# Advanced Lane Finding Project

## Introduction:

The goals / steps of this project are the following:

- 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.

## Camera Calibration

The code for the camera calibration is contained in a *Camera class* in the scripts *camera.py*. (in folder *scripts*)

The calibration is done during the initialization of the class,  through the use of 2D chess board images taken by the camera (& provided with the  assignment) and calculating the distortion in those images with respect to a 3D static representaion of chessboards in a 2D plane.  The specific steps are:

- Setting up a series of static 3D coordinates for each of the expected corners in the chess board images in a perfect undistorted 2D plane. In the assignment, the chess board used in the images was based on a 9x6 number of corners between the black & white squares per 2D image. The 3D coordinates initialized the 3rd coordinate to zero (0), as it was to represent perfect corners  in a 2D space. Data structure *objp* was used to hold these coordinates.

- Reading in a series of chess board images that were provided with the assignment and converting these images to a gray scale. Care must be done at this step to do the colour to gray scale conversion based on the colour scale set when reading the initial images.
- Finding all the actual 2D coordinate locations (*corners*) of each of the chessboard corners in the raw chessboard images using the open cv function *cv2.findChessboardCorners & i*n providing the expected number of corners in the images (9x6 for our case). Note: Only the images in which all 9x6 expected corners were found were used later in the calibration process. An example successfully finding all the required chess board corners in one of the images is in illustration 1.
- Calbrating the camera using the open cv function *cv2.calibrateCamera* , and using all the successfully found 2D coordinate locations of the corners (*corners*) found in the previous step on the raw images, and the corresponding 3D->2D coordinate locations (*objp*) for those points on a chess board.
- This camera calibration produces 2 important data – a camera matrix (*mtx*) & distortion co-efficients (*dist*) that is saved in a pickle file & are to be used later to undistort images from the camera, through the use of *cv2.undistort* function. An example of undistorting an image is shown just below.
- Two additional methods with the camera class is provided:
    - An undistort method (*undistort*) to be used to undistort images for testing purposes.
    - A test undistort (*test_undistort*) method that displays the raw image and the undistorted images produced with the class.
    -

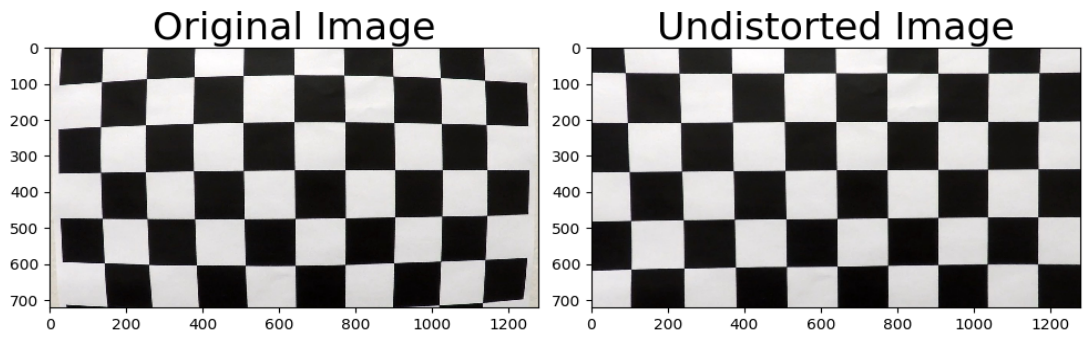
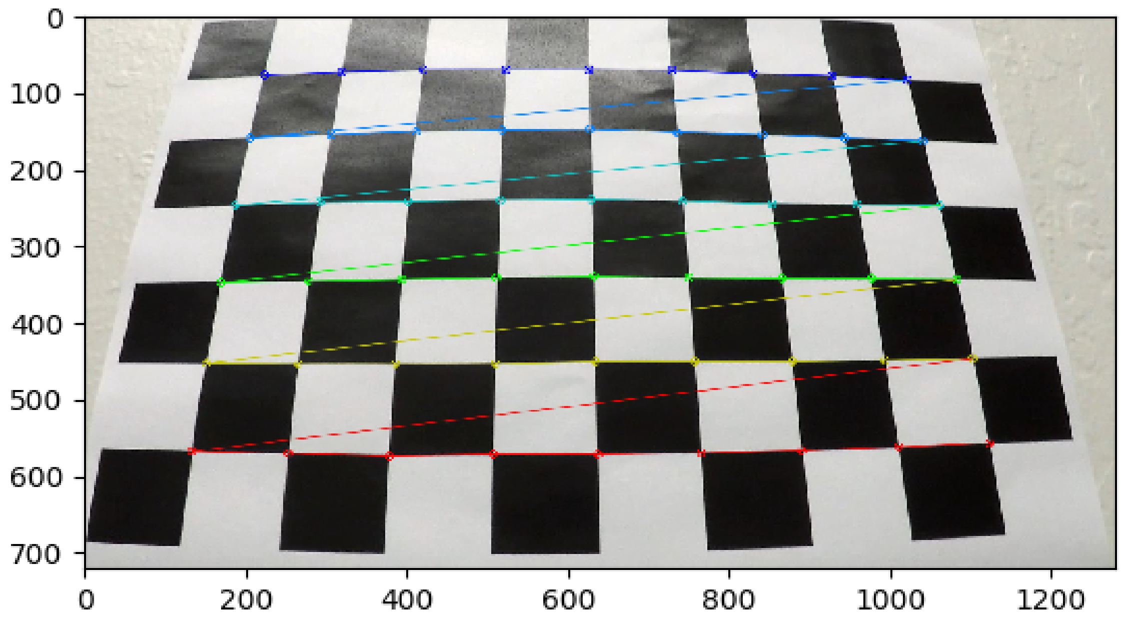


