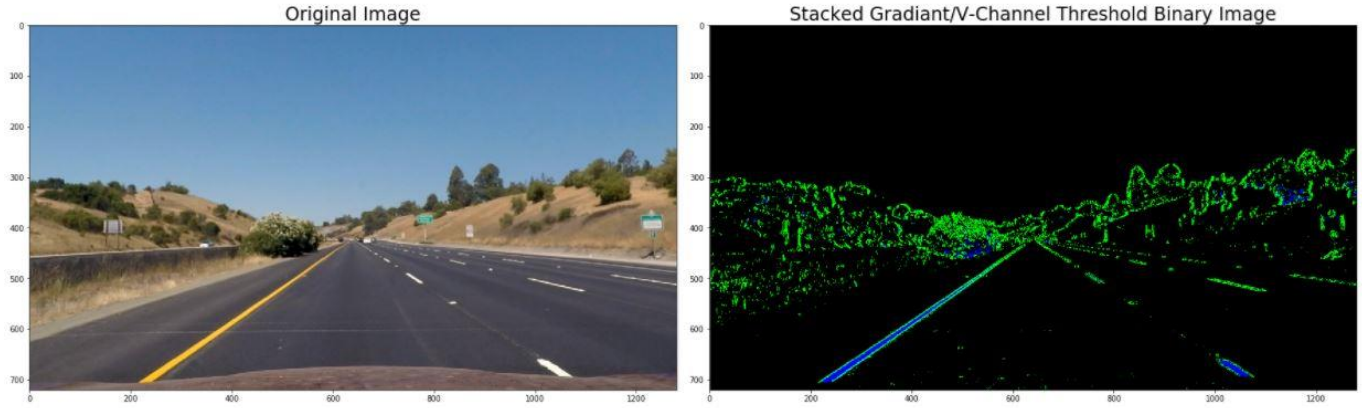
a. Gradient Thresholding and V-Color Channel Thresholding

By applying a combination of Gradiant Thresholding (a technique involving using the Sobel matrix multiplication operation to obtain the change in derivative or "gradient" – indicating line edges in the x or y direction), and V-Channel Color Thresholding (by converting an image to HSV color space, and separating out the V channel), lane lines are very easy to isolate in the original image. The stages of this technique will now be demonstrated by a series of three sets of side-by-side image comparisons…

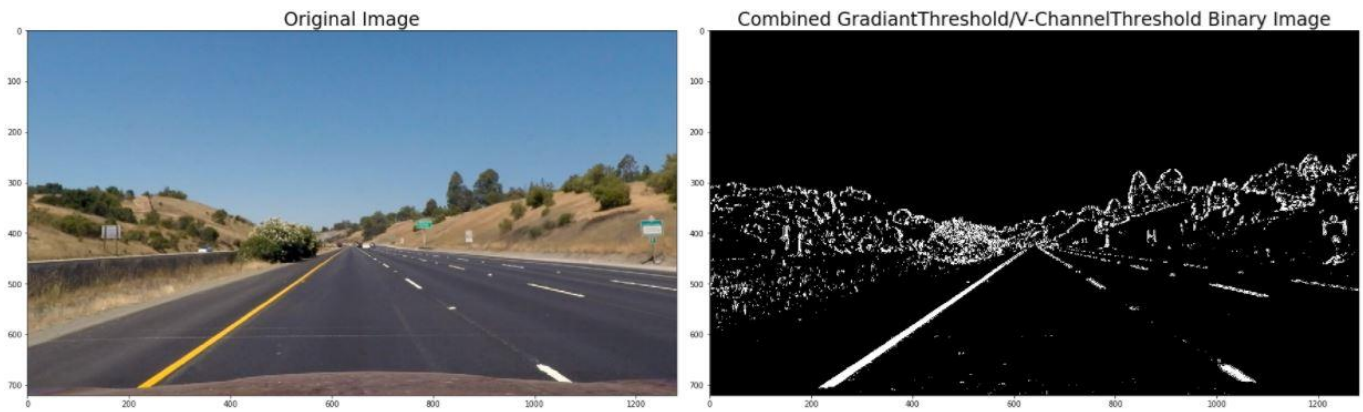
First, the binary image result of applying only the gradient threshold operation, implemented in the Jupyter Notebook in the 6th code cell, by using the function abs_sobel_thresh() implemented in the Utility Function code section:



Second, the binary image result of applying the gradient threshold operation (depicted in green), and stacking the V-Channel Color threshold component (depicted in blue) - implemented in the Jupyter Notebook in the 7th code cell, by invoking the function GradThresh_ColThresh_Stacked () implemented in the Utility Function code section:



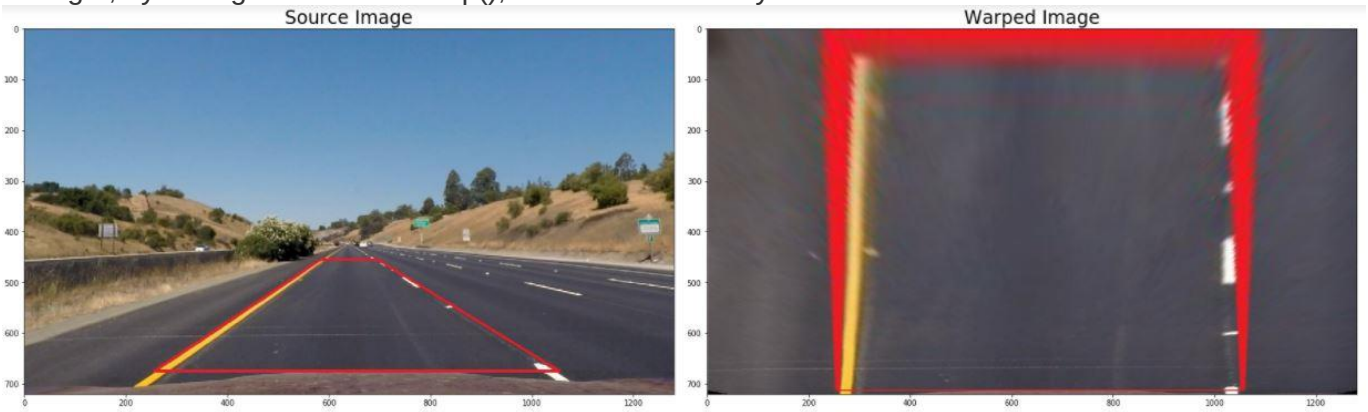
And then the final binary image result of applying the gradient threshold operation, and combining the results with the V-Channel Color threshold component - implemented in the Jupyter Notebook in the 8th code cell, by invoking the function GradThresh_ColThresh_Combined (), also implemented in the Utility Function code section:



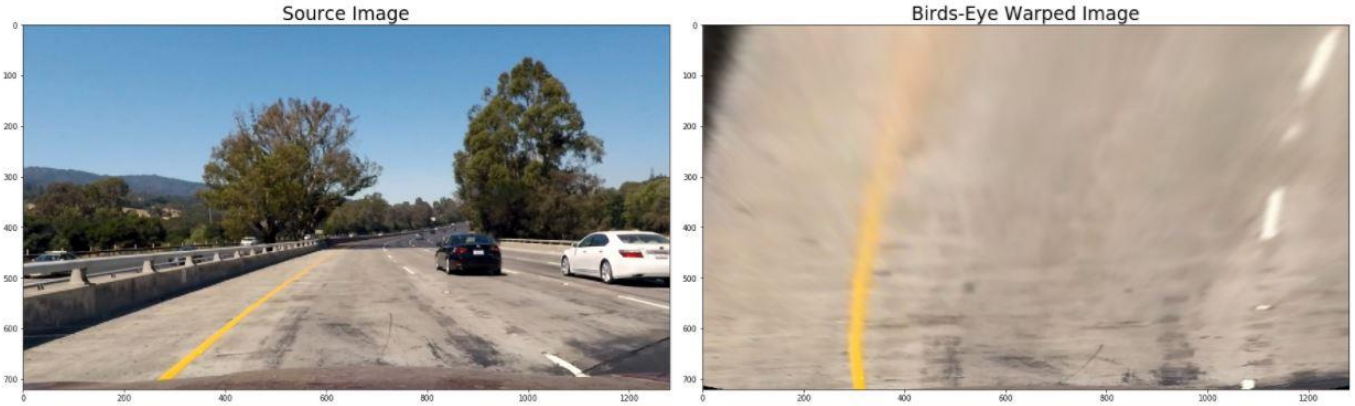
b. Apply a perspective transform to rectify binary image ("birds-eye view").

