Advance Lane Detection Project

Abstract

In the growing domain of the autonomous driving, it is essential to know the lane in which the vehicle is driving, and which path is to follow on the roads. This lane line finding techniques is a simple way of approach to detect lines using the image processing technique on videos. The algorithm uses different ways of processing the image at first and using the same as reference for the continuous image processing on video.

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.
- Wrap 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.

I have created two Python notebooks for my project

- ALDP_Study → here I used it for my study purposes also I could take some output images for my write up. Where I can proceed step by step techniques.
- AdvancedLaneDetectionVideoProject → This is the finalized work where all the functions are well organized and called in final pipeline for the video processing

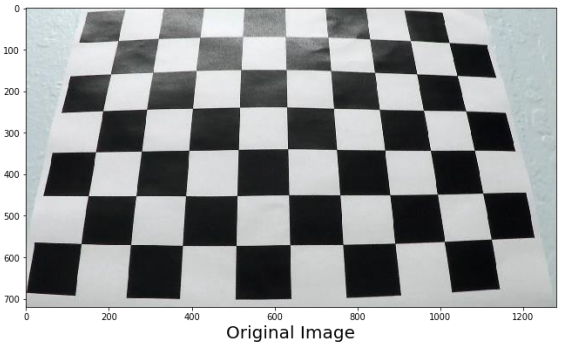
STEP 1 - Camera Calibration

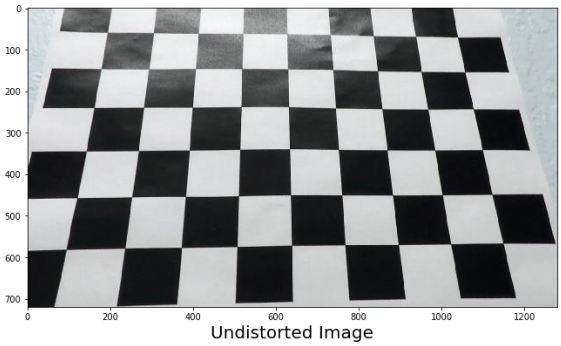
As different cameras used different lenses to form an image, there are possibilities that some of the image created by the cameras are often distorted. This distortion must be corrected before proceeding to the image processing a line detection.

To calibrate the camera, we will be using certain chess board images taken in different angles. We will now create the 'object points', which will be the (x,y,z) coordinates of the chessboard corners in the world. We assume the chessboard is fixed on the (x,y) plane at z=0, such that the object points are the same for each calibration image. Thus, **objp** is just a replicated array of coordinates, and **objpoints** will be appended with a copy of it every time we successfully detect all chessboard corners in a test image. **imgpoints** will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. We then will use the output **objpoints** and **imgpoints** to compute the camera calibration and distortion coefficients using the **cv2.calibrateCamera()** function.

STEP 2 – Distortion Correction

The computed Camera matrix and distortion coefficients is used for the distortion correction to the test image using the **cv2.undistort()** function and obtained this result





Lets try the camera calibration and undistortion function on an road image with lanes in the **"test_images"** folder and the below is the output image.

STEP 3 – Create thresholded binary image

We will now use certain methods to create some thresholded binary image and create some function that returns the binary image after filtering.

1) Gradient Threshold

- Sobel Operator

  Sobel operator is a way of taking the derivative of the image in the x or y direction. The operators for **Sobelx** and **Sobely** respectively, look like this:

  $$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

  $$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

  , this is of kernel size 3.

  After converting the raw image to gray, we need to pass the gray image to **cv2.Sobel()**.

- Magnitude of the Gradient

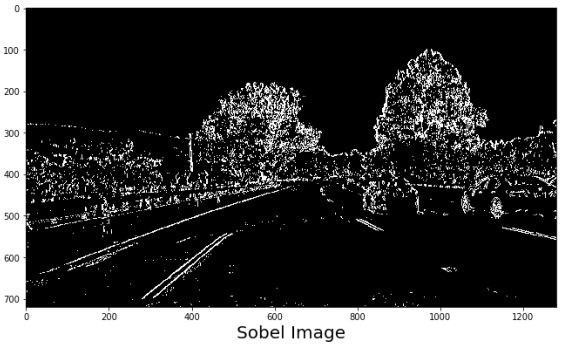
  The magnitude, or absolute value, of the gradient is just the square root of the squares of the individual x and y gradients. For a gradient in both the xx and yy directions, the magnitude is the square root of the sum of the squares.