*Illustration 1: Example of Successfully finding chess board corners*



## Pipeline (Single Images):

A pipeline function was created to perform each of the required transformations on each image. It was tested on the test images provided with the assignment and the output of this testing is shown in the folders *output_images* & *test_images*.

This pipeline performs a series of steps as follows:

**Step 1**: Undistorts the image using the camera matrix (*mtx*) & distortion co-efficients (*dist*) determined earlier. An example of this is shown next as part of step 2.

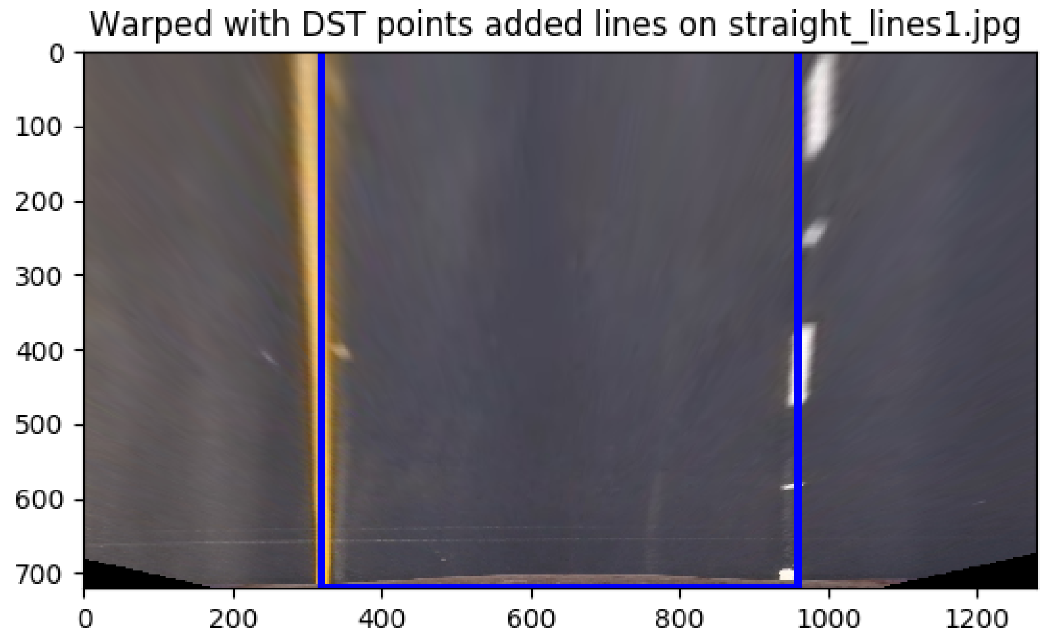
**Step 2 and 3**: Transforms or warps the image perspective to a "top down view".

