Advanced lane Finding

Author: Aneeq Mahmood

Email: [Aneeq.sdc@gmail.com](mailto:Aneeq.sdc@gmail.com)

The goal of this work has to enhance lane finding techniques developed at the start of the term and do advanced lane detections which can especially work well against shadows, tire-tread marks, missing lane lines etc.

The individual rubric points are addressed below

# Files Submitted:

Following files are submitted for this work:

*main.py* : File used to create the video output.mp4; the pipeline is present inside this file

*output.mp4: output video file*

*main\_without\_pipeline.py: This script is used just to view output images after different stages, on the contrary main.py file does not show any output images. The backbone of the code is similar in both scripts. However, there is no pipeline function to accept images from a video stream in this script. It runs on the images in the test\_images/ directory*

pipeline\_helper.py: This file contains all the helper functions needed by the pipeline

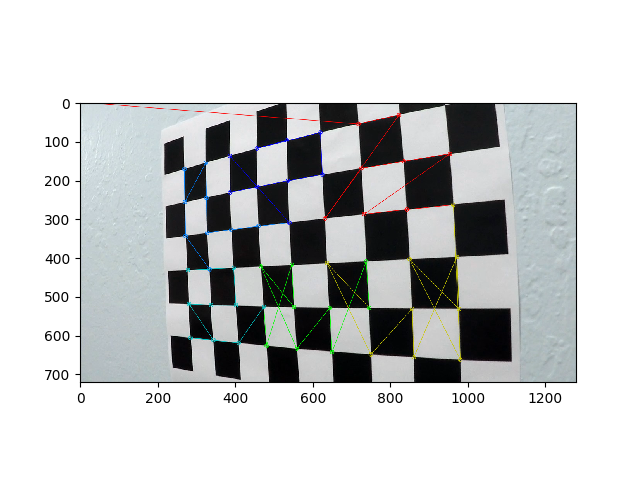
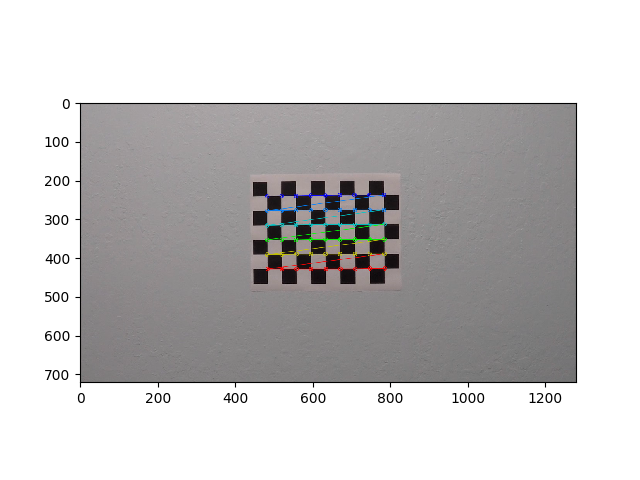
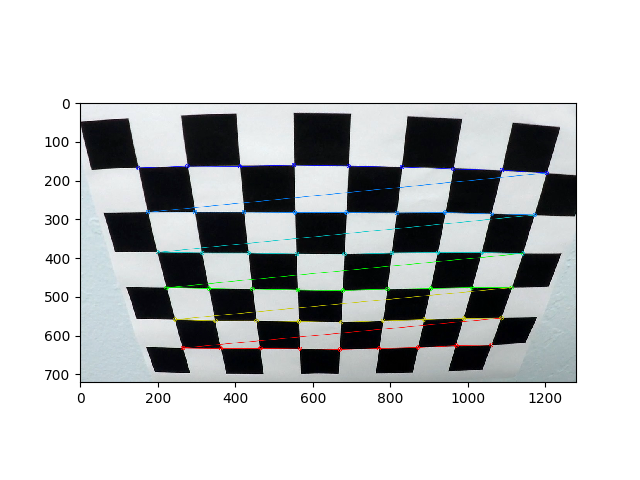
convol.py: Contains the documented code regarding convolution