To more easily detect the lane lines in each video frame, a perspective transform will be applied to each undistorted frame image, to transform the image into a “birds-eye view” of the area directly in front of the vehicle, out to a distance where the lane lines will not become too distorted by the transform. The stages of this technique will now be demonstrated by a series of three sets of side-by-side image comparisons, with explanation of the associated code used for generating the images…

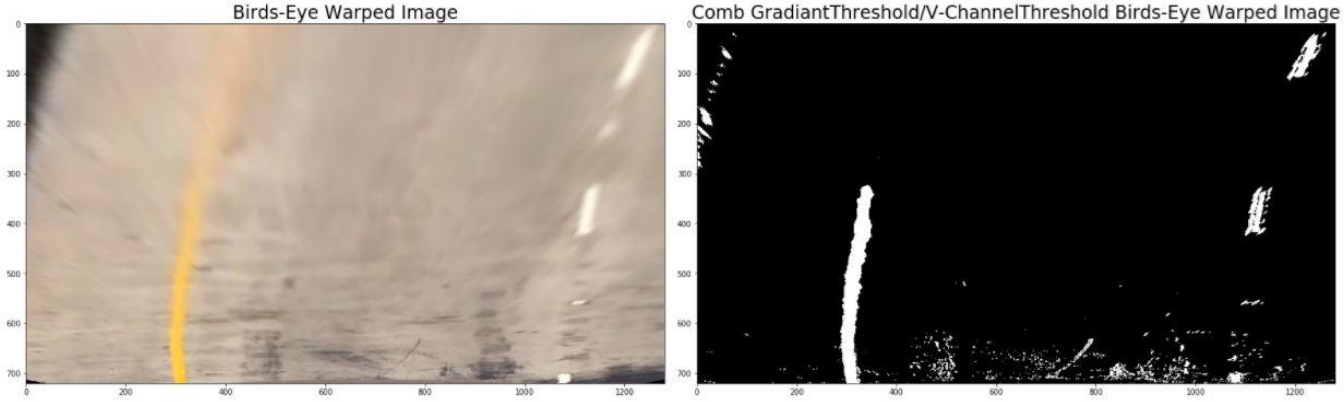
First, Microsoft Paint was used to draw a red trapezoid on top of a straight lane sample image (seen below on the left), the “source” x,y coordinates of the corners of the trapezoid were noted, and then used to in the 9th code cell to “warp” the source image into the “destination” birds-eye warped image shown below on the right, by calling the function warp(), defined in the Utility Functions code cell.



Secondly, the code in the 10th code cell was used to “warp” a sample source test image, taken while the vehicle was in a moderate curve, into the “destination” birds-eye warped image shown below on the right, again by calling the function warp(), defined in the Utility Functions code cell.



Lastly, the code in the 11th code cell was used to call the GradThresh_ColThresh_Combined() utility function, to take that color birds-eye image (shown below on the left), and perform a Combined Gradient Threshold and V-Channel Color Threshold operation, to create the image shown below on the right:



c. Identifying Lane Lines and Determining Curvature of the Lane

To start identification of lane lanes in images that have been put into the Combined Gradient Threshold / V Color Channel Threshold Birds-Eye Warped Image format, histograms are taken of the lower half of each image processed. In Figure 1 below, is a graphic showing the histogram for the lower half of the Combined Gradient Threshold / V Color Channel Threshold Birds-Eye Warped Image, that was shown at the end of the last section of this document. This graphic was generated in the Jupyter Notebook's 12th code cell.