In order to transform the image, source and destination points were created based on the size of the image, but were more or less hardcoded as follows, as is shown in the python script file "***pipeline***" code at scripts/main.py around lines 64-74.

```
img_size = (img.shape[1], img.shape[0])
src = np.float32(
         [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
         [((img_size[0] / 6) - 10), img_size[1]],
         [(img_size[0] * 5 / 6) + 60, img_size[1]],
         [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
         [[(img_size[0] / 4), 0],
         [(img_size[0] / 4), img_size[1]],
         [(img_size[0] * 3 / 4), img_size[1]],
         [(img_size[0] * 3 / 4), 0]])
```

A test image that was provided with the assignment, from t*est_images/straight_lines1.jpg* was used to manually select & validate these source and destination points by first undistoring the image, and drawing the lines to connect the source points (shown) then performing a perspective transform, and also drawing lines to connect the destination points on the  warped image.

As can be seen from the regular perspective image, the source points form a nice boundary of the perspective we are interested in. The destination points in the warped image,  form almost straight lines, which is what is desired.



Using these source and destination points and cv2 function *cv2.getPerspectiveTransform*, two transformation functions M and Minv are created  that are then used to transform or warp all the images to top-down perspectives, using M, and from top-down back to regular using Minv.  This is shown in lines 75-76 in procedure *pipeline()* in *scripts/main/py*.

The undistorted, then warped (to top down perspective) images of all the  test images can be seen in folder  ***test_images/trackedWarped.***

**Step 4:** Perform image analysis using various types of thresholding to identify important features in the image. In this case, it specifically performs “colour thresholding” to pick out white & yellow based sections of the "top down" view.

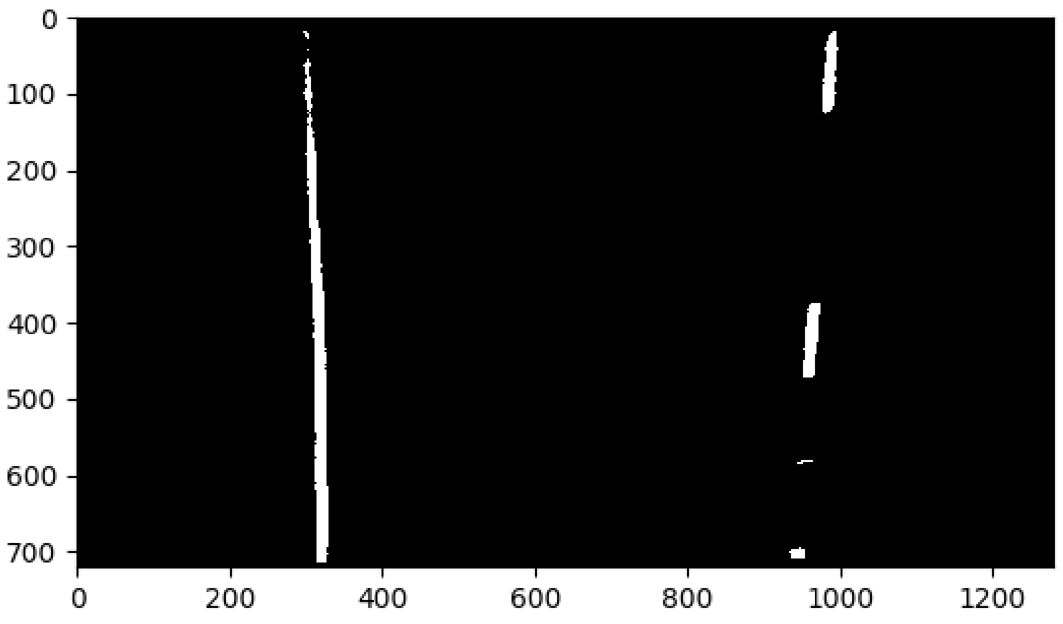
Much (very much!!!) experimentation was done on various colour and other types of thresholding applied both before or  after performing the perspective transform. In the end, the best & simplest results were obtained when the colour image was first transformed to a “top down” view, and colour thresholding was applied to this warped/transformed colour image to detect pixels in the white & yellow zone.

