SELF DRIVING CAR NANODEGREE – TERM 1 / PROJECT4 – Advanced Lane Finding

ADITHYA RANGA
March 2016 – COHORT STUDENT

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.

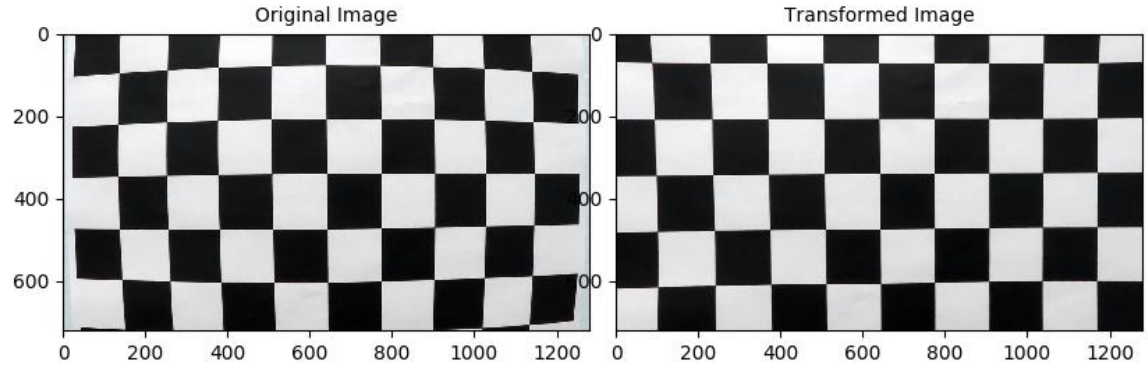
CAMERA CALIBRATION:

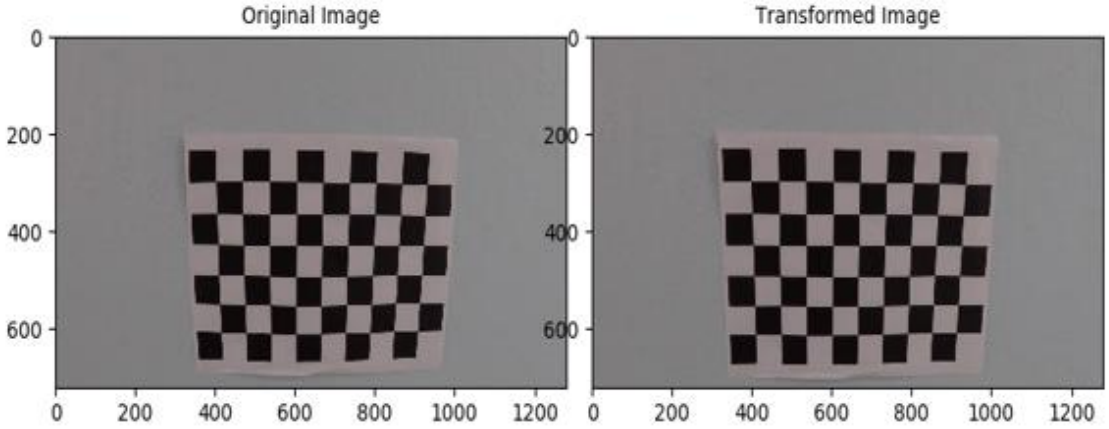
(See in advanced-lane-detection.py from line #11 to #79)

Firstly we need to find the camera matrix and the distortion coefficients for the camera used for lane detection. This is a calibration procedure where the object points (3D) from the real world are mapped to the image points (2D) on the images. I start by defined the grid space of the chess board images provided i.e. 9x6. Then the object points are set up assuming the chess board images are fixed on (x, y) plane at z = 0. Hence we define the grid points for objp and objpoints are appended when the corners are detected. Image points are appended with the pixel positions of the corners in the image plane.

Image and object points are used to calibrate the camera and return the distortion coefficients and camera matrix using cv2.calibrateCamera () function.

The distortion coefficients are applied to test images (please see undist_images (img, mtx, dis)) using cv2.undistort() function to correct for the distortion.
Below shown are couple chessboard images – and also the undistorted image after applying the calibrated distortion coefficients and correcting them.