NOTE: Created in the 11th code cell, and used in most of the subsequent Python code cells in this project's Jupyter Notebook, the important Combined Gradient Threshold / V Color Channel Threshold Birds-Eye Warped images, that provide the core processed image format for use in the lane identification process in the code, are referred to by the variable name/shortened acronym **CGTCTBEWimg**.

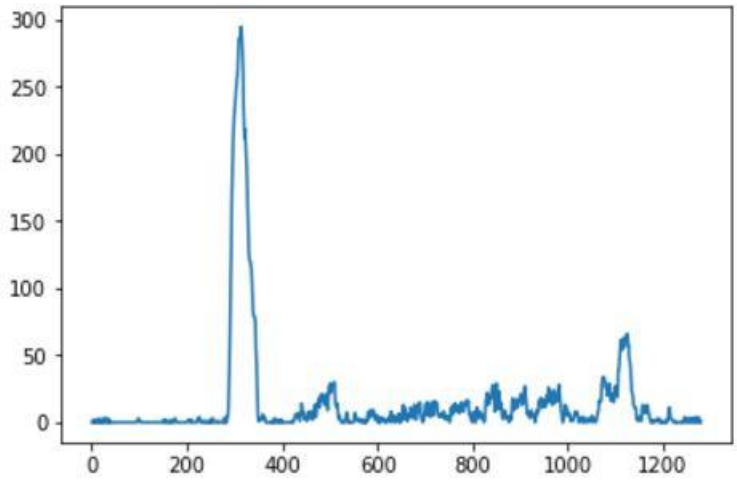
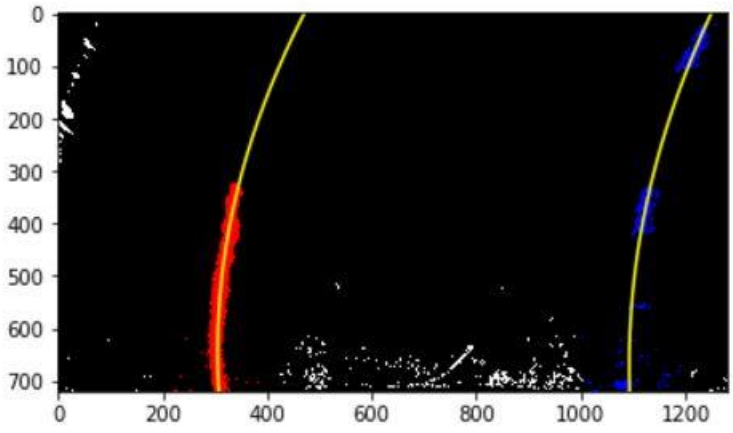


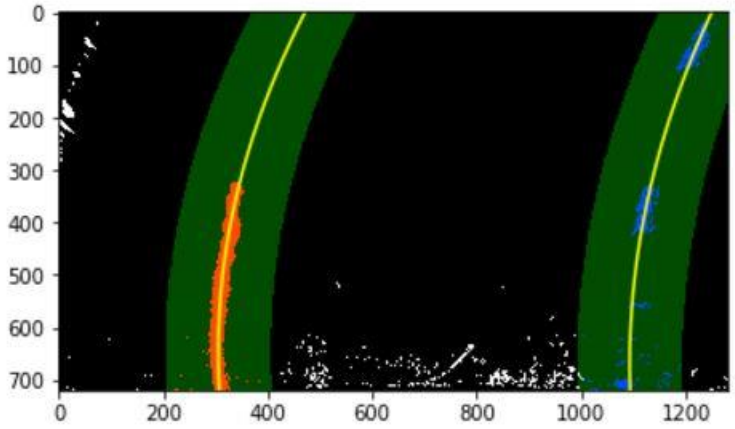
Figure 1

In the 13th code cell, the left and right peaks of the histogram are used as starting points, at the bottom of the image, to begin a series of 9 pairs (left and right) of sequentially stacked window areas to search for the lane pixels, with the individual search window areas adjusted left or right, based on the results of the pixels found in the previous search windows (left and right, correspondingly). To start identification of lane lanes in images that have been put into the Combined Gradient Threshold / V Color Channel Threshold Birds-Eye Warped Image format, histograms are taken of the lower half of each image processed.