The code for the colour thresholding that was finally used is shown *scripts/threshold.py* lines 120-164 in the procedure *colour_select.*

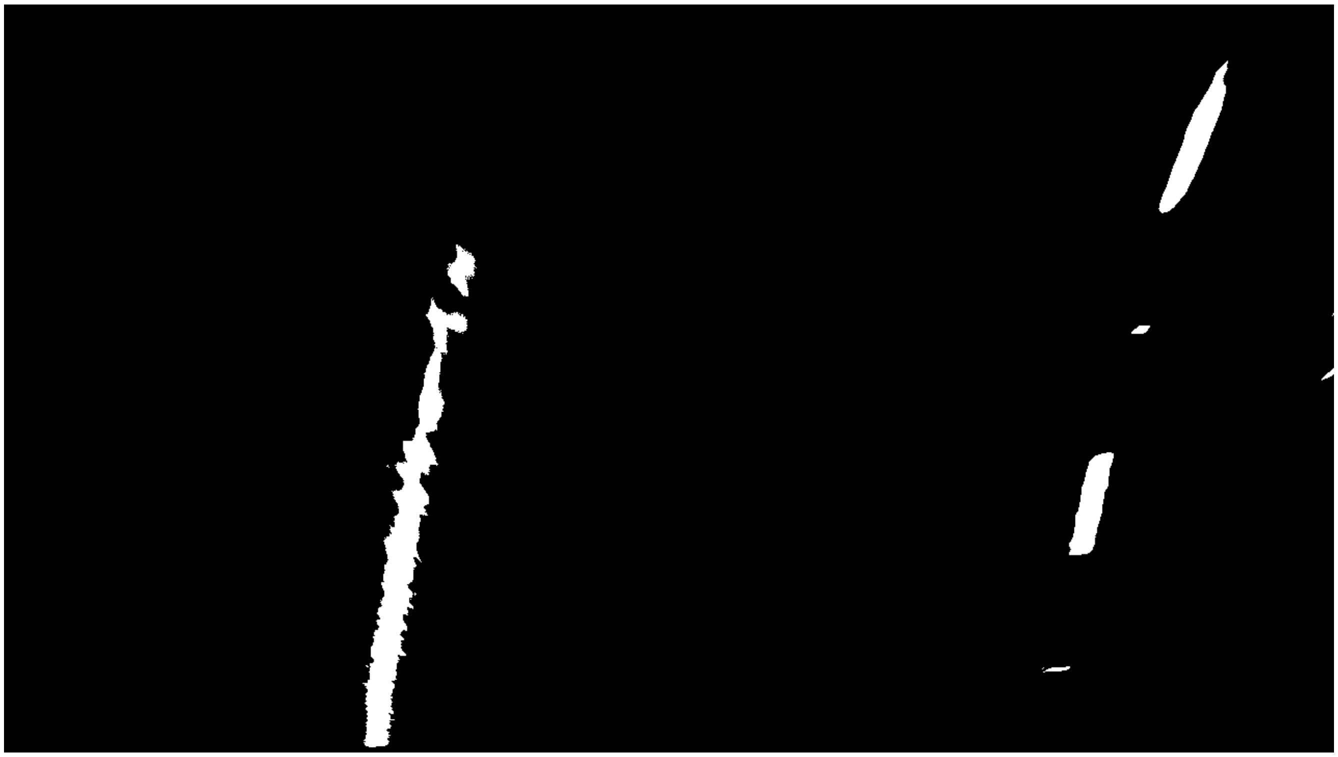
As seen in that procedure:

- All channels in the RGB image were used with specific thresholds to yield the best selection of pixels with yellow & white.
- All RGB channels in the image converted to HSV & specific thresholds applied to select only yellow & white pizels.
- All RGB channels in the image converted to HSL to select additional “white” pixels.
- All of these selected pixels in the  transformed image were then combined to form one binary image, for use in the  next step.