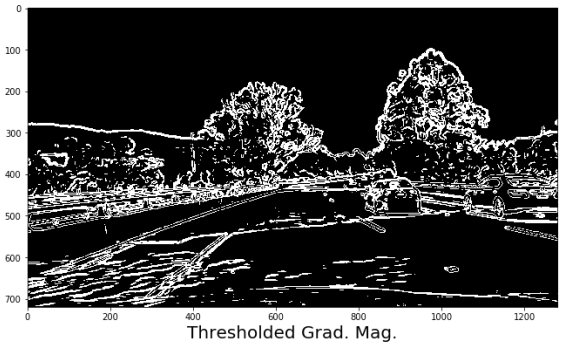
- Direction of the Gradient

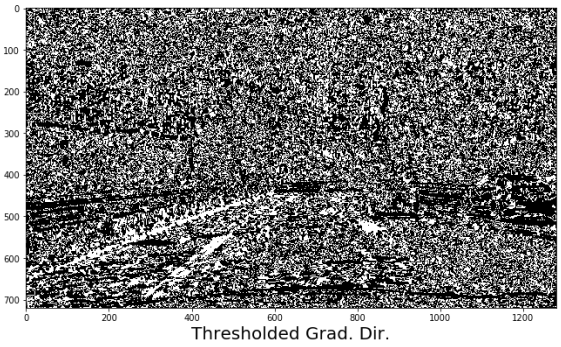
  The direction of the gradient is simply the inverse tangent (arctangent) of the yy gradient divided by the xx gradient:

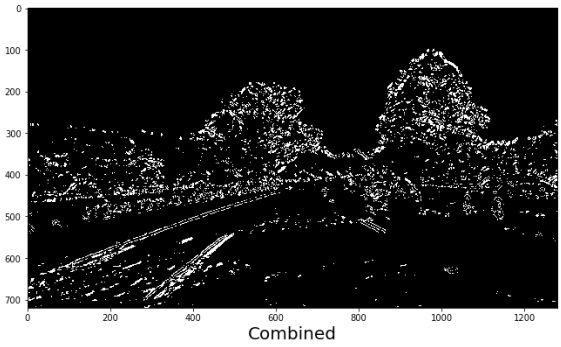
  arctan{(sobel_y/sobel_x)}arctan(sobely/sobelx).

  Each pixel of the resulting image contains a value for the angle of the gradient away from horizontal in units of radians, covering a range of -\pi/2−π/2 to \pi/2π/2.







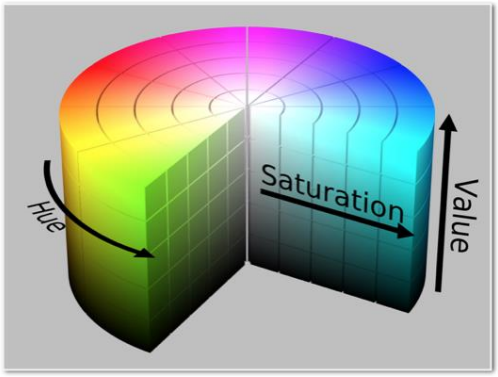


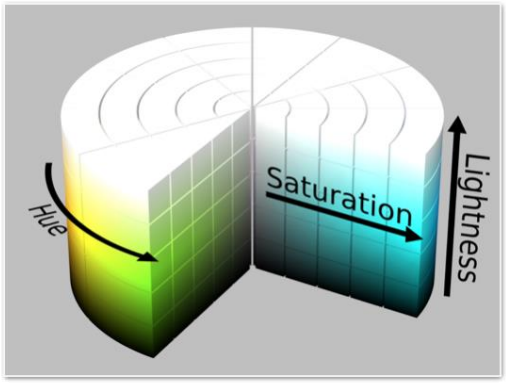
2) Color Thresholding

A color space is a specific organization of colors; color spaces provide a way to categorize colors and represent them in digital images.

RGB is red-green-blue color space. You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values. For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.

There is also HSV color space (hue, saturation, and value), and HLS space (hue, lightness, and saturation). These are some of the most commonly used color spaces in image analysis.





We will also be looking into LAB color space and LUV color space in order to check if the brightness and change in light is detected and filtered.

Here are some examples of the color thresholding and their respective binary image.

