PROJECT TITLE: - VEHICLE LICENSE PLATE RECOGNITION USING PYTHON AND OPEN CV

February 3, 2021

Table of Contents

INTRODUCTION

License Plate Recognition is a Digital image-processing technology used to identify vehicles by their license plates. This technology is used in various security and traffic applications.

Digital Image processing (DIP) refers to process real world images digitally by a computer. It is a broad topic, which includes studies in physics, mathematics, electrical engineering, and computer science. It studies the conceptual foundations and deployment of images and in detail the theoretical and algorithmic processing. It also aims to improve the appearance of the images and make them more evident in certain details that you want to note.
DIP Processes the capture is the process through which a digital image is obtained using a capture device like a digital camera, video camera, scanner, satellite, etc...

The preprocessing includes techniques such as noise reduction, contrast enhancement, enhancement of certain details, or features of the image. Some processes include:-

The segmentation is the process which divides an image into objects that are of interest to our study.

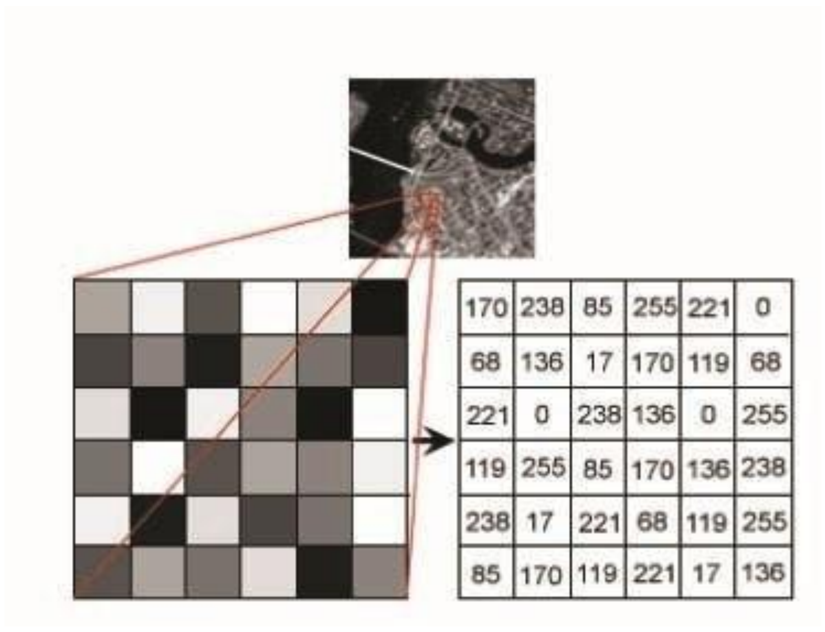
The recognition identifies the objects, for example, a key, a screw, money, car, etc...

The interpretation is the process that associates a meaning to a set of recognized objects (keys, screws, tools, etc...) and tries to emulate cognition.

Theoretical background

What is an image: An image is an array, or a matrix, of square pixels (picture elements) arranged in columns and rows. . An image is represented as a dimensional function F(x,y), where x and y are spatial coordinates, and the amplitude of F at any pair of coordinates (x,y) is called the intensity of that image at that point. When x,y , and amplitude values of F are finite, we call it a **digital image**.

Each pixel also has its own value. For a grayscale image, each pixel would have an intensity between 0 and 255, with 0 being black and 255 being white. f(x,y) would then give the intensity of the image at pixel position (x,y), assuming it is defined over a rectangle, with a finite range: f: [a,b] x [c,d] → [0, 255].