For the result of this step on the colour test image above that was previously undistorted & warped/transformed   (from original t*est_images/straight_lines1.jpg),* see below.



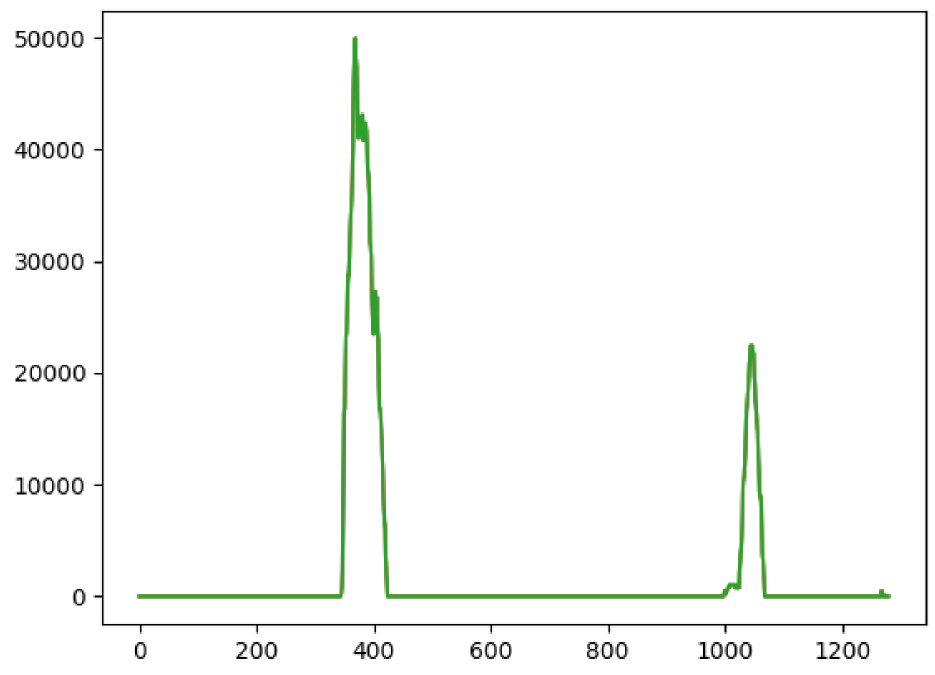
For the results of this step on all the test_images (test1.jpg – test6.jpg) , see folder ***test_images/trackedColourThreshed4-BinaryImages***.  But for the last of these test images, below see the undistorted, warped, & colour thresholded image for test6.jpg.