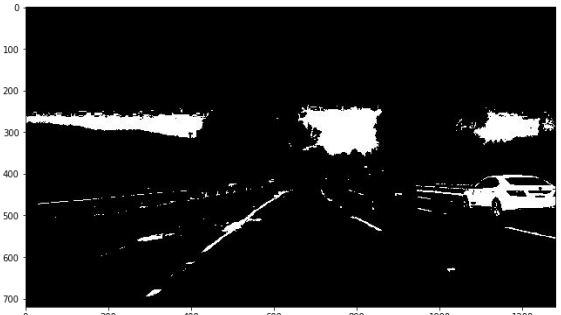
Gray Image

Gray binary Image

rgb R Image

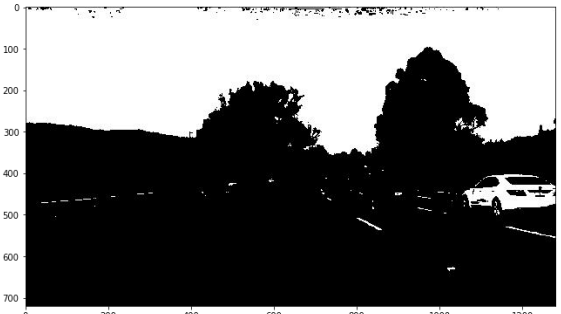
rgb R binary Image

rgb G Image



rgb G binary Image

rgb B Image



rgb B binary Image

hls H Image

hls H binary Image

hls L Image

hls L binary Image

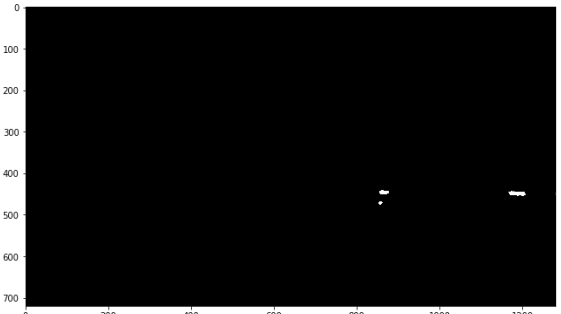
hls S Image

hls S binary Image

lab L Image

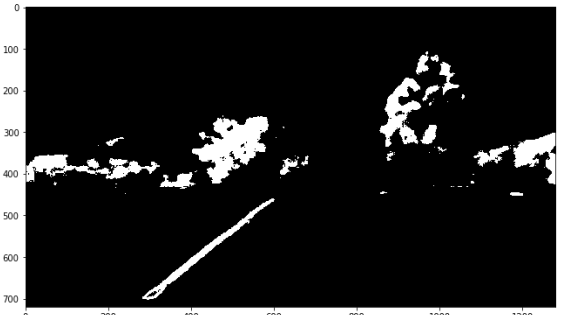
lab L binary Image

lab A Image



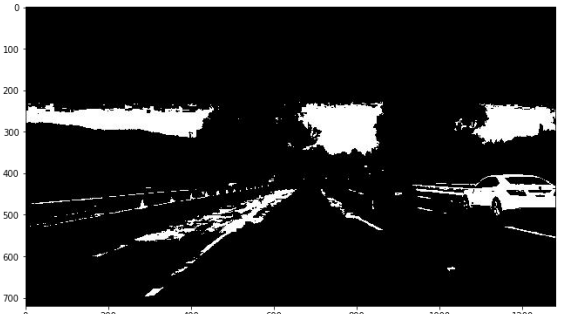
lab A binary Image

lab B Image

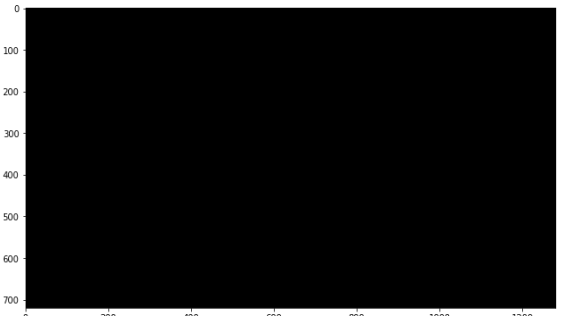


lab B binary Image

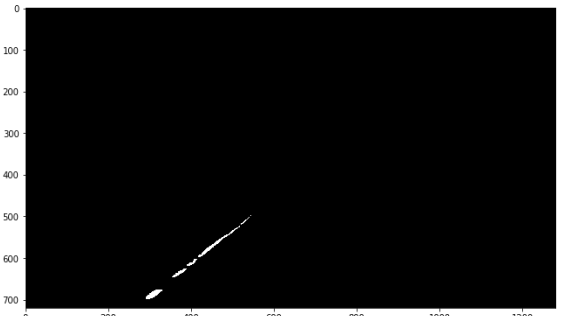

luv L Image


luv L binary Image


luv U Image

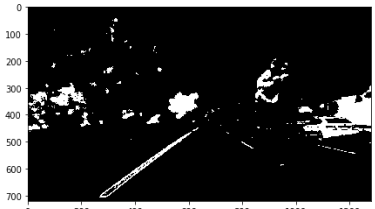

luv U binary Image

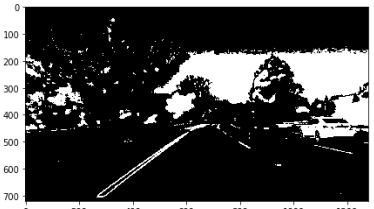

luv V Image

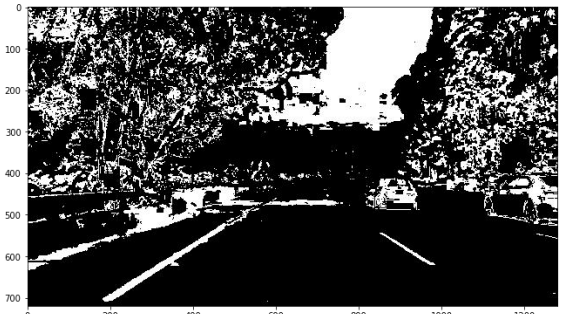

luv V binary Image

Now lets try combining some color threshold and gradient together and test on the images to check which combination is fitting to our expectation. We will use a test function called **color_gradient_pipetest()**.