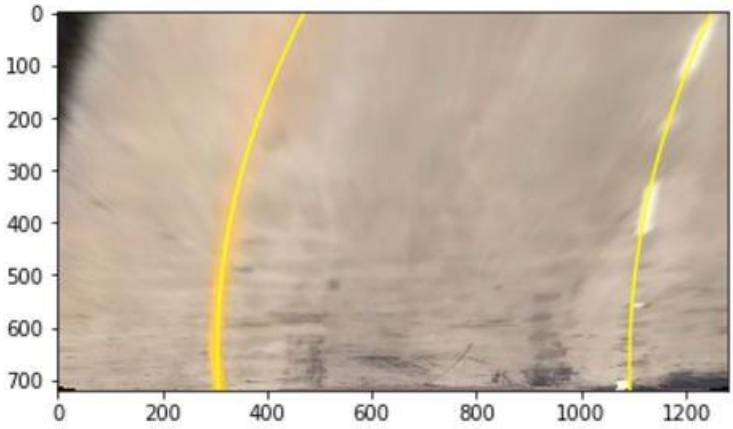
In the 14th code cell, the left and right pixels are colored (red for left and blue for right) and are shown in the image below, along with the curve-fitted polynomials determined for these pairs of lane pixels (shown in yellow).



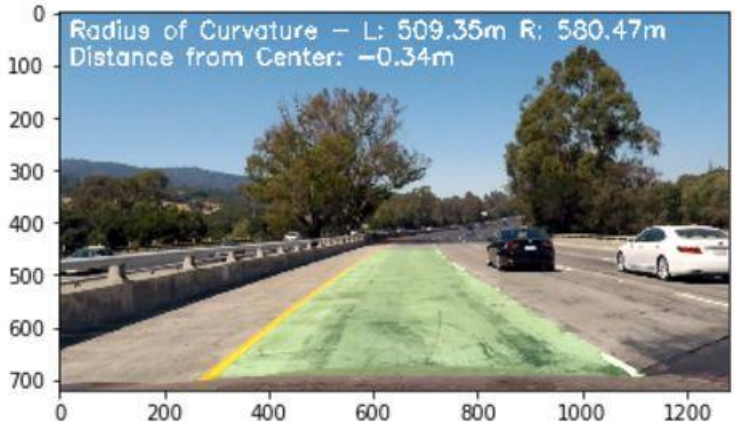
In the 15th code cell, the stacked window search areas are added and depicted in green:



In the 17th code cell, the curve-fitted polynomials for the left and right lanes, is shown overlayed on the true color bird's eye warped version of the input image, as shown here:



In the 18th code cell, the lane lines and "lane extents" (shown in light-green shade) are de-warped back onto the original image space and overlayed onto the original image, along with text showing the current Radius of Curvature (of the left and right lane lines) and Distance From Center (of the vehicle in the lane) values, as shown here:



d. Discussion

Some of the more challenging parts of this project were related to finding and fine-tuning a good set of source and destination point coordinate parameters in the Birds-Eye Warp process. However, once a good set of parameters was found, the lane finding via polynomial fitting process worked amazingly well – better than I would have expected having first seen the images.

Areas for improvement mainly relate to speed optimizations to allow the process to work better in real-time. If full-quality image frames were not required to be written out during the real-time process, this would greatly improve the real-time performance… perhaps only saving lower-quality and/or compressed images at run-time, would help a great deal to improve processing speed, as disk I/O is always one of the most costly processes in image-related computing.