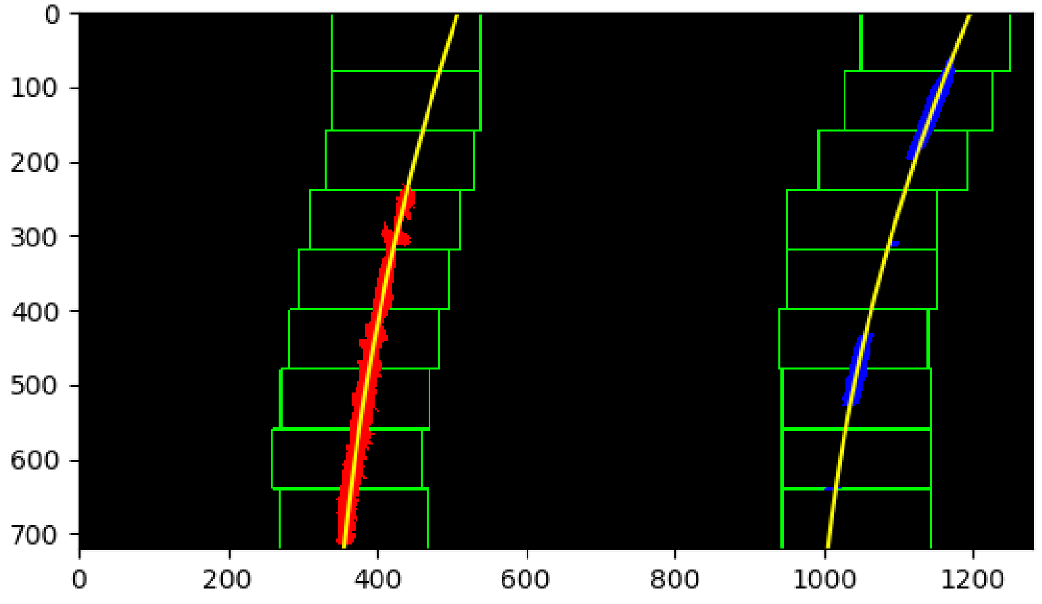


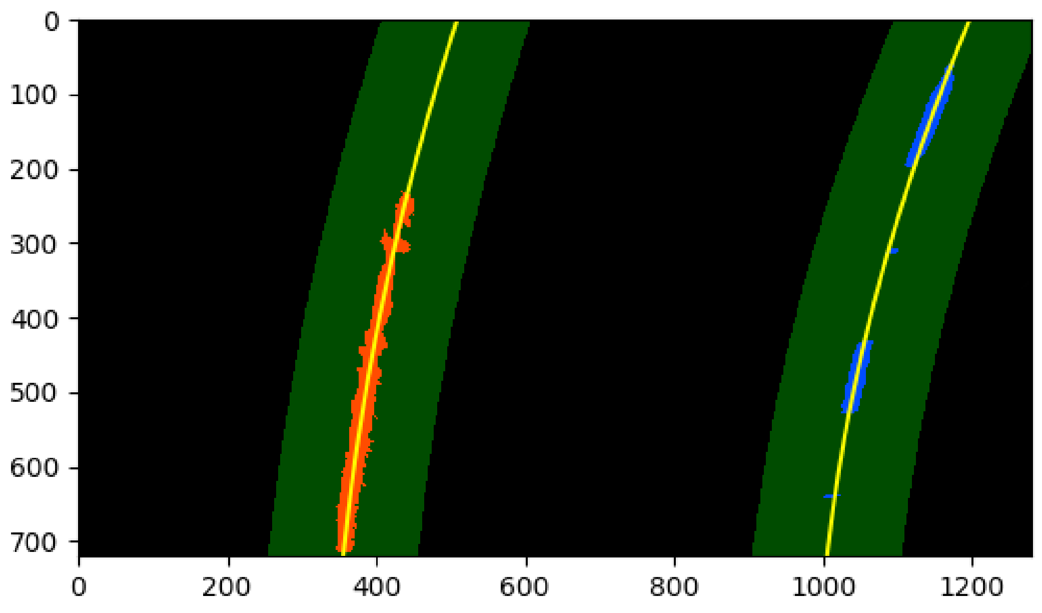
**Step 5:** Identifies the Lane Lines in the top down thresholded image. This is accomplished by using  a histogram & a sliding window algorithms to detect the two lane lanes, left & right, in which the car taking the images is traveling.

In this step a histogram was used on the undistorted, warped & colour thresholded to identify the most likely places in which the  lane lines would be. This histogram was then used with a sliding window algorithm, using 9 windows, to identify the lane line locations using a “blind search “ type approach. Then,  a “margin search” type algorithm was also used to identify where the lane lines would be expected given one knew where they were in the previous frame. The code for this is shown in the file in *scripts/lane.py*. Most of this code is taken verbatim from the Udacity classroom, with some modifications that are identified in teh code itself.