Original Image

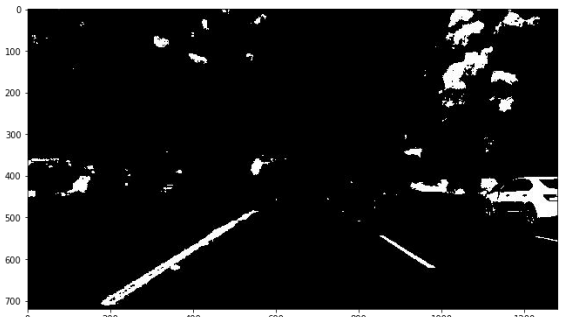

Lab_B,luv_L processed Image


Lab_B,hls_L processed Image



It looks like the combination of B in LAB and L in LUV provides us better filtered binary image which we can use for further calculation of the lanes.

We will now create a function **color_gradient_pipe()** which will combine the B in LAB and L in LUV. Below is the output of the combined image.



This way we ignore the sudden change in road color and the shadows. But still not very perfect when it comes to really strange roads and bright images.

STEP 4 – Perspective Transform

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform you'll be most interested in is a bird's-eye view transform that let's us view a lane from above; this will be useful for calculating the lane curvature later on. Aside from creating a bird's eye view representation of an image, a perspective transform can also be used for all kinds of different view points.

We will define the Source and Destination points of the image that has to be converted like below:

```
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]])
```

We will first compute the Perspective Transfrom M, providing the Src and Dst points using the following method

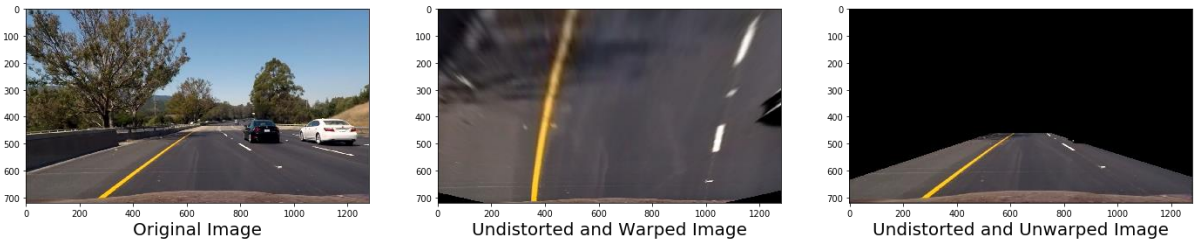
M = cv2.getPerspectiveTransform(src, dst)

And at the same time compute the inverse perspective transform Minv suing the following method

Minv = cv2.getPerspectiveTransform(dst, src)

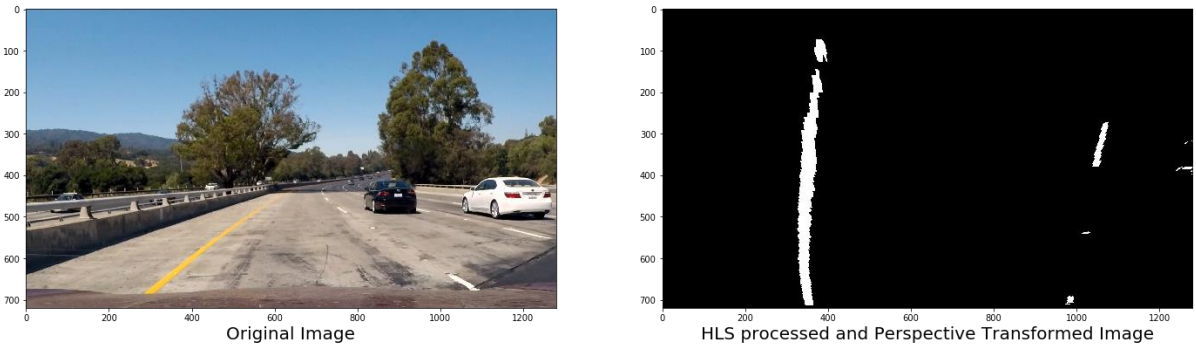
We will create two functions that will return the perspective transform and inverse perspective transform which are **perspective_transform()** and **inv_perspective_transform()**. And using the **cv2.warpPerspective()** we will warp an image.

Here is an example:



We will now create a pipeline which performs all the above discussed steps called as **Image_process_pipeline()** and returns us the Warped image.

