**Senior Year Project (Spring 2018)**

**Lane Detection (OpenCV Python)**

**Farwa Iftikhar**

**N13426915**

**Submission: 8/31/2018**

**Abstract**

The objective of this project was to implement a lane detection algorithm to detect straight and curved yellow and white lanes on the road using computer vision techniques implemented through OpenCV Python.

**Introduction**

One of the tasks performed by human drivers is to identify lanes on the road and stay in them for smooth driving and avoiding collisions. Among one of the daunting tasks for the future road vehicles especially autonomous road vehicles is road lane detection or road boundaries detection. A robust and reliable algorithm is a minimum requirement as erroneous findings could generate wrong steering commands jeopardizing vehicle’s safety.

The state-of-the-art method to detect road lane lines is to use vision system on the vehicle. In this project well-developed computer vision techniques are employed for detection of lane lines in real time with robustness and accuracy. The front view of the road is acquired using a camera mounted on the vehicle and the input is processed using the below mentioned pipeline.

Project Pipeline:

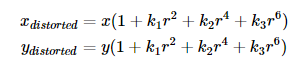
* Computation of camera calibration matrix and distortion coefficients from a set of chessboard images
* Distortion removal on images
* Application of gradient thresholds to find edges
* Application of Hough Transform to generate lane lines
* Perform perspective transform to generate bird’s eye view
* Use of sliding windows to find hot lane line pixels
* Fitting of second degree polynomials to identify left and right lane curves composing the lane
* Displaying the lane lines on the image

**Method**

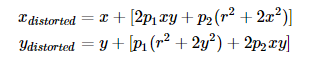
**Camera Calibration:**

Pinhole cameras used most commonly nowadays have significant distortions due to the convex shape of the lens which causes light rays entering the camera to bend. This distortion can be corrected by finding intrinsic and extrinsic parameters to the camera. The intrinsic factors include image center, focal length, scaling factor, skew factor and lens distortion while extrinsic factors include translation and rotation vectors which are used to convert 3D image points to 2D points.

Two major kind of distortions are radial and tangential distortions. Radial distortion causes straight lines to appear to be curved and it increases as you move away from the center of the image. Radial distortion is represented by the following equations:



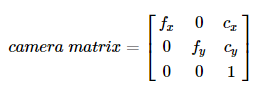
Tangential distortion is caused by the parallel misalignment of the lens and the imaging plane in the camera. It is given by the following equations:



The unknown parameters in the above four equations are known as the distortion coefficients.



To undistort the image the distortion coefficients along with the camera matrix is required which is unique to the camera being used.



Here fx,fy = focal length of the lens used in the camera and cx,cy is the optical center.

In this project the images taken form the stationary positioned camera on the vehicle are first undistorted before processing them further. To undistort the images camera calibration is performed using a series of OpenCV python functions.

Images of a chessboard pattern are taken using the camera which significantly displayed the distortion caused by the camera.

In OpenCV 2D image points are found from 3D real world points from the chessboard image. The 2D points are the points where black squares touch each other in the chessboard image. Function def findChessboardCorners() takes in a grayscale image of the chessboard and the number of inner corners per chessboard row and column and uses cv2.findChessboardCorners() function to find the chessboard corners. In the given case chessboard has *9* inner corners in the x direction, and *6* in the y direction. We will use these as parameters to *findChessboardCorners()*. The return valuel is True if a pattern is found and corners in order of left-to-right, top-to-bottom are also returned.

These corners found are then displayed on the chessboard image using the function *showChessboardCorners()* which uses the OpenCV function *cv2.drawChessboardCorners()* to display the corners on the image as shown in the following figure.