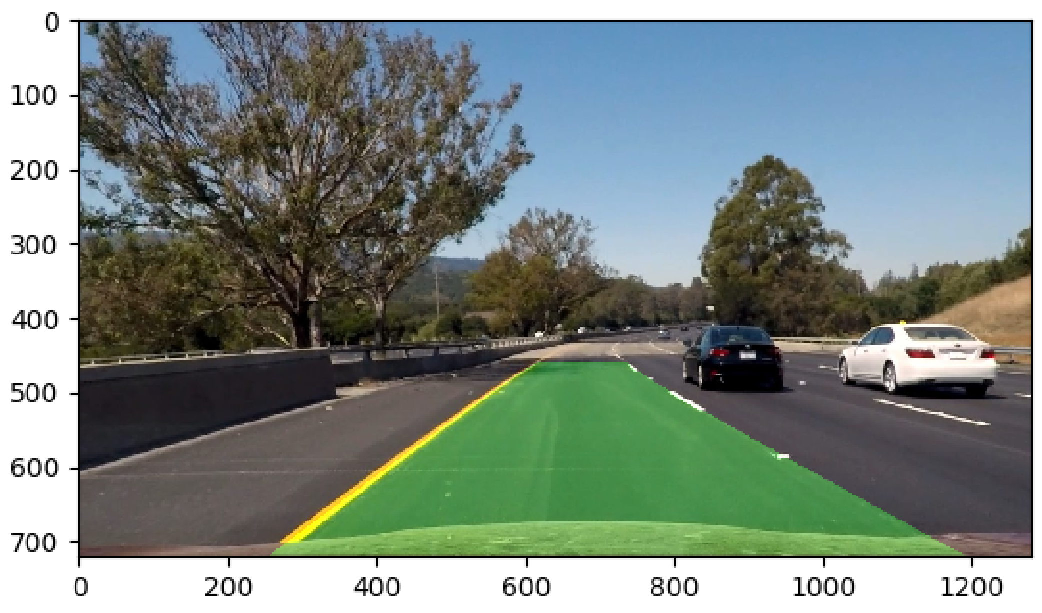
 An image of the histogram of the above colour thresholded (based on test6.jpg) is shown below, followed by a visualization of what was produced by the  blind search sliding window algorithm, and the other algorithm, when the approximate location of the lanes are known.


50000
40000
30000
20000
10000
0
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



**Step 6:** Draws the lane lines onto the image in regular perspective. Once the lane lines were identified earlier using either/both of the algorithms above (in step 5), then the lanes are drawn on the image in a solid green colour using the procedure *get_drawn_lanes()* in *scripts/lane.py* (lines 164-188) , and the image was warped back to regular perspective, using the inverse perspective transform that was calculated earlier from the source and destination points. An example of the results of this with test image test6.jpg is shown below.



Examples of lane lines drawn on the all the sample test images can be seen in folder ***test_images/ trackedLaneFilledGreen.***

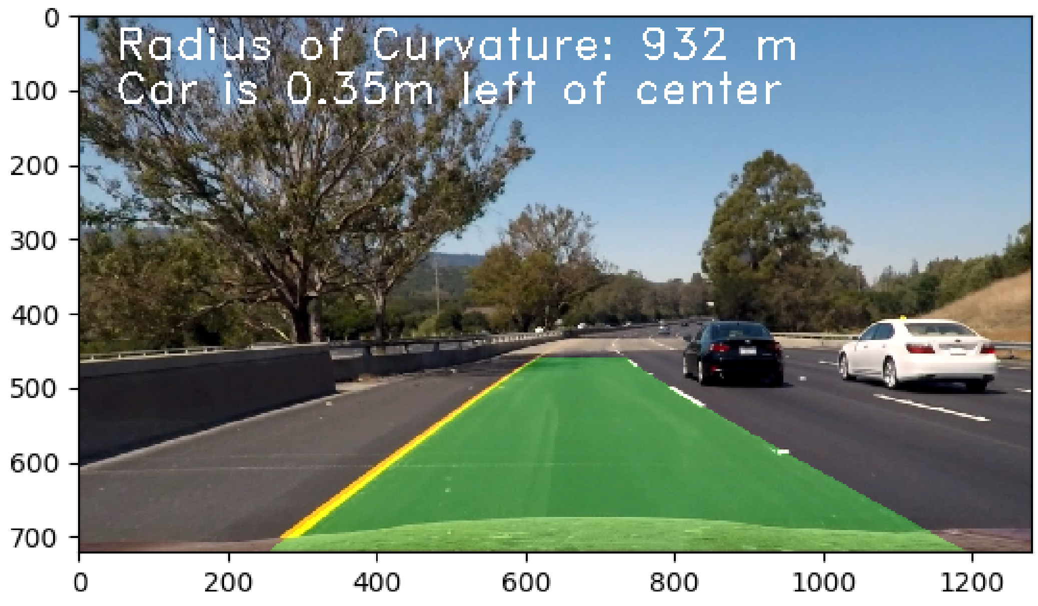
**Step 7:** Determines the radius of curvature of the lane line & the relative distance of the car to the centre of the lane line (& if right or left of that centre line), and add this information to the image in regular perspective. This is implemented in two parts.