Here is an example:



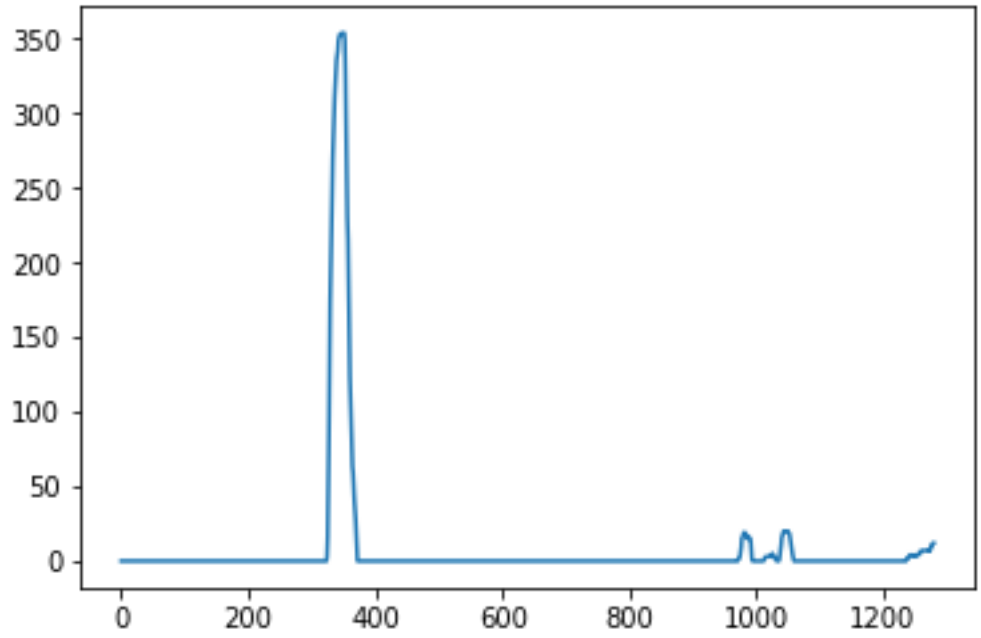
STEP 5 - Detect lane pixels and fit to find the lane boundary

Now we proceed further to the important steps of detecting the lane pixels and fit the lane boundary.

Peaks in a Histogram: After applying calibration, thresholding, and a perspective transform to a road image, we will have a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

Plotting a histogram of where the binary activations occur across the image is one potential solution for this.

Here is an example of the histograms peaks of the above example image.

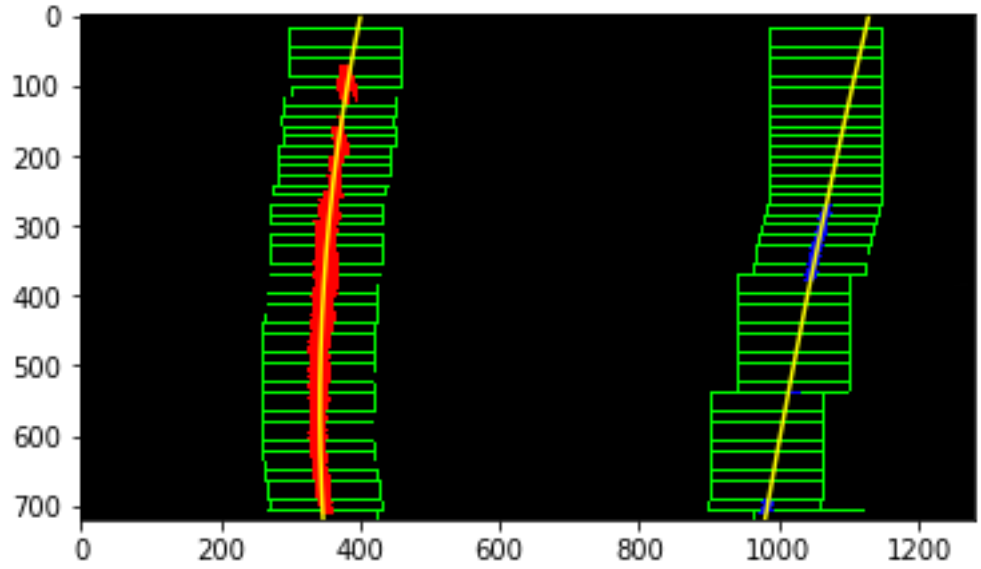


We will be using the two highest peaks from our histogram as a starting point for determining where the lane lines are, and then using the sliding windows moving upward in the image to determine where the lane lines go.

We have to split the windows to 2, one for left lane and another for right. We also have to set up some hyperparameters such as sliding windows **(nwindows = 50)**, width of the windows **(margin = 80)**, minimum number of pixels found to recenter windows **(minpix = 50)**. The function **find_lane_pixels()** is made which iterates through the nwindows and calculate the **leftx_current** and **rightx_current**.