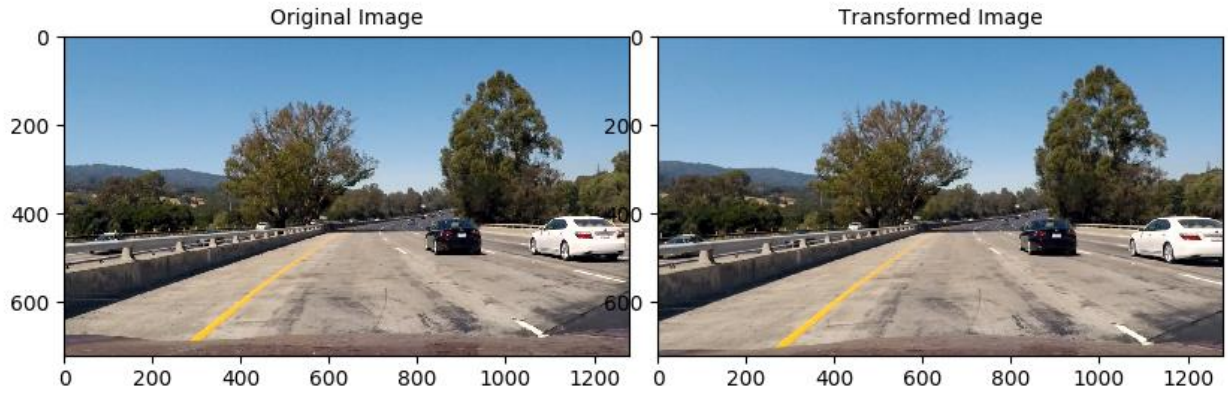




PIPELINE:
(TEST IMAGES):

Please see lines # 522 – 548 for the test_pipeline (img, mtx, dist) function.
Firstly in the pipeline we apply the distortion coefficients & camera calibration matrix to undistort out sample test images.

Below is one such example where we have the original image on the left and the undistorted transformed image on the right. Please see the car hood in the bottom to apparently notice the change.



WARPING LANE IMAGES:

In this next step in the pipeline we apply a perspective transformation on the lane images to transform the images to top view.
This is achieved by using the perspective_transform(img) function in line # 79-100.

We use the cv2.getPerspectiveTransform (src, dst) function to compute the warp matrix which is then applied to the image using cv2. warpPerspective() function to warp the image.

Importantly the perspective transformation is done to mark the region of interest i.e. the lane lines in the original test image and transform into the top view.

For the test images we select our region of interest or the source points to be quadrilateral pixel positions covering the lane in front and then the destination points onto which we want to compute the warping matrix.

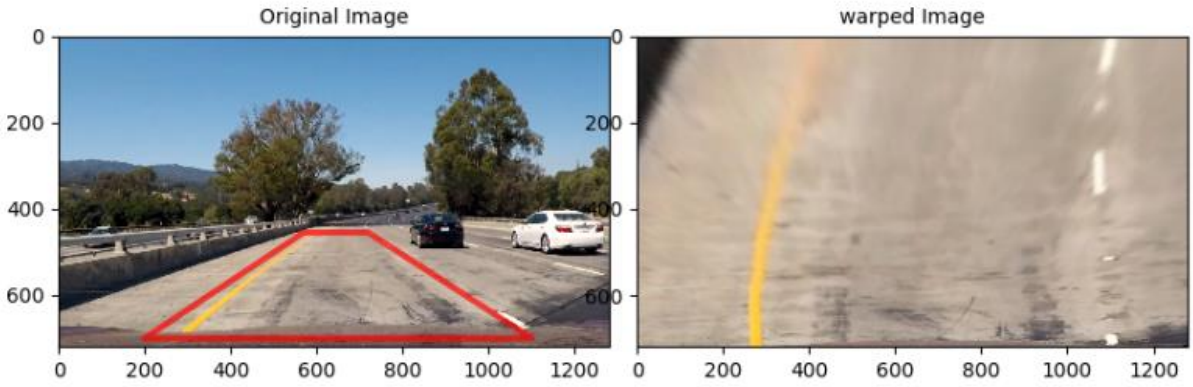
For the pipeline (used for test images and video) we use the following source and destination points for all the images:

Src: [[570. 455.] [720. 455.] [1100. 700.] [200. 700.]]

dst: [[200. 0.] [1080. 0.] [1080. 720.] [200. 720.]]

Below is one such test image – The left original image shows the region of interest marked on the figure using red lines.

The right image is the warped one highlighting the lane lines in top view.