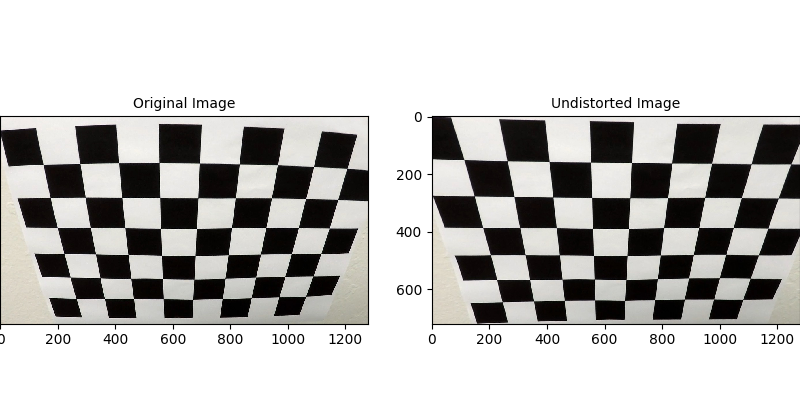
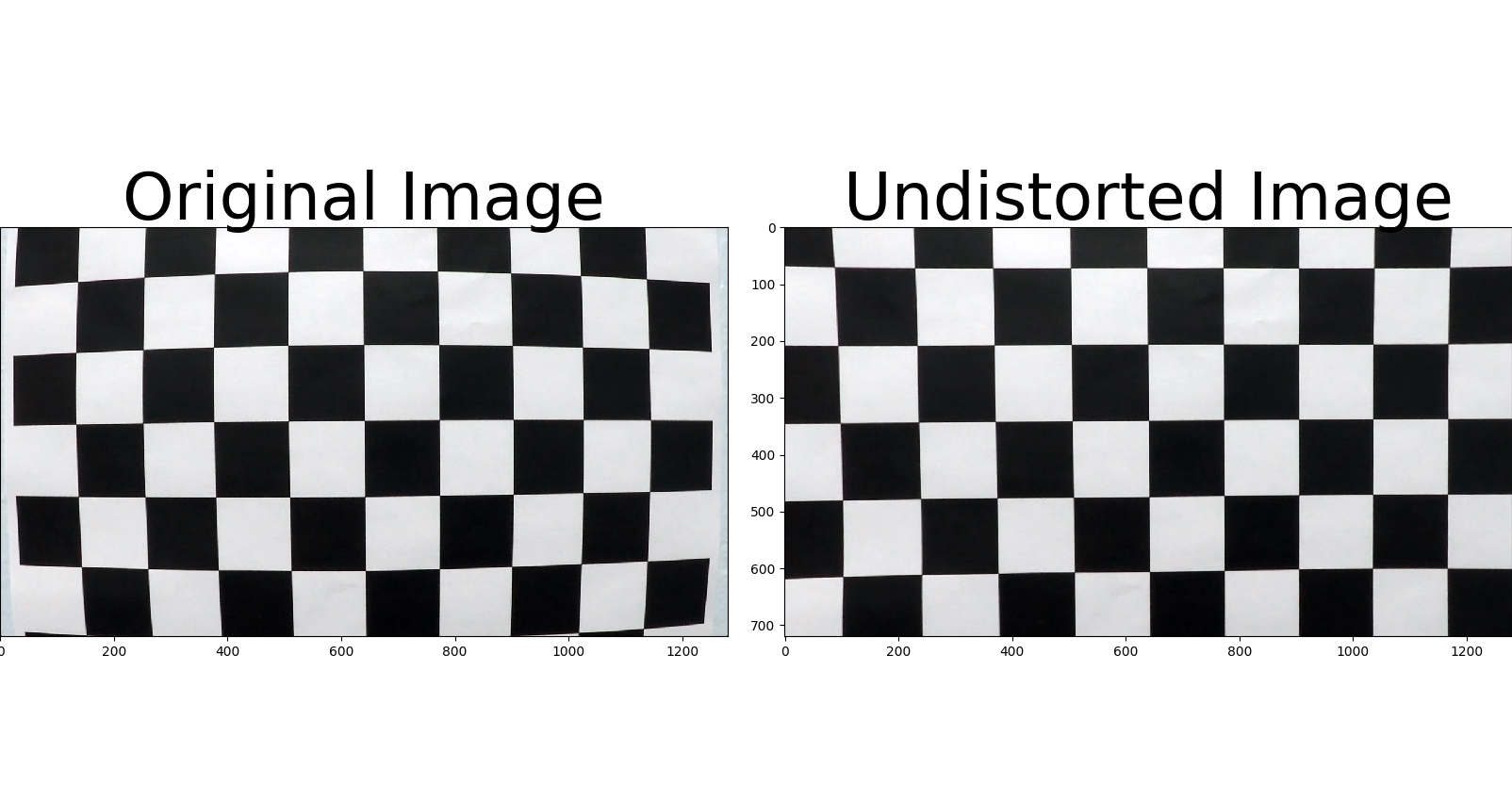
This write-up is also part of submitted work.

The individual Rubric points are addressed below

# Camera Calibration.

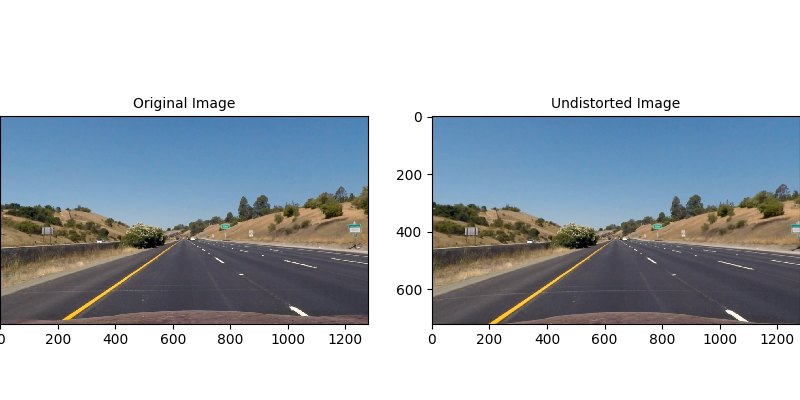
The camera calibration is using the calibration images inside the “camera\_cal/” directory. The cameraCalibMatrices() function in the pipeline\_herlper.py script uses these images as inputs, tries to find the corners, and if successful, stores the object and image points related to the image. Afterwards, cv2.calibrateCamera() is used to obtained a camera matrix and output vector of distortion coefficients. This whole function of cameraCalibMatrices() is based on the exercise which has been used in the classroom videos. Following images show the some cases of correct and incorrect chessboard corner detections. The object and image points for wrong corner detections are skipped. In total, out of 20 calibration images, only 3 images have been observed with wrong corners.