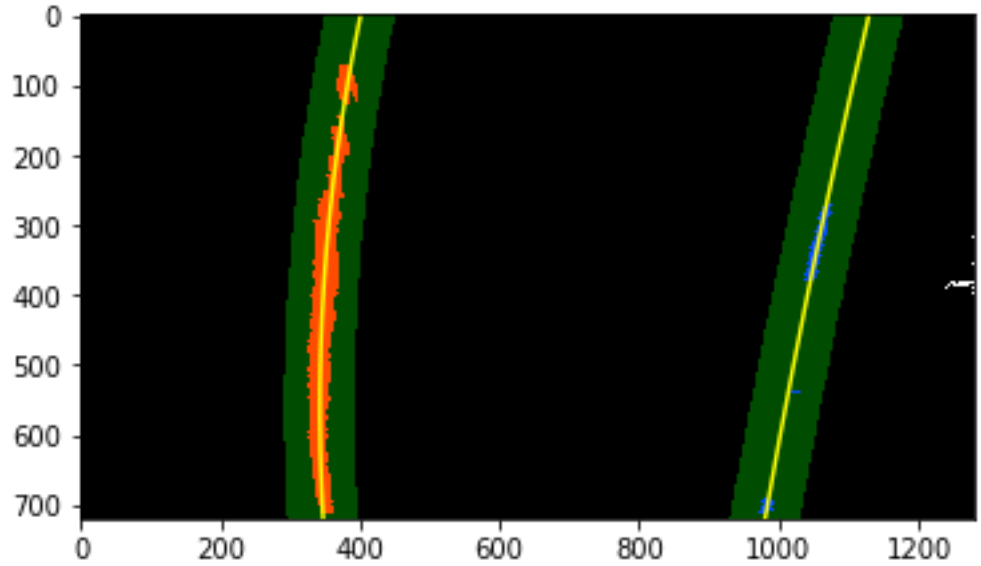
We will now separate the nonzeroy and nonzerox based on the activated pixels using the minpix value and append these to our lists **left_lane_inds** and **right_lane_inds.** We will also create a function **fit_polynomial()** which will be used to fit a polynomial to the line.

And the out put looks something like this.

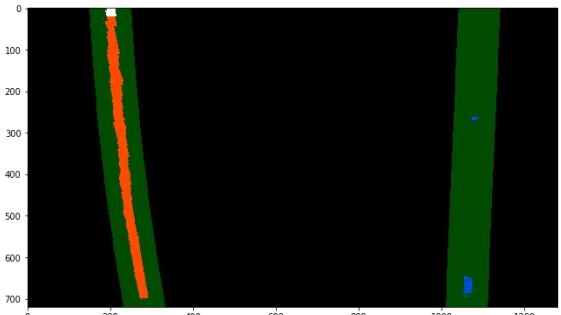


Now that we have some lines found, now we will create another function **search_around_poly ()** which will take the existing lane lines and search around on in those regions for the pixels so that there is no need of a blind search again.

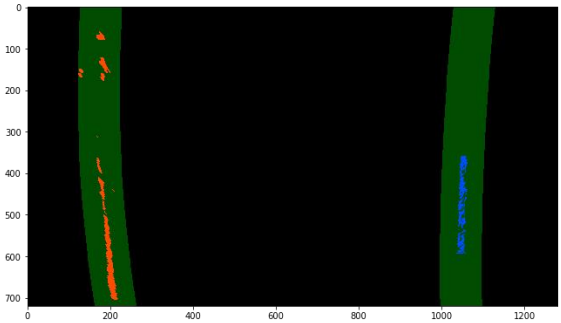
The output looks something like this:



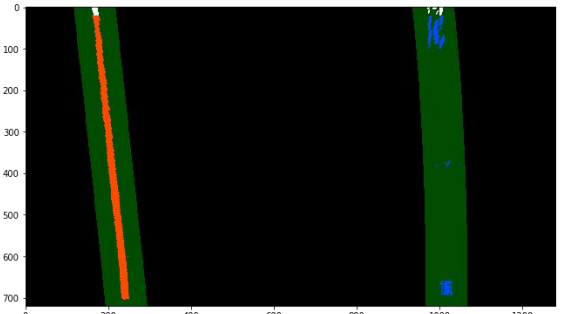

Original Image


Lane detected Image

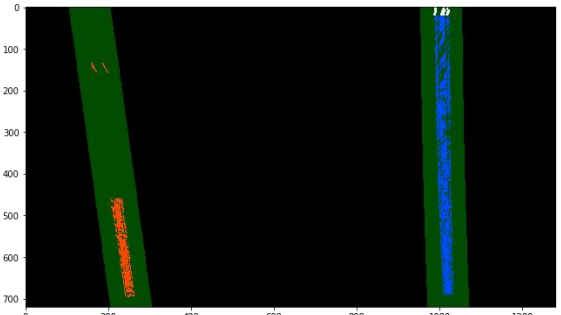

Original Image


Lane detected Image

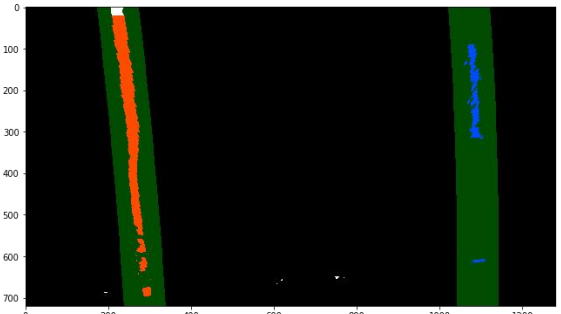

Original Image


Lane detected Image

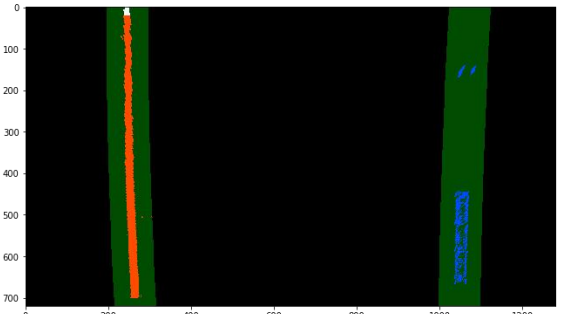

Original Image


Lane detected Image

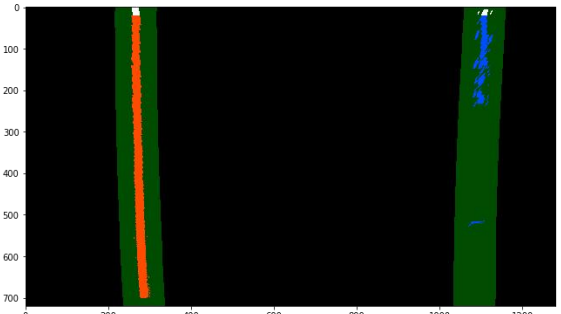

Original Image

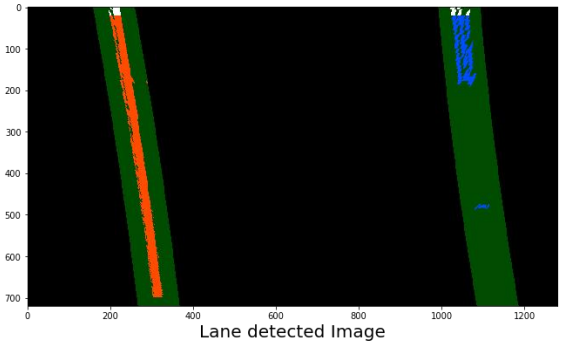

Lane detected Image


Original Image

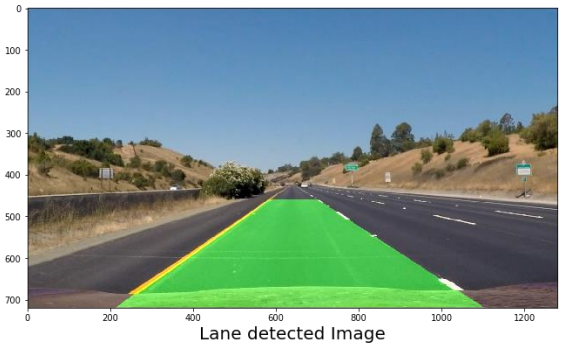

Lane detected Image


Original Image

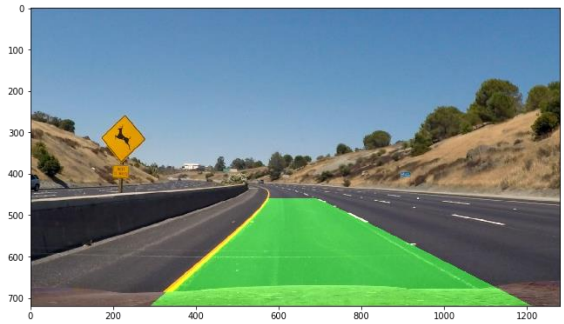

Lane detected Image



Now lets create a pipeline **LaneFitPipeline()** which does every thing we discussed earlier and returns us the unwarped image and we will combine the lane lines to the raw image using **cv2.addWeighted()**.