170	238	85	255	221	0
68	136	17	170	119	68
221	0	238	136	0	255
119	255	85	170	136	238
238	17	221	68	119	255
85	170	119	221	17	136

Figure 1: grayscale image sample representation

A color image is just a simple extension of this. f(x,y) is now a vector of three values instead of one. Using an RGB image as an example, the colors are constructed from a combination of Red, Green, and Blue (RGB). Therefore, each pixel of the image has three channels and is represented as a 1x3 vector. Since the three colors have integer values from 0 to 255, there are a total of 256*256*256 = 16,777,216 combinations or color choices.

$$f(x, y) = \begin{pmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \end{pmatrix}$$

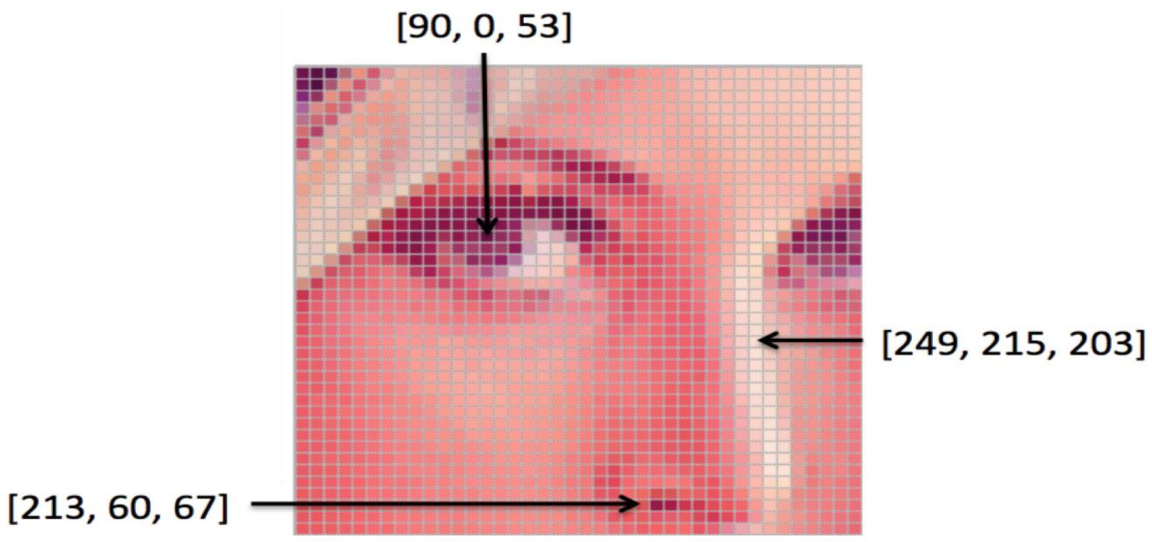


Figure 1: color image sample representation

Convert Colour Space

There are a number of commonly used methods to convert an RGB image to a grayscale image such as average method and weighted method.

Average Method

The Average method takes the average value of R, G, and B as the grayscale value.

Grayscale = (R + G + B) / 3.

Theoretically, the formula is 100% correct. But when writing code, we may encounter uint8 overflow error — the sum of R, G, and B is greater than 255. To avoid the exception, R, G, and B should be calculated respectively.

Grayscale = R / 3 + G / 3 + B / 3.

The average method is simple but doesn't work as well as expected. The reason being that human eyeballs react differently to RGB. Eyes are most sensitive to green light, less sensitive to red light, and the least sensitive to blue light. Therefore, the three colors should have different weights in the distribution. That brings us to the weighted method.

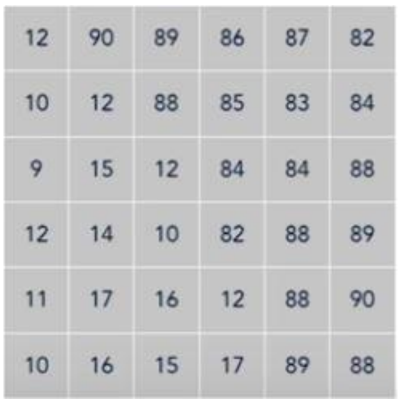
The Weighted Method

The weighted method, also called luminosity method, weighs red, green and blue according to their wavelengths. The improved formula is as follows:

Grayscale = 0.299R + 0.587G + 0.114B

Edge Detection

Edges are significant local changes of intensity in a digital image. An edge can be defined as a set of connected pixels that forms a boundary between two disjoint regions.