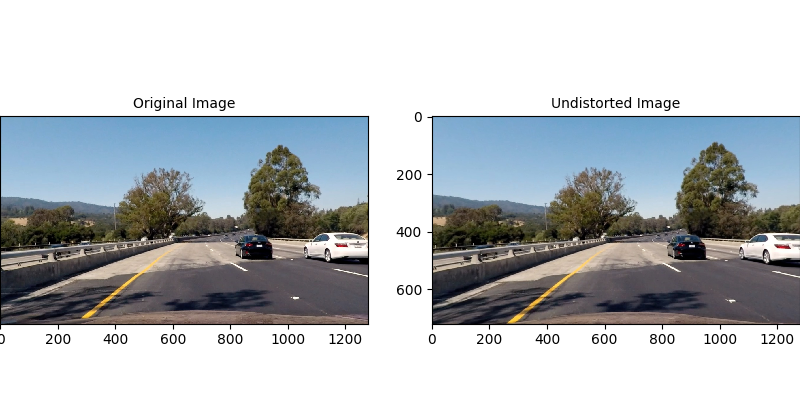


The result of distortion correction, using the distortion matrix and distortion coefficients, is shown below on two images which have also been used for calibration. The functionality is present in cameraDistremove() function inside pipeline\_helper and is based on cv2 built-in functions.

# Pipeline Description:

The first step in the pipeline is to use the above mentioned functions to undistort the image passed as input. Below is the result of this image distortion shown for a couple of test images provided in the repository.