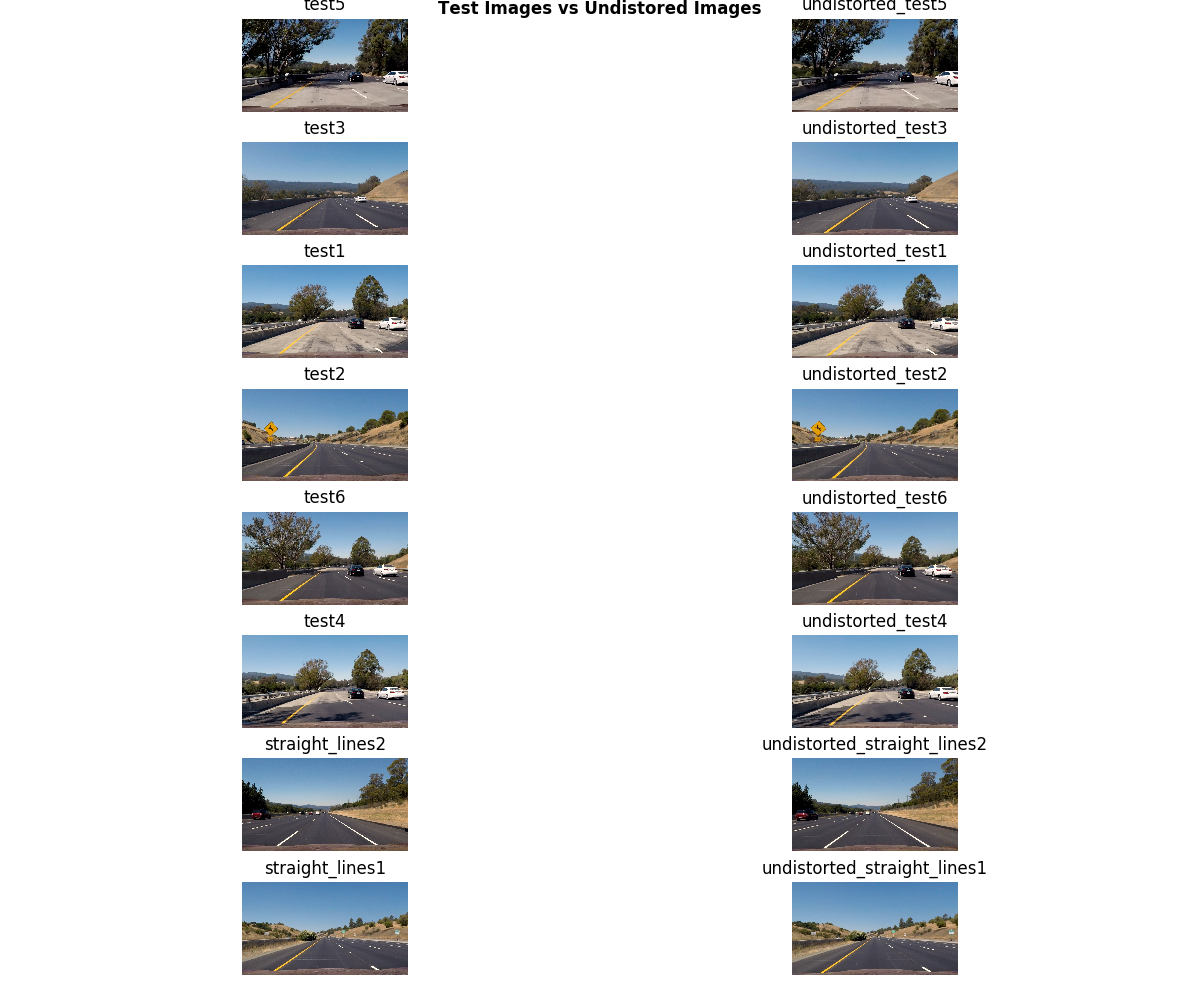


***Figure 1: A distorted image of chessboard with the corners displayed on it.***

Then image and object points are defined to calibrate the camera. The object points are the (x,y,z) coordinates of the chessboard in the real world where the z coordinate remains zero as the chessboard is fixed in the xy-plane. While the image points are the 2D points on the image of the chessboard.

This is achieved in the function *findImgObjPoints()* which takes in the path to a chessboard image and the number of the inner corners. In this function *objp* is a replicated array of coordinates, and *objpoints* will be appended with it each time the chessboard corners are detected in a test image while *imagepoints* will be appended with the 2D points or the pixel position of each of the corners in the image plane which are found using the functions described previously.