12	90	89	86	87	82
10	12	88	85	83	84
9	15	12	84	84	88
12	14	10	82	88	89
11	17	16	12	88	90
10	16	15	17	89	88

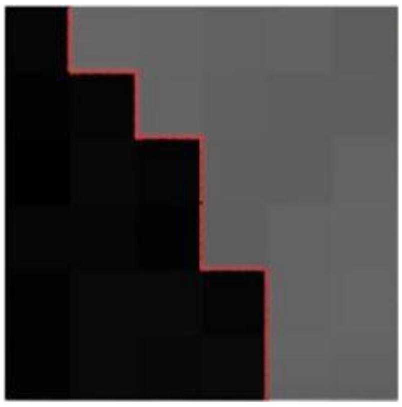


Figure 3: example of gray scale image with some intensity values

Edge Detection is a method of segmenting an image into regions of discontinuity. It is a widely used technique in digital image

Edge detection allows users to observe the features of an image for a significant change in the gray level. This texture indicating the end of one region in the image and the beginning of another. It reduces the amount of data in an image and preserves the structural properties of an image. Some edge detection algorithms are:

Sobel Detection Algorithm

The Sobel algorithm was developed by Irwin Sobel and Gary Feldman at the Stanford Artificial Intelligence Laboratory (SAIL) in 1968.

In the simplest terms, their algorithm works by running a 3x3 matrix (known as the kernel) over all the pixels in the image. At every iteration, we measure the change in the gradient of the pixels that fall within this 3x3 kernel. The greater the change in pixel intensity, the more significant the edge there is.

The algorithm relies heavily on a mathematical operation called convolution.

Convolution is simply the process of multiplying and adding corresponding indices of two matrices together (the kernel and the image matrix) after flipping both the rows and the columns of the kernel.

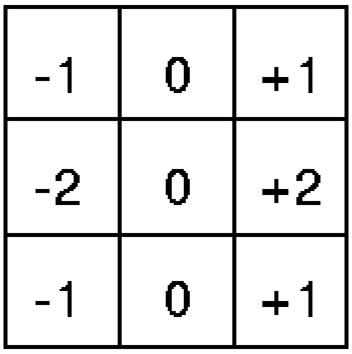
$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

The kernel on the left has its rows and columns flipped and then multiplied by the corresponding indices in the image matrix.

The standard implementation of the algorithm uses the following 3x3 matrices for the *x* and *y* axis, respectively.

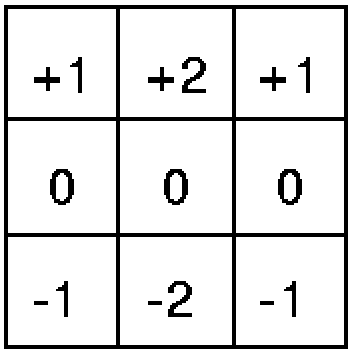
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Pre-Convoluted Kernel (A is a matrix containing the image data)



-1	0	+1
-2	0	+2
-1	0	+1

Gx



+1	+2	+1
0	0	0
-1	-2	-1

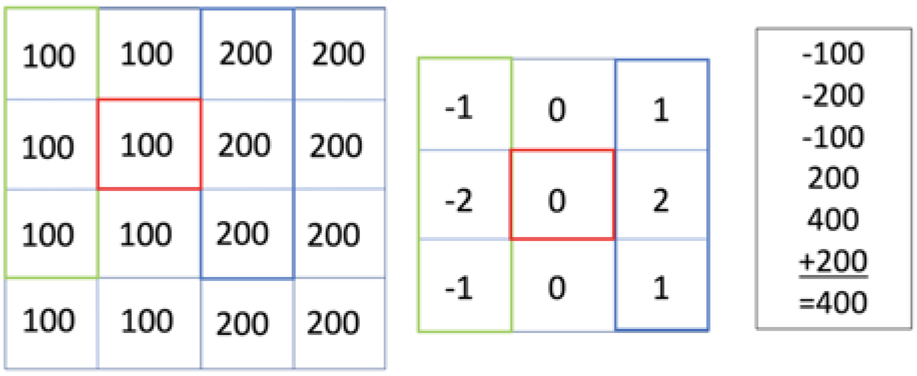
Gy

Convoluted version of the kernel above

Now, we have completed all of the pre-processing and have the convoluted matrices in hand.

We can iterate through all pixels in the original image and apply the image convolution kernels Gx and Gy at every step of the way. This convolution operation will return a single integer for each kernel.