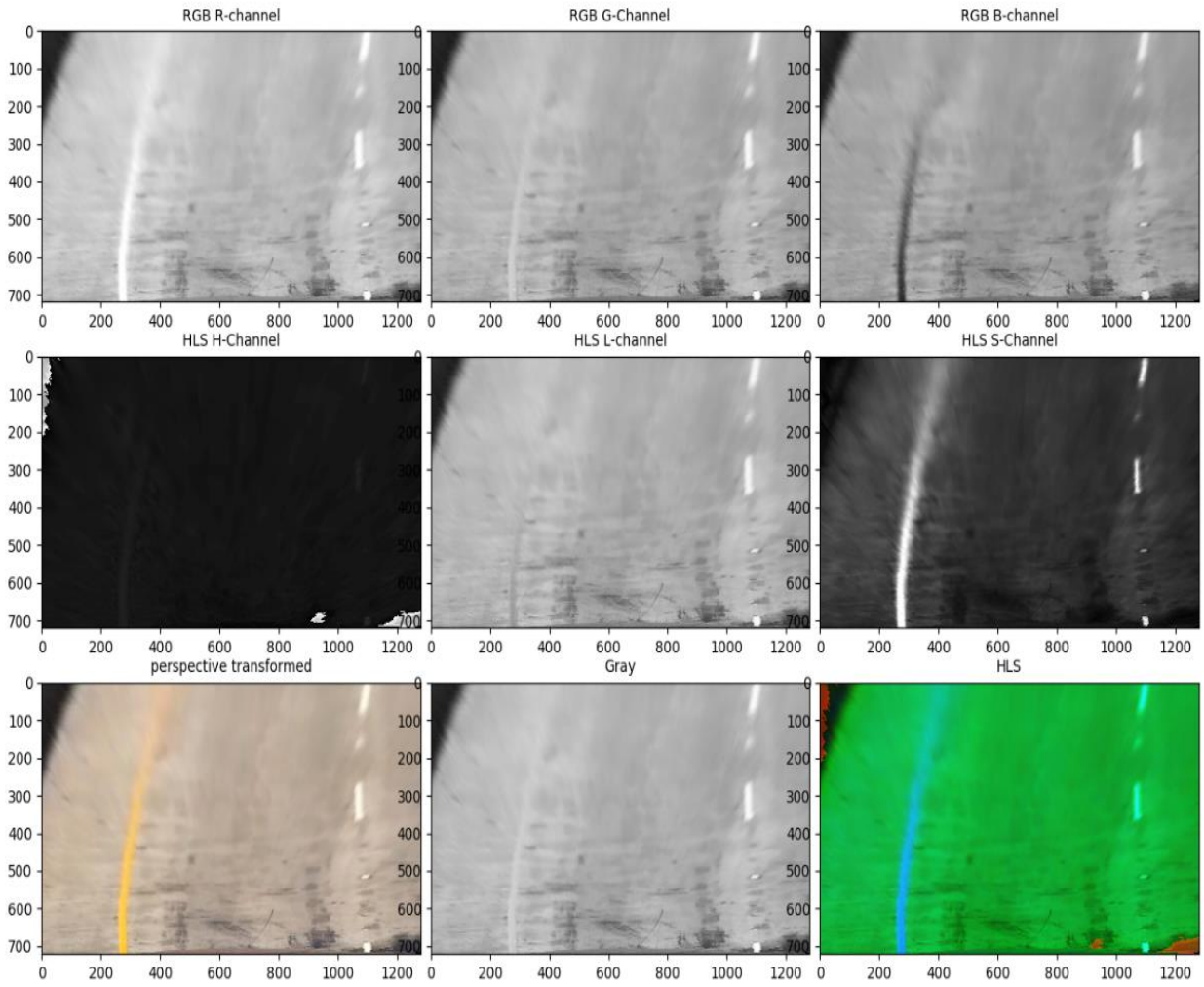
SOBEL GRADIENT THRESHOLDING:

The next step in the pipeline is to use gradient thresholding technique on a single color channel of the warped image to extract the pixels along the lane lines from the original warped image.

For this I implemented the sobel gradient function *sobel_abs_thresh (), sobel_magnitude_thresh (), sobel_dir_threshold (). (See lines# 174 – 234).*

Now the input image should be a single color channel image and I did investigate all the color channels in the RGB, HLS and gray space to examine which works best for our case.

Below image has all the color channels highlighted and apparently S channel worked best for me.