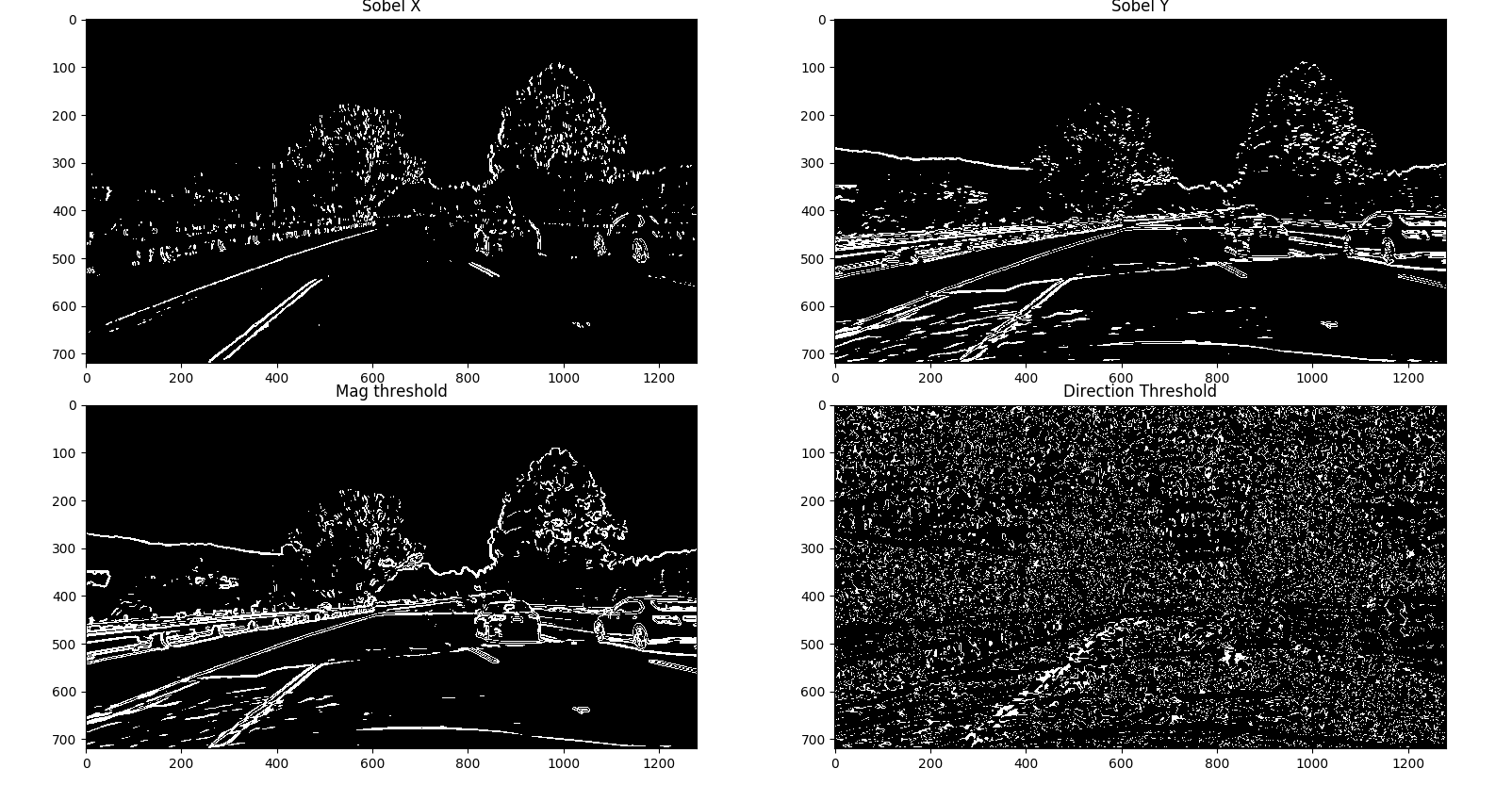


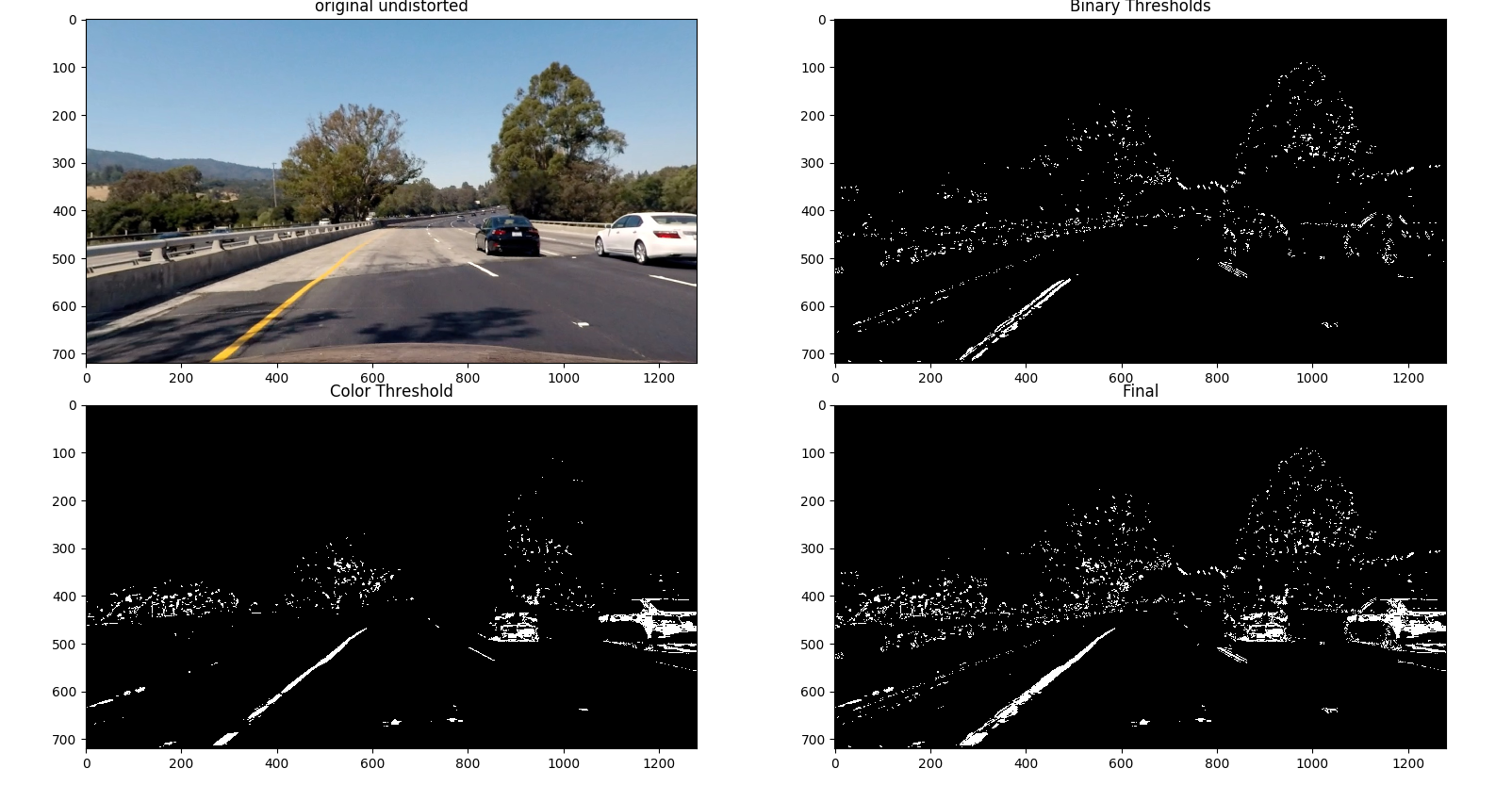


# Binary Image Creation

The binary image to remove edges around the lane and to highlight lanes further is done in two steps. First calcGradients() function in line 88 of main.py is called to detect edges. This is done by calculating derivate in x and y direction using Sobel operator, combining the magnitude of the derivate, and doing directional threshold detection. The thresholds and Sobel kernel sizes are obtained through experimentation, and are described at line 39 of main.py. The four techniques are combined in the same way as the video lesson to create a single image. For all these four operations, instead of gray picture, the R channel is used, as it has proved to be more robust to detect lanes.