Firstly, in *scripts/lane.py* procedures f*ind_lane_lines_new()* lines 25-30, where data from the histogram which was calculated earlier in determining a starting point for lane line detection, is used to locate the car position relative to the centre of the lane in pixels. The centre of the lane is obtained by knowing the left lane position (or the right) & adding half the lane width (or subtracting half if using right lane position). The difference between the centre of the lane and the centre of the image is then obtained in pixels. As the car position is in the centre of the image, we then know where the car is w.r.t the lane centre in pixels.

Secondly, in the same script, one can see the code to determine the lane curvature is *find_lane_curvature()* lines 192-218, and also where the car location is converted from pixels to meters. The radius of curvature is determined by fitting a polynomial to the lane line (left and right lane lines), and then converting the distances from pixels to meters. The conversion from pixels to meters is done by using the hard-coded conversation factor of pixels to meters for the x & y direction for the given images.

Also in the same script, in procedure *add_curve_and_car_info() i*n lines 221-246, one can see the code that was implemented to actually write the curvature & car location onto the image. In this procedure, it also determined if the car is to the "right or left of the centre of the lane line.

An example of this is shown for the test image test6.jpg below.



Examples of car distances & lane curvature information that is written on the lane filled images for all the all the sample test images can be seen in folder ***test_images/ trackedLaneFilledInfoWritten.***

## *Pipeline (Video):*

The pipeline function was then successfully used to process a video. The output shown the video of lane lines being tracked with solid green ahead of the car taking the video The information on lane curvature & the distance of the car from the centre of the lane is also on the video. This is shown in output_videos/project_video.mp4.

## Discussion:

This section provides a discussion of problems/issues faced in the implementation of the project.

### **Thresholding**:

Most of the effort in implementing this project (for me) was in determining the thresholding function. Much experimentation on different thresholding functions such as sobel gradients, magnitude and directional gradients and various colour thresholding in various colour spectrums (RGB, HSL, HSV etc) were explored in being applied in the regular perspective and the top-down perspective. One of the reasons why this was difficult & long is that it required manually viewing the results in an image to determine which threshold yielded the best results. It was not possible to fully determine the best parameters programmatically.  A great deal of “rule of thumb” knowledge of image processing was really needed to do this.

### Maintaining State Information Between Frames & Blind Search:

Helpfully,  the “blind search sliding window” & “informed search” algorithms and code were provided in the  Udacity classroom. However, due to time limitations, these were just both used one after the other, and no attempt was made to avoid the more lengthy “blind search” by maintaining state information between  frames.  Thankfully I have a fast computer that was able to do the analysis, but this could become an issue if the analysis was unable to keep up with the movement of a car in “real time”.

Also, without maintaining state information,  the lane drawing, and curvature & car location information changes from frame to frame, and does not allow for the later support of a smothing function to avoid jitter.

**Lack of Smoothing Function over Frames:**

As there is a lack of a smoothing function over the frames, there would be a lot of jerky movements in a car that would be autonomously reacting to this  information. This jerky motion would be uncomfortable to a human occupant.

**Lack of Error Detection & Handling:**

Again due to time contraints for this project, there really was no error analysis or checking that was done, and no software error handling provided.  Functions could fail for many reasons and the result would be the program crashing. This would not be acceptable for production level code. In particular, the lane finding algorithm has many places where this could happen.

**Environment Factors and Road Lane Markings:**

The techniques here in this assignment absolutely rely on fairly robust yellow or white lane markings on the roadway. However, many environmental influences such as weather (snow, rain),   road construction, may cause the lines to not be available or readily detected, or something suddenly on the road could  hide the lane markings.  In that case, the pipeline would not work.