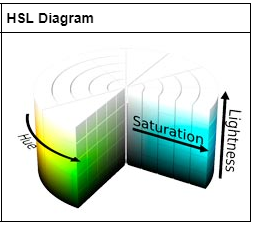
These image and object points are then used in the function *undistort\_image()* to find undistorted image. The function uses OpenCV function *cv2.calibrateCamera()* to find the distortion coefficients and camera matrix along with the translation and rotation vectors which are then used by *cv2.undistort()* to find the undistorted image. The undistorted road images are shown below.



***Figure 2: The road images before and after being undistorted.***

**Alternate Color Space:**

To detect lanes everything except the yellow and white lanes need to be masked out of the image. To achieve this the image is converted into HSL format from RGB since HSL is proven to be best in distinguishing colors. Hue (H) is the perceived color number representation based on the color channels red, blue and green. The saturation(S) represents how dull or colorful the images are while Lightness(L) is how close to white color it is.



***Figure 3: A visualization of the HSL color space.***

Converting to HSL space is achieved in the function *compute\_hls\_white\_yellow\_binary()* which takes in a RBG image. In this function pre-determined ranges of the B, G and R channels are used to isolate yellow lanes by thresholding the image using these values to retain yellow lane lines. For yellow lane lines the thresholding range for B channel is 15-35, G channel is 30-204 and R channel is 115-255. Any pixel between these ranges is set equal to 255. Similarly, white lanes are also isolated using thresholding with specified values in BGR channels. The masked image which contains the white and yellow elements of the image is returned. The following figure presents the results of this process:



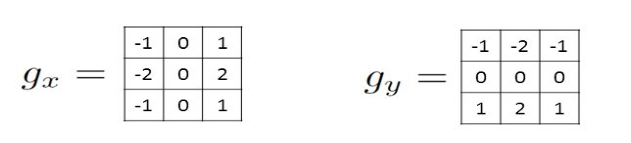
***Figure 4: A road image in RGB space (Left). Same road image in HSL color space (Right).***

**Gradient Thresholding:**

Edges are significant local changes of intensity in an image and occur on the boundary between two different regions in an image and hence important features can be extracted from an image such as lanes and curves. A drastic jump in the intensity from one pixel to the next indicates the presence of an edge.

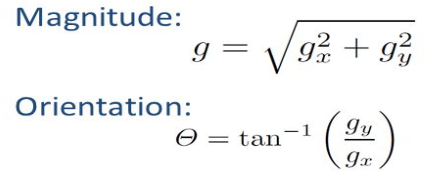
One of the two main categories to detect edges is by using gradient method which detects edges by looking at the maximum and minimum in the first derivative of the image. The Sobel operator is used to perform 2D spatial gradient measurement on the image and accentuates the regions of high spatial gradient that represent edges.

A kernel or filter (matrix of typically size 3x3) is convolved with the image to find a pixel value in the resultant image. The kernel for x direction is rotated by 90 degree to receive the kernel for the y direction. The kernels used are as follows:



The function *abs\_sobel()* uses the built in *cv2.sobel()* function to find the gradient of the input grayscale image in both x and y directions. The image received is scaled so that none of the pixel values exceed the upper limit of threshold after the filtering. A masked image is produced, and binary thresholding is done to keep the pixel values which are within the threshold range. The masked image is returned.

The gradient in the x and y direction can then be used to find the absolute magnitude of the gradient at each point and orientation of the gradient as follows:

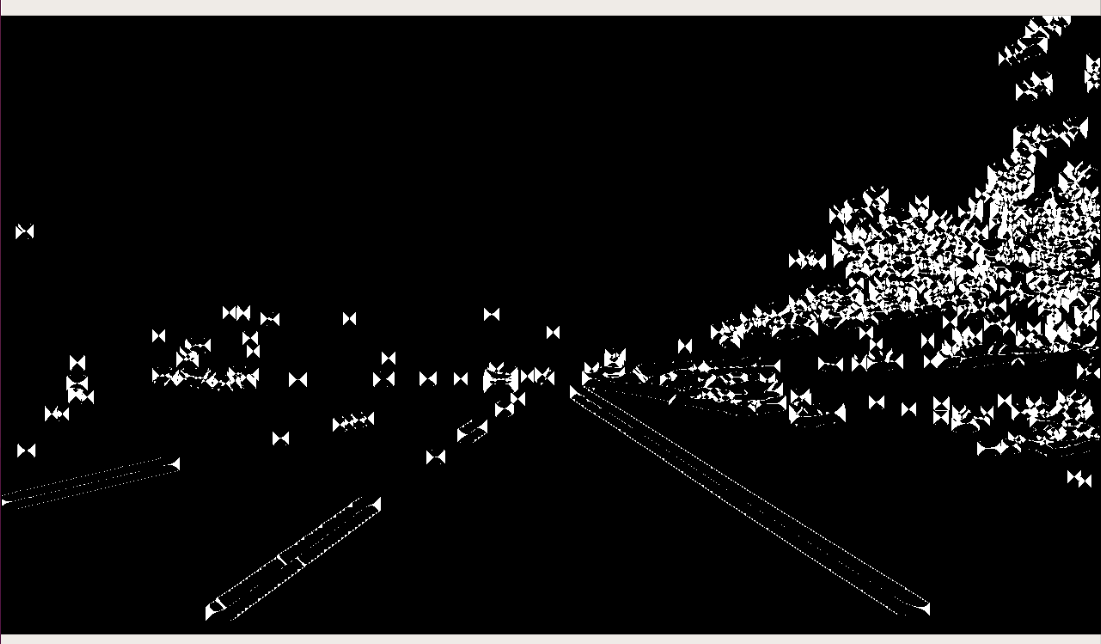
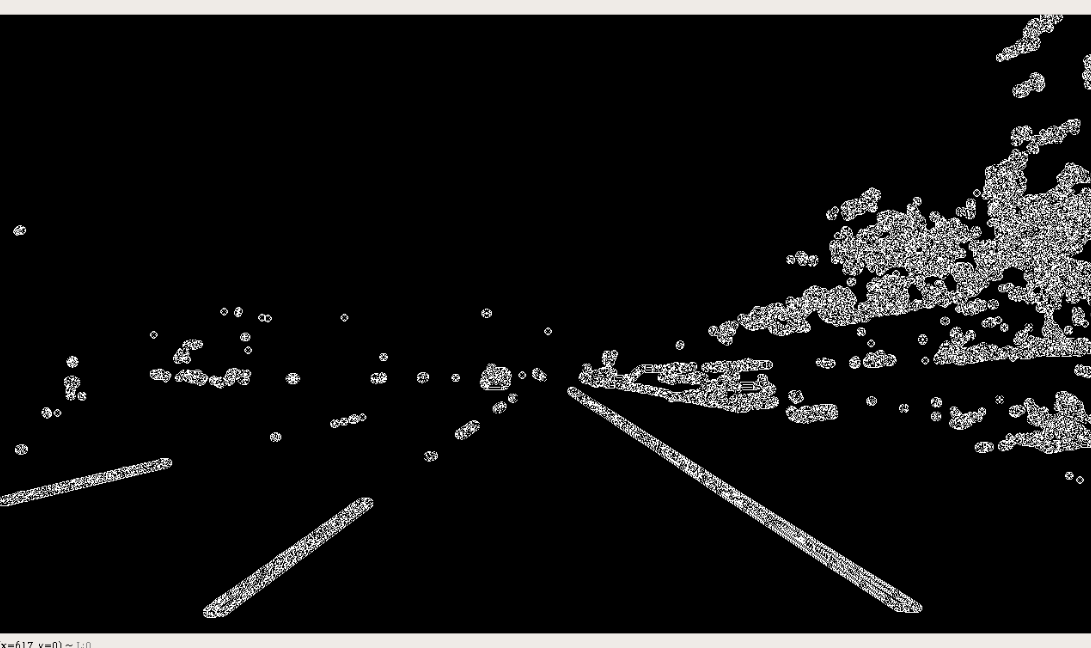