For color threshold removal, S channel is used as it also removes noise from the image but preserve lane lines. Line 80 in the main.py file has the function colorThr\_s() which performs threshold on the S channel of the image. Finally, the binary images form color detection and gradient technique sis combined using the getBinaryImg() function in line 83 of main.py. The results of individual processing are given below. The first picture below shows four gradient based techniques to get the binary images. The second shows results of all operations on the input image.





In this second picture, top left is the original image, top right is the combination of all gradient based techniques, bottom left is the result of color threshold-ing, and last one is the final image.

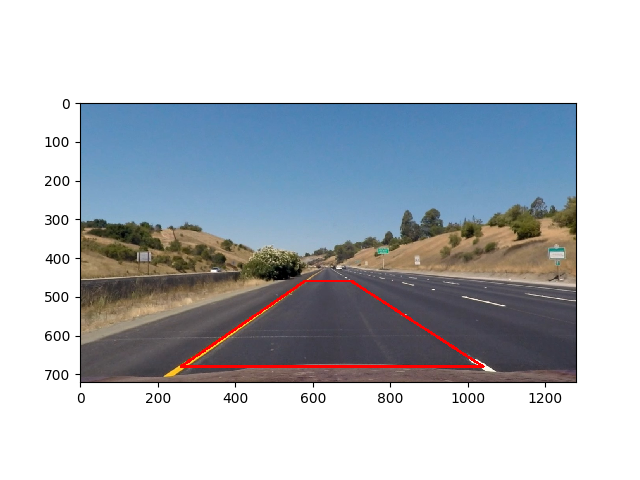
## Perspective Transformation

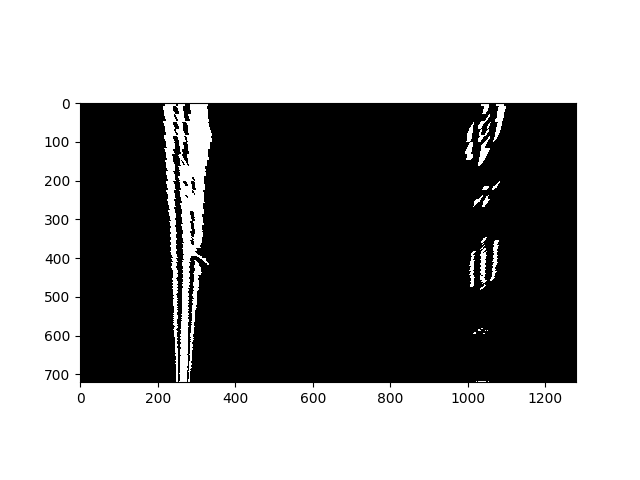
This process has been inside the main.py script at line 86; this function itself calls the getPerspectiveTransform() to get warp matrix ‘M’ and warpPerspective() from cv2 to do perspective transform on the image. For this work, the warping is done on the lane line using the following x and y coordinates vectors, defined in the pipeline\_helper.py file.

x = [580,700,1040,260]

y = [460,460,680,680]

The lane area marked by the above coordinates is shown below



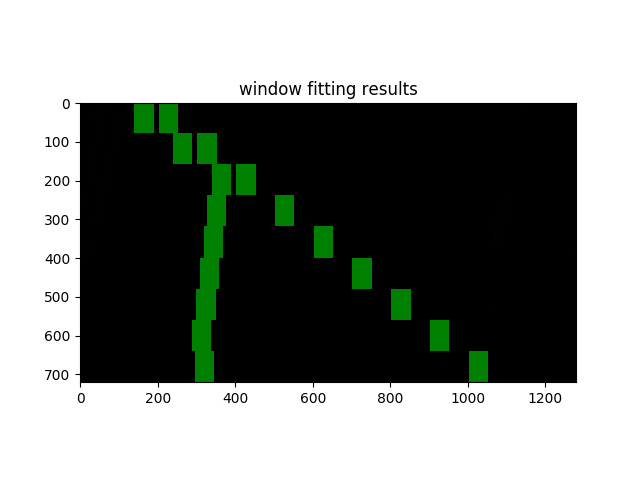
The warped image is shown below

# Lane Line Detection:

Lane lines detection on these binary warped images is done via convolution using the find\_window\_centroids() and find\_all\_centroids() functions in line 89 and 92 of mian.py. These functions are defined in the convo.py and the basic code is similar to what is presented in the classroom lectures. A couple of changes are made to the code though; firstly initial left and right centroid are sought in last 1/8 of the image. Moreover, as these centroids only give the x-coordinates of possible lane marking, a y coordinate is also obtained by taking this y coordinate as the center of the area being convolved (in this case, center of the last 1/8th of the image).