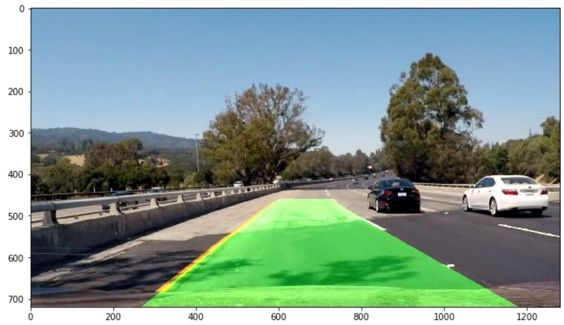


Original Image



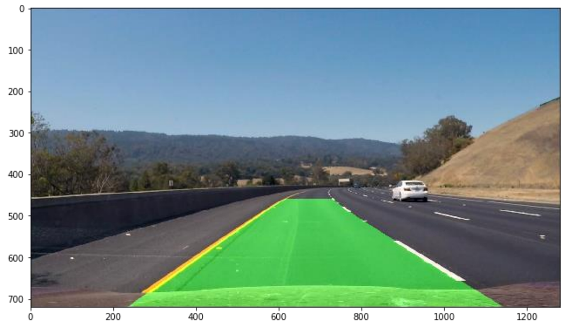
Lane detected Image

Original Image



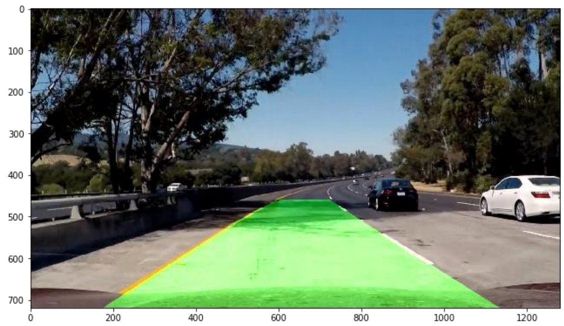
Lane detected Image

Original Image



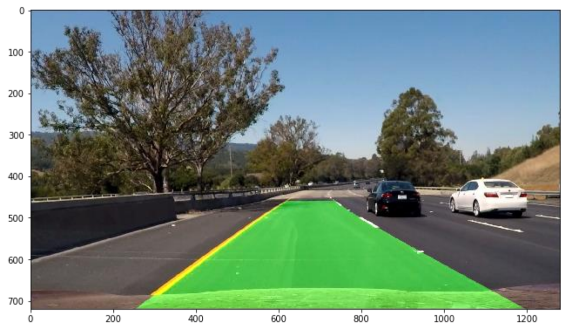
Lane detected Image

Original Image



Lane detected Image

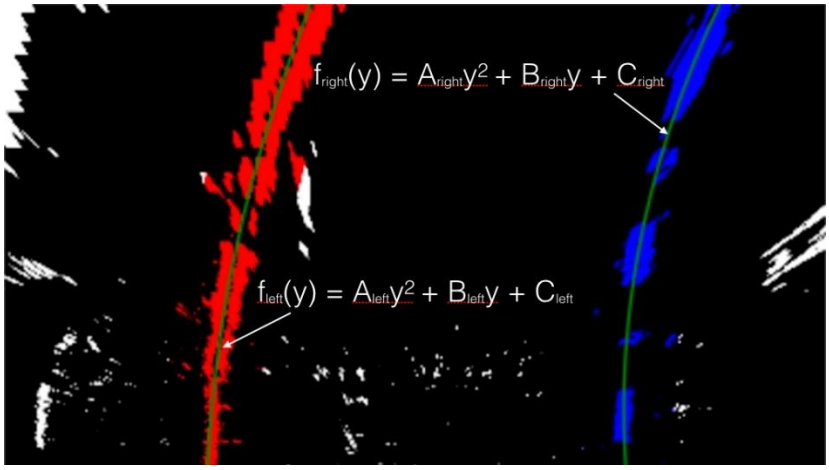
Original Image



Lane detected Image

STEP 6 – Curvature calculation

Proceeding further with the lane detection, it is important to calculate the radius of curvature of the fit.



In order to calculate the radius of curvature we will use the following equation:

$Rcurve = ((1+(2Ay+B)^2)^{3/2}) / |2A|$

Also it is important to note that the calculated curvature is in pixels and we need the information in meters in the real world.

We will define conversions in x and y from pixels space to meters

ym_per_pix = 30/720 # meters per pixel in y dimension

xm_per_pix = 3.7/700 # meters per pixel in x dimension

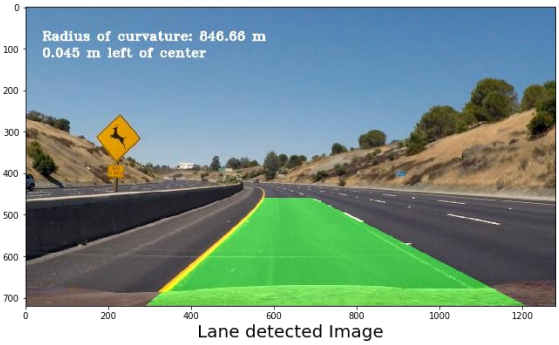
For this calculaltion of radius of curvature we will create a function called **Calc_Curv()** which will also add the text together with the final lane detected image.

STEP 7 – Final pipeline

We will now create a final pipeline which performs all the action discussed until now and provides us the final image together with the calculation of the curvature.

Below is one example:


Original Image



Lane detected Image

STEP 7 – Sanity Checks

We will also create a class called Line(), which we will be using to keep track of all the important fits and parameters. We will create two instances, one for left lane and another for right lane. We will use this class to choose the find lane function or search around function based on the sanity checks we do. In this case, we will check if the lanes are detected or not based on the length of the arrays that are passed over to fit poly functions. In case some better fits are found continuously for 4 times, we will save those values in our self.bestfits array using the Line() class.

I have created another Python notebook where all the relevant functions are organized orderly to process a video which is named, **AdvancedLaneDetectionVideoProject.ipynb**.

Please find the output videos in the folder 'test_output_images/'

Discussion:

At the end of the advance lane detection, we learn how to use different techniques used in computer vision for the detection of lanes. There are several situations where the lanes could not be detected or miss-assumptions of some other object like cars or shadows of trees or some curvy roads which cannot work using these techniques. There are lot more to explore in the world of self driving car. One can tune even better the thresholds to have better binary images. More important things should be considered in our case to have a better output result.