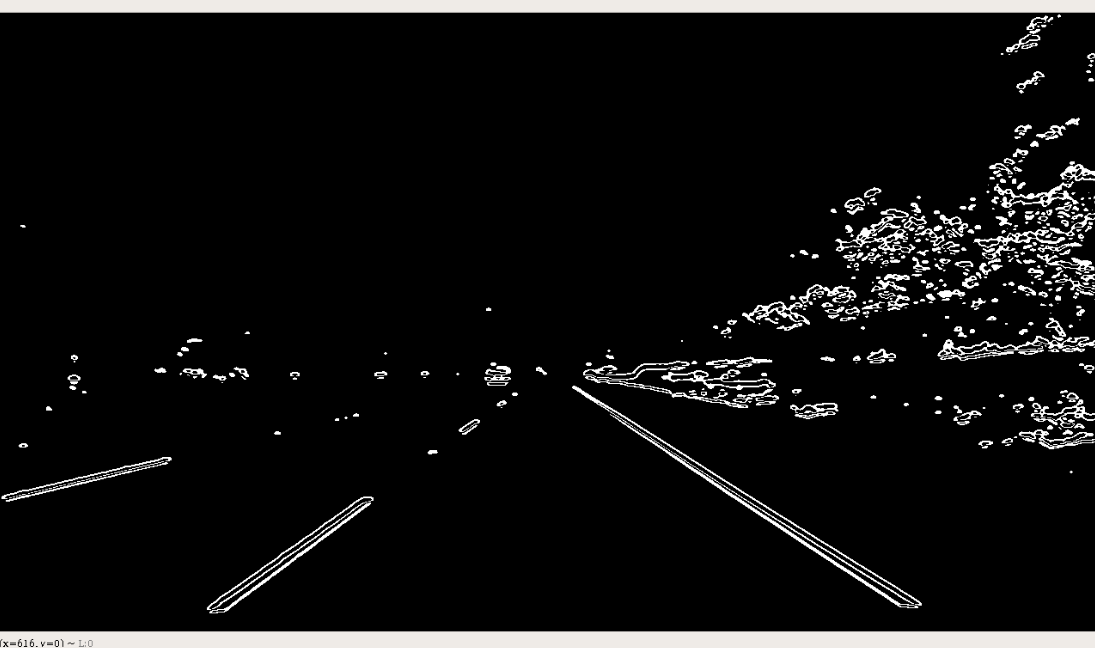
The angle of orientation of the edge gives rise to the spatial gradient. Theta is the direction of maximum contrast from black to white that runs from left to right on the image and angles are measured in anti-clockwise direction.

The function *mag\_sobel()* computes the magnitude of the gradient while *direction\_threshold()* finds the directional gradient and thresholds the image with respect to it.

OR and AND operators are used to find the combined result of gradient thresholding and outputs an image in which edges are successfully detected.

The results of gradient thresholding are shown in the following figure:

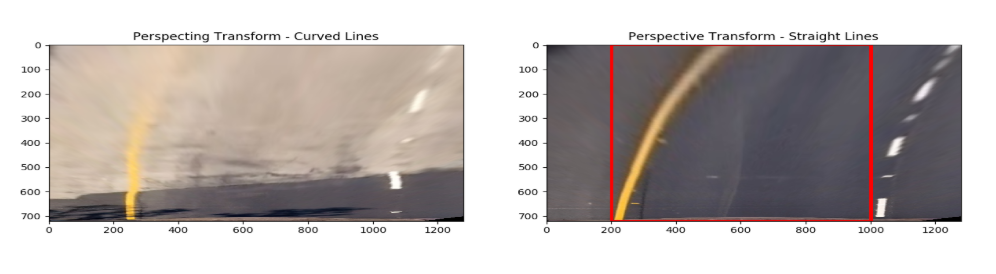
 



***Figure 5: 1st image : Sobel-directional gradient . 2nd image: Magnitude of gradient. 3rd image: Sobel-gradient in x-direction. 4th image: Sobel-gradient in y-direction. 5th image: Combined result of edge detection using sobel.***

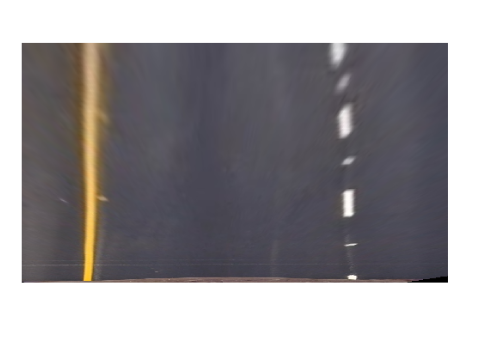
**Perspective Transform:**

The image with edges detected is then be perspective transformed. The transformation matrix can be derived by using pixel coordinates of points on the input image and the corresponding pixel coordinates of the perspective transformed image. A trapezoidal region is defined in the input image that goes through perspective transform to convert it into a rectangle seen from a bird’s eye view as shown in the following figure:



***Figure 6: The process of perspective transformation. A trapezoidal region is converted into a rectangle indicated by the red lines in the figure.***

The source (*src*) and destination (*dst*) points are given as parameters to the function *compute\_perspective\_transform\_matrices()* which uses OpenCV function *cv2.getPerspectiveTransform()* to find the transformation matrix and the inverse transformation matrix. The warped image is found using *cv2.warpPerspective()* in *perspective\_transform().*



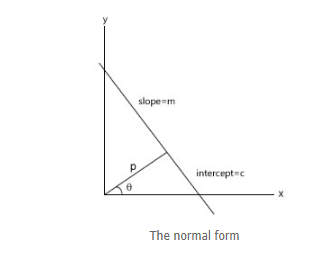
***Figure 7: The image in figure 6 after perspective transform.***

**Hough Transform:**

Hough Transform is then performed on the warped image from the previous step.

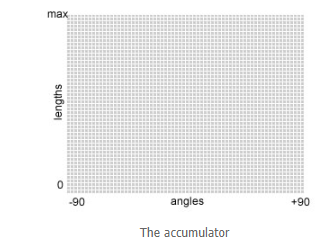
Edges detected in the image are just a sequence of pixels. In order to make lines using this sequence of pixels the technique of Hough Transform is used which performs explicit voting over a set of parameterized image objects.

A line is just a collection of points which is presented by the equation y = mx + c. In the m-c space the line represents a point and vice versa. Hough Transform uses this idea to convert points in the xy plane to lines in the mc plane. The interception points of these lines in mc plane give the parameters of the desired line in the xy plane. When lines are represented in their normal form in pθ space they form sinusoidal curves and vice versa. pθ space representation is shown in the following figure:



***Figure 8: The parameters of the pθ space.***

A 2D array called accumulator cells is formed where θ is placed on the x-axis and the p values on the y-axis. Pixels in the image vote for each bin. Sinusoidal curves are generated for each pixel and bright points (which receive votes by many pixels) on the curves correspond to parameters of the desired line on the original image.



***Figure 9: A 2D array called accumulator cells.***

The *cv2.HoughLines()* is used to implement Hough Transform. It returns an array of (\rho, \theta) values and takes in the binary image which has undergone edge detection. The binary image is first horizon masked that is it cropped in order to just keep the portion of the image in which lane edges are detected. Hough Transform is successfully performed on this binary image and *cv2.line()* is used to draw the lanes detected. The results of Hough Transform on the original image are shown in the following figure:

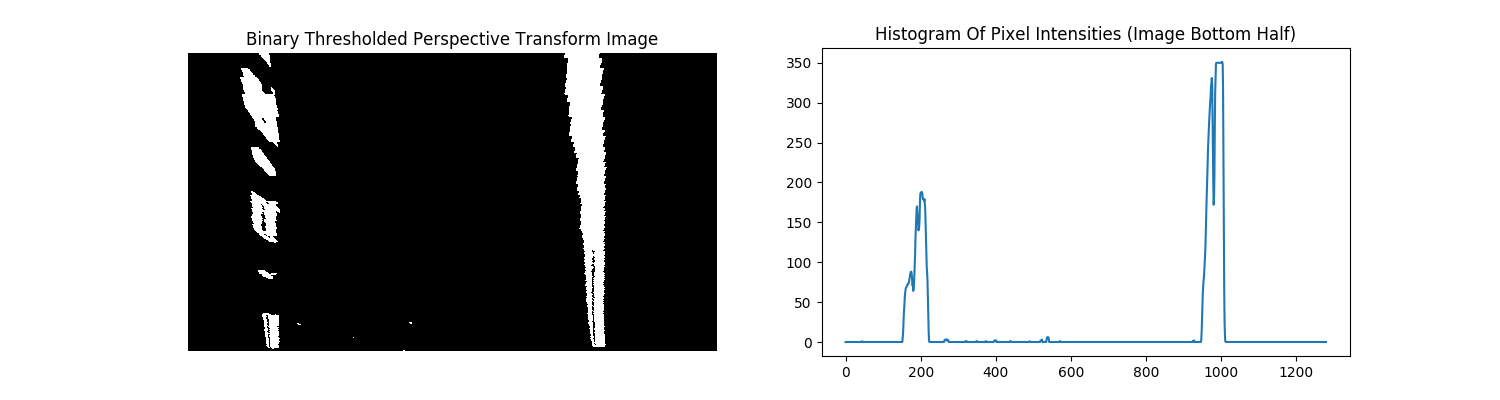


***Figure 10: The lane lines detected on the original road image through Hough Transform.***

However, this technique of Hough Transform to find and draw lane lines does not work with curved lane lines. For this purpose, another technique which is known as polynomial fitting is used.

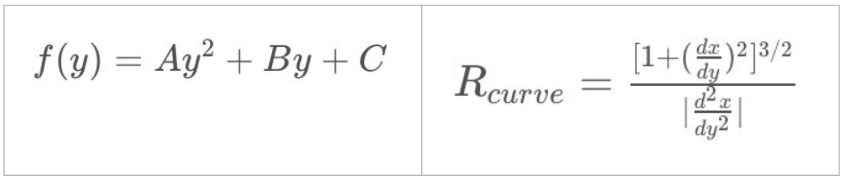
**Polynomial Fitting:**

In this algorithm a histogram is taken of the image where x-axis is the pixel position while the y-axis is the pixel count at that position. The peaks of the histogram indicate the approximate positions of the lane lines on the x-axis. From here on a sliding window algorithm is performed. A window of a fixed size is moved along the y-axis at the approximate positions found by the histogram on the x-axis. A weighted average is performed on the x values of the pixels in a window to find the base of the next window. The following figure shows the histogram generated:

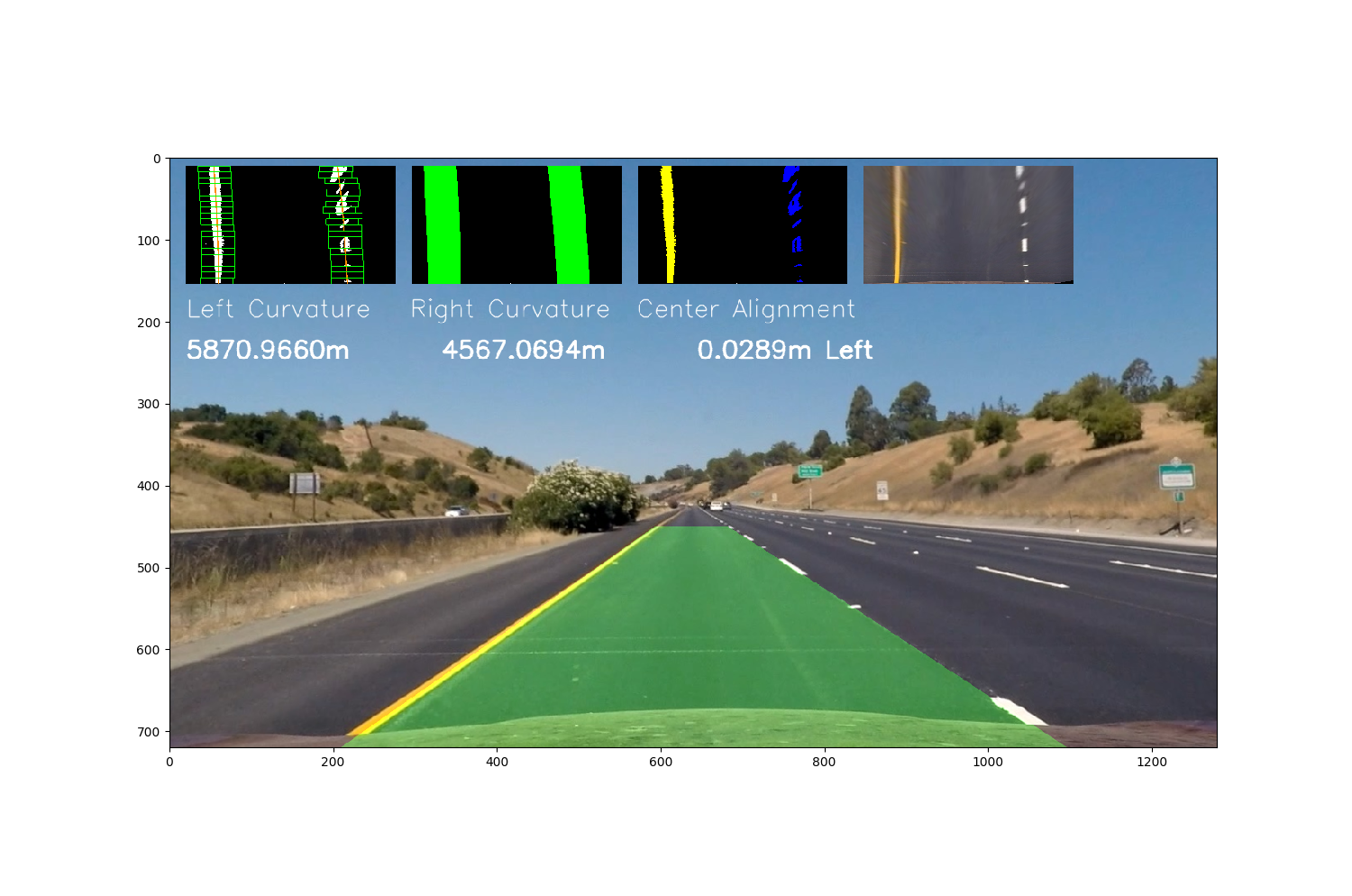


***Figure 11: The binary image after gradient thresholding and perspective transform (Left). The histogram of the image to the left where peaks show the approximate positions of the lanes (Right).***

Once sliding window algorithm is performed the *polyfit()* function is used to draw a second polynomial curve in order to detect the lanes. The radius of curvature is then figured out using the following formulae:



The function *process\_image()* performs this algorithm on the original image captured by the camera. It undistorts the image using the techniques and methods described previously and then does gradient thresholding on the image to find the edges in the binary image. The function *compute\_lane\_lines()* implements the window sliding algorithm and returns two arrays containing points for the left and right lanes. Using these points the lane curvature is found along with the center position. The function *draw\_lane\_line()* successfully draws the two lane lines. The final results of the lanes detected are shown below:



***Figure 12: The lanes detected on the original image of the road after polynomial fitting.***

**Conclusion:**

In RGB color-space, the Red (R) and Green (G) colors independently were somewhat effective in picking out lane lines and better when combined with & operation.

The Saturation channel was very effective in picking out lane lines in various light conditions. However, Saturation (S) channel’s drawback is that it highlights too many other features as well, like other cars and discolorations on the road.

Slicing out and applying thresholds to color-spaces is a relatively inexpensive operation computationally and helped reduce unwanted pixels at every stage in the program.

Furthermore, perspective transform naturally sheds regions outside the trapezoidal region eliminating the need for horizon masking.

The techniques used in the project work in predefined conditions i.e the lanes are yellow and white hence it would not work under conditions if the color of lane lines is different or if there are shadows on the road.

The results of the Hough Transform shows that numerous lines are drawn on the image so the threshold of the maximum votes for a point to be counted as a line in the accumulator could be increased to find lane lines accurately.

**Works Sited:**

Eddie Forson *,“Teaching Cars To See — Advanced Lane Detection Using Computer Vision” (* [*https://towardsdatascience.com/teaching-cars-to-see-advanced-lane-detection-using-computer-vision-87a01de0424f*](https://towardsdatascience.com/teaching-cars-to-see-advanced-lane-detection-using-computer-vision-87a01de0424f) *)*

Abdulhakam.AM.Assidiq, Othman O. Khalifa, Md. Rafiqul Islam, Sheroz Khan, *“Real time lane detection for autonomous vehicles” - Published in:* [*2008 International Conference on Computer and Communication Engineering*](https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4569830)

Samta Gupta, Susmita Ghosh Mazumdar*, “Sobel Edge Detection Algorithm”- Published in International Journal of Computer Science and Management Research- Feb 2013*

Utkarsh Sinha, *“The Hough Transform”* – AI Shack ( <http://aishack.in/tutorials/hough-transform-basics/> )