Thus at the end of find\_all:centroids(), not only x- but y-coordinates of the lane marking are available. Thus we have points on both left and right traces of the lane, which are used to obtain proper lane markings using line fitting techniques.

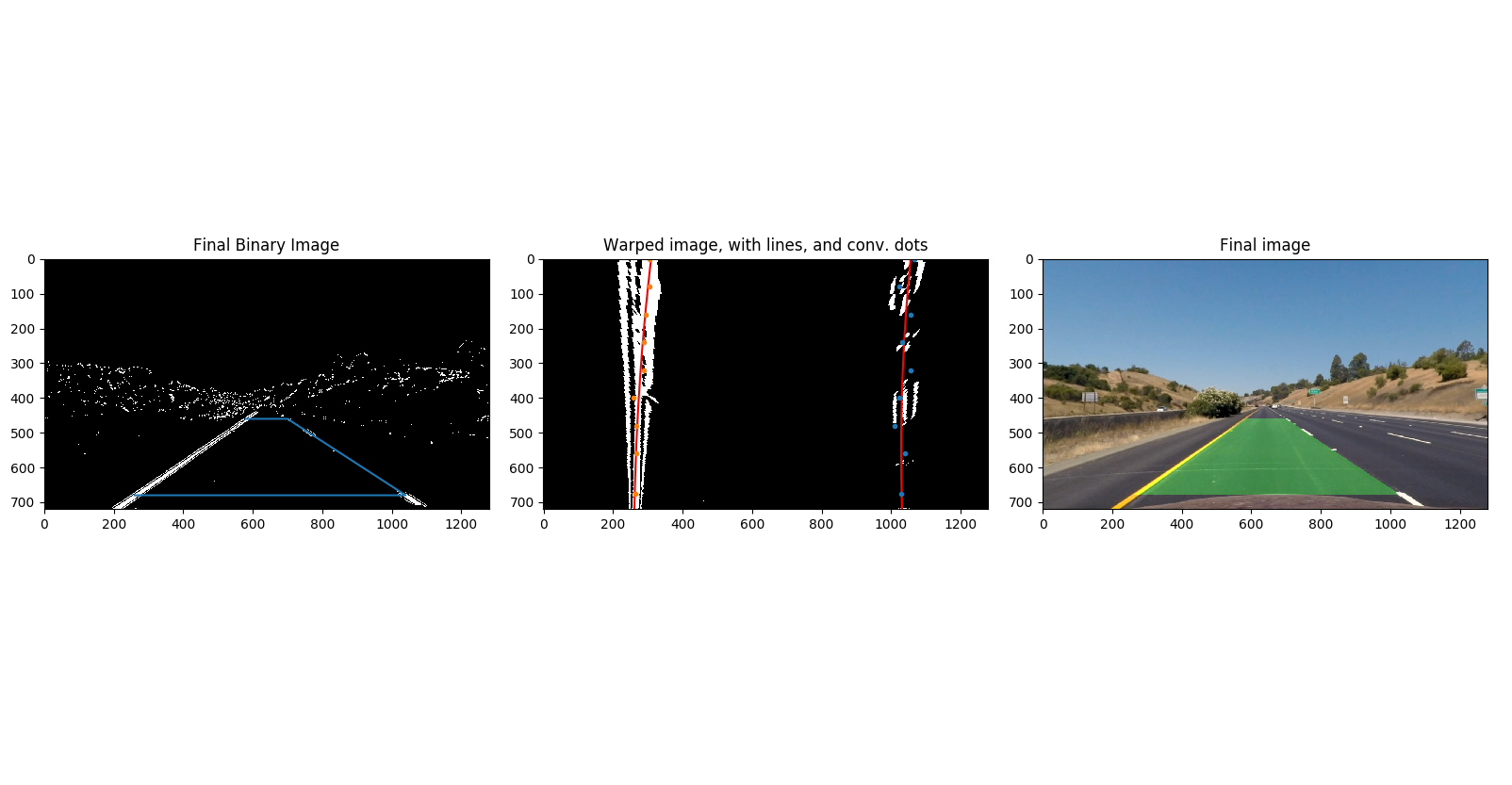
Here at this point, sanity checking is introduced inside the function checkLaneWidth() inline 518 of the pipeline\_herlper file, as convolution does lead to wrong answers. Following image shows the case by showing wrong window fitting after convolution.



To avoid such scenarios firstly the left and right centroids should stay in their respective halves of the image, the difference (right-left) of centroids should never be negative, and any left centroid should not be negative. Lastly, as the lane width is 700 pixels, the difference of right and left centroids should not differ by a tolerance of 100 pixels (0.5 m) around this value of 700. If these criteria are met, then centroids are considered ‘sane’, they are used to obtain coefficients for left and right lane marking using numpy polyfit() function. These coefficients are saved in a deque() ring buffer of size 30. The ‘sane’ centroids are also saved as the most recent reasonable lane points.