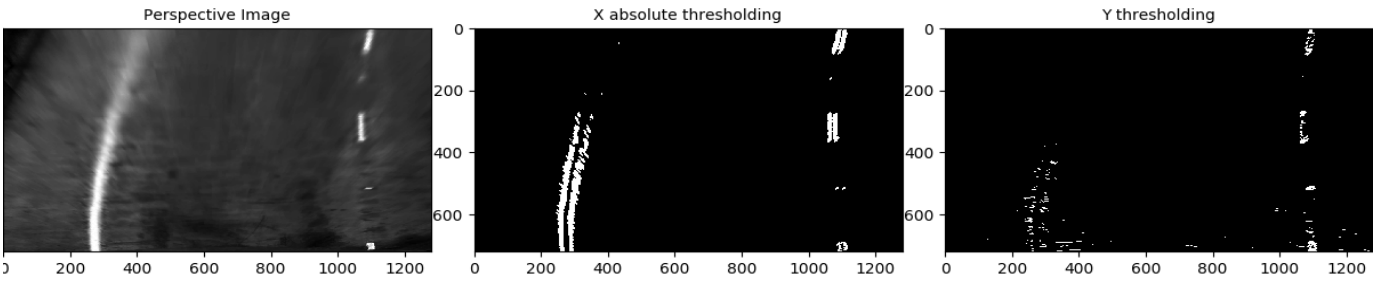


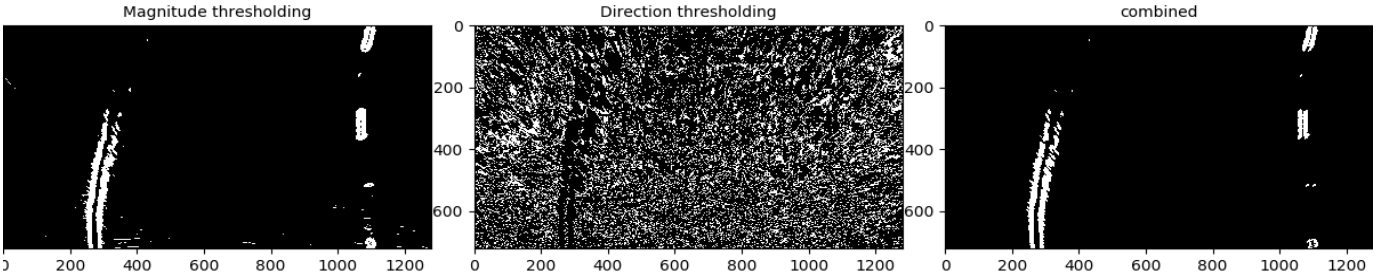
I used the HLS S – channel as input image for all the sobel gradient thresholding functions.

I choose a kernel size of 7 and for gradient in x and y direction and magnitude gradient a threshold of (30, 150) worked best for me.

I then choose to logically use a combination of these gradients in all directions to have best combined binary output image highlight the lane pixels. Using only the gradient in x orientation worked best for me.

Below images show the absolute sobel gradient applied to the images in all orientations & the binary output images.





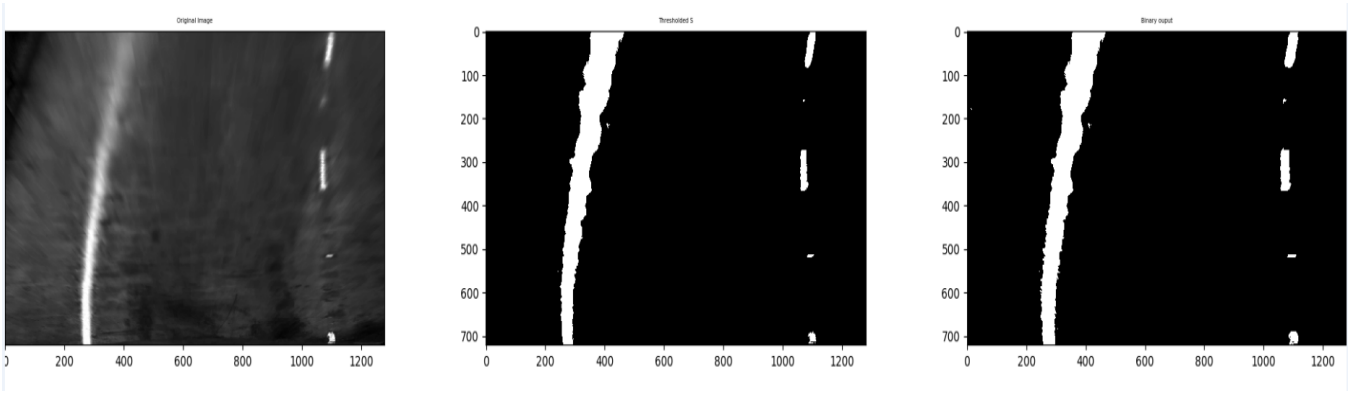
COLOR THRESHOLDING:

Parallel to the gradient thresholding step in the pipeline I also applied a threshold for the HLS s channel to extract the lane pixel positions.

Min/Max threshold values of (100, 255) worked best for me. Below is an image with the applied color threshold.

Finally I choose to extract all the pixel positions from the combination of color thresholded and gradient binary outputs.

The final binary image is also shown in the figure below:



FITTING LANE LINES:

Once I was able to robustly extract the lane pixel positions using the pipeline as described about – I tried to fit a second order polynomial to get my lane lines.

Firstly I used the binary_output image from the test_pipeline () as an input to my fit_lane_lines () function. (Line# 290 – 393).

In the fit_lane_lines () function I compute the histogram of the pixels over 10 sliding windows in along the columns.
Using the histogram in each sliding window the base x pixel positions in the left and right quadrant of the binary image are identified – potential two lane lines. I choose to use a window margin of +/- 80 pixels to search for the pixel positions.

Once all the lane pixel positions in all the windows are identified I use an np.polyfit () function to fit a second order polynomial for both the left and the right lanes.

RADIUS OF LANE CURVATURE / CAR DISTANCE FROM CENTER:

Used the function calc_lanecurvature () (line# 425) to calculate the lane curvature for the right and left lanes from the polynomials fitted.
First I used the conversions for in x & y from pixels to meters as mentioned in the lectures (website).

Approximately 3.7meters/pixel in the x direction and 30 meters per pixel in the y direction.

Using the formula mentioned the left_curveradii and right_curveradii is calculated. The average lane curvature is then (left_curveradii + right_curveradii)/2.

Using the right and left intercepts respectively the car position from the center is calculated as follows:

```
left_x_int = left_fit[0] * h ** 2 + left_fit[1] * h + left_fit[2]
right_x_int = right_fit[0] * h ** 2 + right_fit[1] * h + right_fit[2]
lane_center = (left_x_int + right_x_int) / 2

center_distance = (car - lane_center) * xm_per_pix
```

X intercepts of the left and the right lanes are respectively calculated using the maximum y value – h. The car position is hence the difference between the image mid-point and the intercepts midpoint.

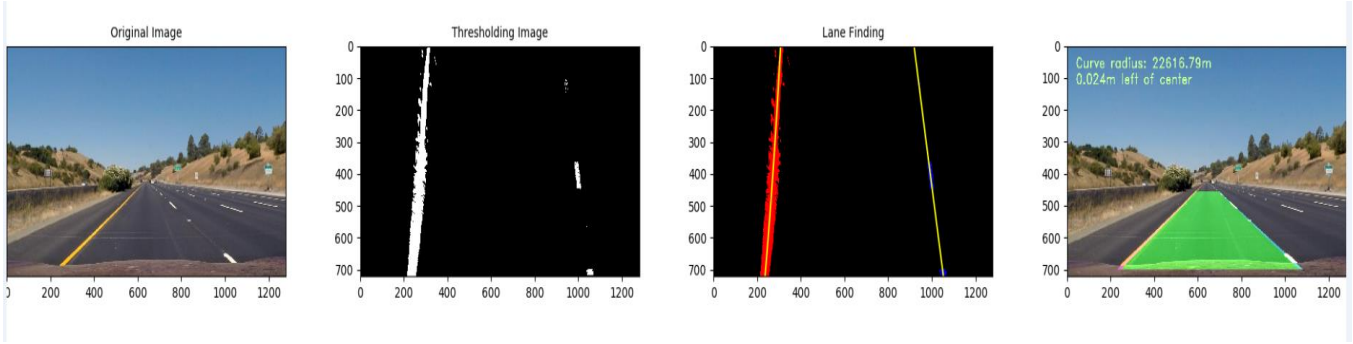
DRAW LANE ON ORIGINAL IMAGE:

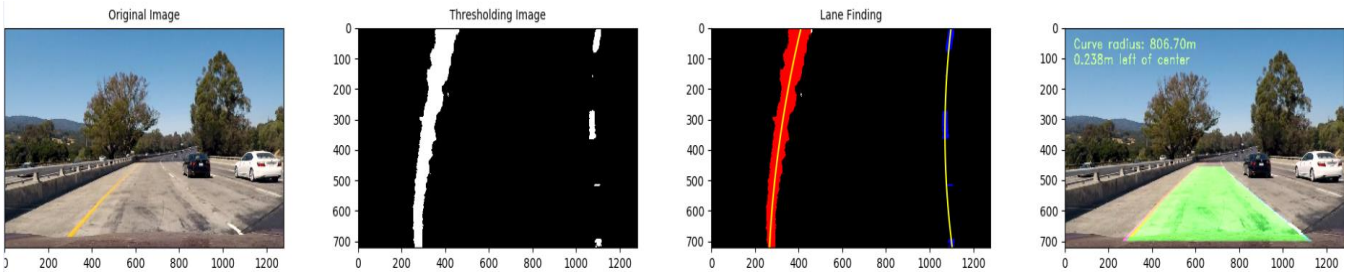
Final step of this pipe line is to take the original left and right lane fits and fit a polygon. Then we plot this polygon back onto the image and then unwarp the image back to the original image using the inversing warp matrix generated in the perspective transform step.

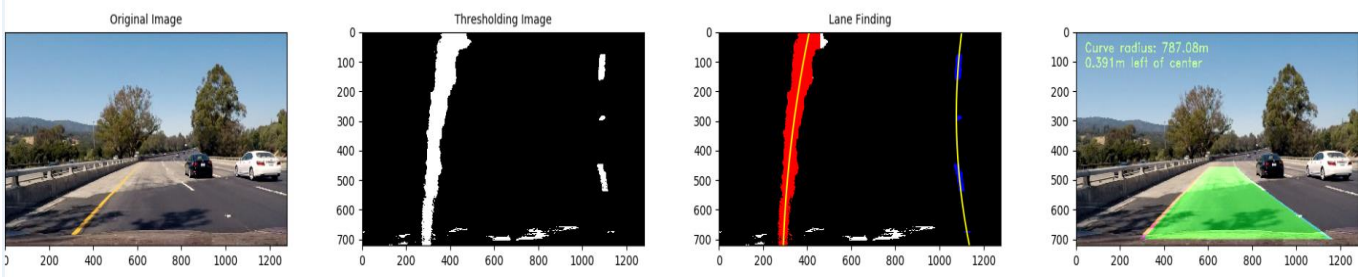
Please see the function draw_lanes () (line # 478).

IMAGE OUTPUT PIPELINE:

Below are results of running the test images through this pipeline:







PIPELINE (VIDEO):

Link for the final output video – (output).

For the video pipeline please see the process (img) function in the code from lines # 670. This uses the same pipeline as mentioned above for processing the images.

Now the additional changes included to robustly track the lanes in the video are as follows:

- The camera is capturing video at like 30 fps i.e. the lane lines won't change by a large value from image to image. Keeping this in mind additional function fit_lane_prevframe () is used to use the lane line fits from last frame to fit the lanes in the next frame rather than continuously searching for the lane pixels using sliding window in each frame.

- While I implemented that logic I created a 'Line' Class – thanks for the lectures and online forums and slack channels for help on this. The main attributes in the Line class the self.detected and self.best_fit which are updated using the lane.update () method.

- After fitting lanes from every frame the left and right fits are compared against last frames best fit values and appended to an array of last 10 current_fit values. Best fit values are an average of the last 10 current fits.

- In case the lane lines are not detected in a frame or are wrongly detected they are rejected and the last best fit values are used for drawing lanes. This is only for last 10 frames and the pipeline uses the sliding window search after that to detect the lane lines again.

DISCUSSION:

Most difficulty occurred to tune and select the best threshold values, because of the shadows, lighting, road color etc. Also the lane lines width and location are not equally distributed in each frame and different videos.

Some of the things that I will consider for improving the robustness is:

- Filtering the images – each frame.
- Using a dynamic and adaptive threshold to choose the color and gradient threshold values
- Dynamically search each frame to choose the range of interest values to extract the lanes.

I will keep working to implement these changes to the pipeline.