To obtain the right lane lines, between line 100 and 110 of main.py, first of all it is checked if the deque() is full of ‘sane’ values. If yes, then the average of coefficients of left and right lane lines are used to draw lines using fitVector() function (line 116). If the buffer is not full, then the current ‘sane’ values are used. If the current values were not reasonable, then most recent ‘sane’ values for centroids are obtained; if no reasonable old values of centroids are present, then this frame is dropped. Once the left and right lane fits are obtained using fitVector() function, radius of curvature is obtained for the left and right lane lines using getRadiusCurve() on line 119, and their average is used to obtain a final value. The getCameraOffset() function is used to see how much a the central camera (the car itself) is off from the centre. Lastly, the lane markings are then used inside the warpToColor() function on line 128 to map the lane lines on a unwarped image. This function uses the code provided in the lessons to perform this action, and shows the marked area on the initial colored image.

The results of the whole process are summarized in the figure below:



As it is a long and arduous process to find the write CNN architecture, the focus has been on using an existing CNN architecture. The LeNet architecture has been used in the previous project, which could have been a good starting point for this work, however, a big limitation of that architecture is its small image input size, which is 32 \* 32. On the other hand, the smallest possible input image size from the Udacity simulator is 160\*320. Down-sampling the images from the simulator to suit the LeNet architecture would have led to loss of a information. Thus, this work has primarily used the CNN architecture from [1], where the input image size has been 66\*200.



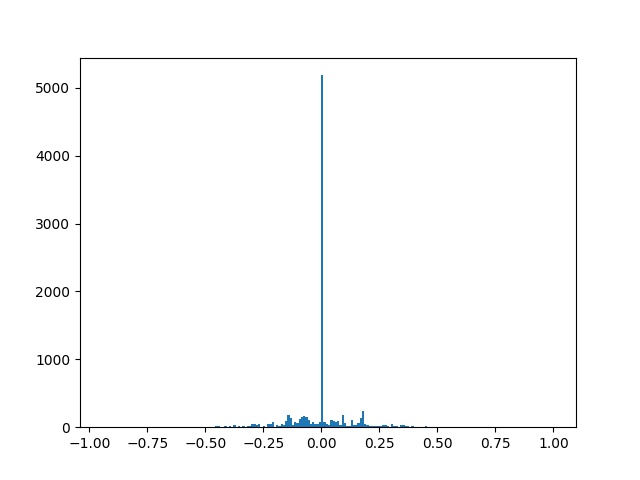
As shown in the figure above (which is an image from the Udacity simulator), information at the top and bottom of the image is not helpful in detecting lanes for steering the car. Hence, if the above image is cropped from top and bottom, the resulting image will have height similar to what is required by [1]. Hence, the cropped images from the simulator will be suitable for training using the model from [1]. Moreover, this CNN model has several layers and employs filters of varying depths, which has come as a result of extensive investigations as discussed in [1]. Hence, this architecture can be used for this work as well.

The architecture used in this work is shown in the figure below. The input image is first cropped by 74 lines of pixels from top, and 20 lines from bottom, to focus the learning on the road .Afterwards the image is reduced to 66\*200 and is normalized. A few other modifications has been made to the initial architecture of [1] mainly due to the lack of details. For example, whereas kernel size and strides are described in [1] for the different convolutional layers, other details such as zero-padding or type of activations are not discussed. In this work, zero-padding of 3\*3 is used with Relu activations in the first three convolutional layers, and no-zero padding is used in the last two convolutional layers (with Relu) before flattening. The reason for using padding is that without it, information at the edges will be ‘dissolved’ in the first few layers. However, edges contain extremely important information such as lane marking, ledges, or dirt roads. Thus to make this information last several layers, zero padding is done in the first three convolutional layers. It should be noted that in the figure below, this zero padding is shown within the same layer. However, in the model\_generator.py script, this is done in a separate layer to breakdown individual steps.

To reduce overfitting, dropout of 50% is introduced after the fully connected layers to reduce the chances of overfitting. To further limit overfitting and increase size of dataset, image augmentation is employed whose details are discussed later. To avoid tempering with learning rate, ‘Adam’ optimizer is used. This choice has been made to reduce the hyperparameters as learning rate for the Adam optimizer can be left untouched. Also, 20% of the dataset is left aside for validation while the rest is used to train the mode.



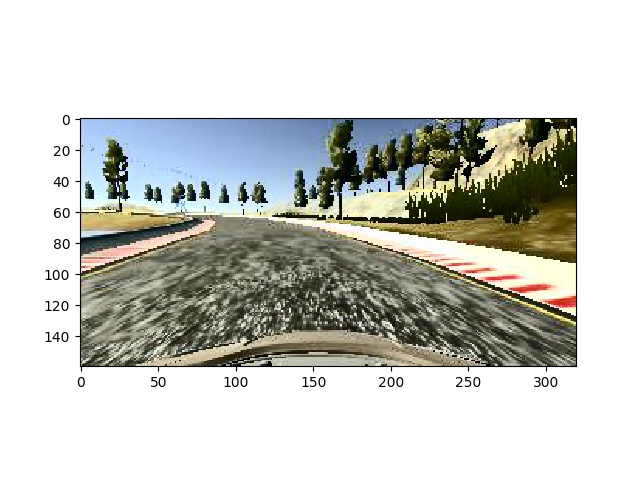
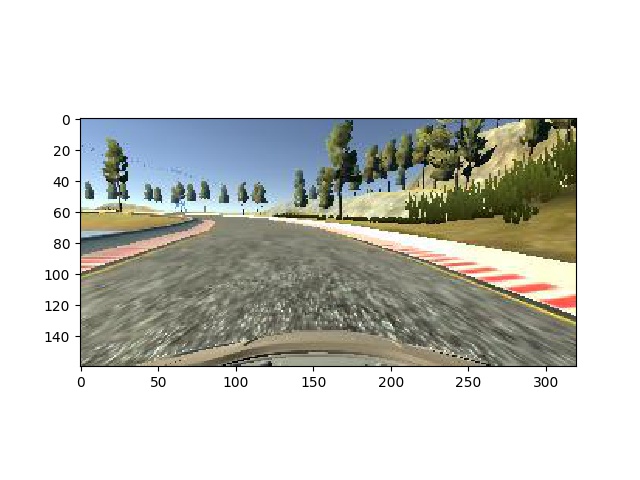
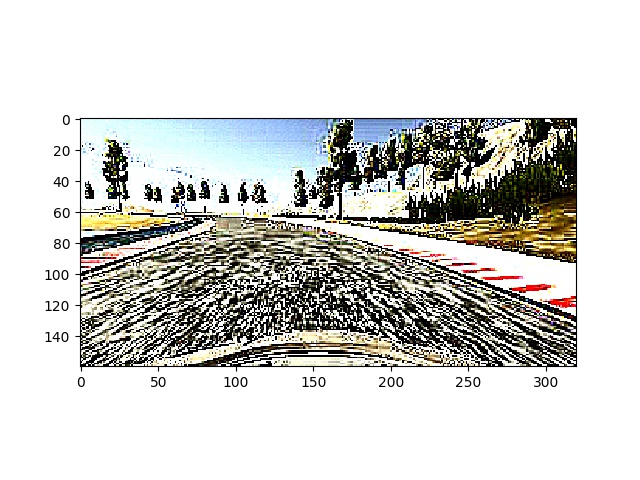
# Training and Validation Documentation

Whereas choice of the mode and model architecture are described above in detail, information about training the network is not yet described. This section deals with this issue. To train this network, an initial dataset has been provided. It has turned out that this dataset was sufficient to train the network. However, this dataset has too many steering values as zeros corresponding to driving straight. This is evident by the following histogram. 

As a first step to add additional images, the simulator is run on those sections where there exist sharp turns. These are sections where there is water and dirt road in view inside the track 1 of the simulator. Afterwards, 50% of images corresponding to steering values less than 0.25 are randomly removed from the dataset. To increase the dataset, all images are also flipped and added to the original dataset.

This dataset is randomly shuffled and 20% of it is set aside as data for validation. Overall, there are around 10372 images out of which 2074 images are for validation and the rest are for training. To train the model, training images are provided from a generator. Inside the generator, for a particular batch size, images are retrieved from the directory data/IMG. For a particular image, it is decided randomly to use either one of the center, left or right camera view. As the default camera view is center, hence for left or right camera view, the steering values are compensated with a value of 0.238. This value has been obtained through hit and trial and is a result of experimentation.

The image is then passed through “histogram equalization” and then through augmentation techniques. This latter augmentation is done using the library from [2]. At first, Gaussian blur and random brightness is added to an image. Afterwards the images are randomly sharpened and then contrast-normalized; the resulting images are collected as output of the generator. Following images show the original, histogram-equalized, and the final image for training, respectively.

,

This image is then cropped to remove the top 74 pixel lines and bottom 20 lines, and then resized to fit the model. The training is done for 5 epochs. The training is done over 20000 images and takes several hours. No GPU support has been present to do this training as the AWS cloud support was not available due to NVIDIA driver problems. After failing to set the environment via the AWs, the whole training and validation was done on a normal desktop computer.

# Simulation

The results of the simulation for one lap are contained inside the run2.mp4 file and also the model\_X.h5 can be used to see the performance of the model by opening the simulator in the autonomous model. The car remains on the road all the time. It does get very close to the road marking at the first major turn, but the wheels do not touch the lane marking. Afterwards, the car drives almost in the middle of the road and does not go off the road, hence fulfilling the project requirement.

When the same model was used on the second ‘Jungle’ track, the model performed poorly and the car could not tackle with sharp turns on that track. The reason can be that model is trained on the first track and a large majority of the training is done to drive in straight lines, and not enough sharp turns are present in the data set.

# References

[1]https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf.

[2] <https://github.com/aleju/imgaug>