Here's an example of what applying the Gx kernel on a specific pixel (in red) would look like:



100	100	200	200
100	100	200	200
100	100	200	200
100	100	200	200

-1	0	1
-2	0	2
-1	0	1

```
-100
-200
-100
200
400
+200
=400
```

Kernel Convolution: The bigger the value at the end, the more noticeable the edge will be.

On the left is the original image with the 3x3 pixel group colored

The penultimate step is to create a new image of the same dimensions as the original image and store for the pixel data, the magnitude of the two gradient values:

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Canny Operator

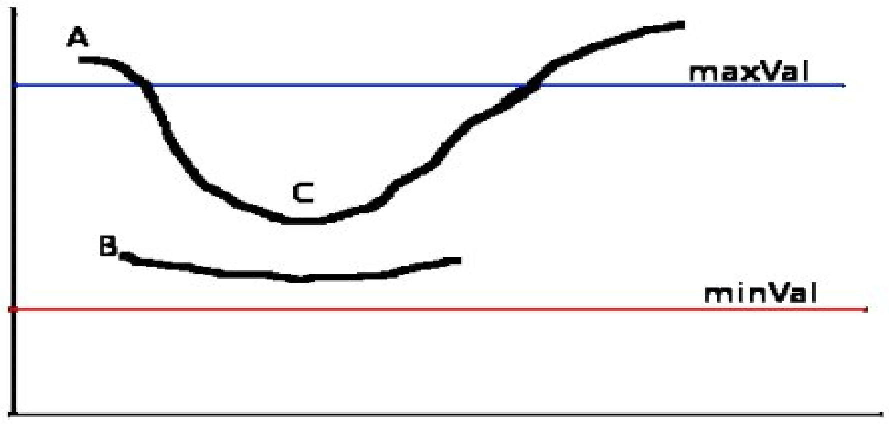
It is a gaussian-based operator in detecting edges. Canny edge detector have advanced algorithm derived from the previous work of Laplacian of Gaussian operator. It is widely used an optimal edge detection technique. It detects edges based on

Non-maximum Suppression

It takes Sobel operator output as input and then, at every pixel is checked if it is a local maximum in its neighborhood in the direction of gradient the result you get is a binary image with thin edges

Hysteresis Thresholding

This stage decides which are all edges are really edges and which are not. For this, we need two threshold values, *minVal* and *maxVal*. Any edges with intensity gradient more than *maxVal* are sure to be edges and those below *minVal* are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to “sure-edge” pixels, they are considered to be part of edges. Otherwise, they are also discarded. See the image below:



The edge A is above the *maxVal*, so considered as “sure-edge”. Although edge C is below *maxVal*, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above *minVal* and is in same region as that of edge C, it is not connected to any “sure-edge”, so that is discarded. So it is very important that we have to select *minVal* and *maxVal* accordingly to get the correct result.This stage also removes small pixels noises on the assumption that edges are long lines. So what we finally get is strong edges in the image.

Advantages:

It has good localization, It extract image features without altering the features

Less Sensitive to noise

Limitations:

Complex computation and time consuming

Problem statement

As the number of traffic on roads increasing constantly, the manual process in car license plate recognition becomes a serious problem for traffic management systems which not only detects and tracks a vehicle but also identify it. Manual car license plate recognition is good with little effort even if the images may vary somewhat in different viewpoints, sizes, and braked letters. However, it is tedious process when doing it repeatedly and also it is difficult to access the manually recognized license plates by other applications such as in traffic enforcement to remember when a car with that license plate is coming, in parking stations to calculate parking fee for vehicles, and stolen car identification to search, filter, and compare with black list database.

License plate recognition is the extraction and recognition of vehicles from visual sources such as camera captured images and recorded video streams. License plate recognition is a challenging task due to the following conditions: Different lighting conditions present at the time of capturing the vehicle license plate image that can affect the quality of license plate recognition. The differences in style, size, orientation, and color of the license plate characters that will affect the quality of the license plate character recognition. Different fonts and formats make the plate detection recognition difficult. Varying lighting, camera distance angle, and format of the image poses problems for recognition.

Objective

General Objective

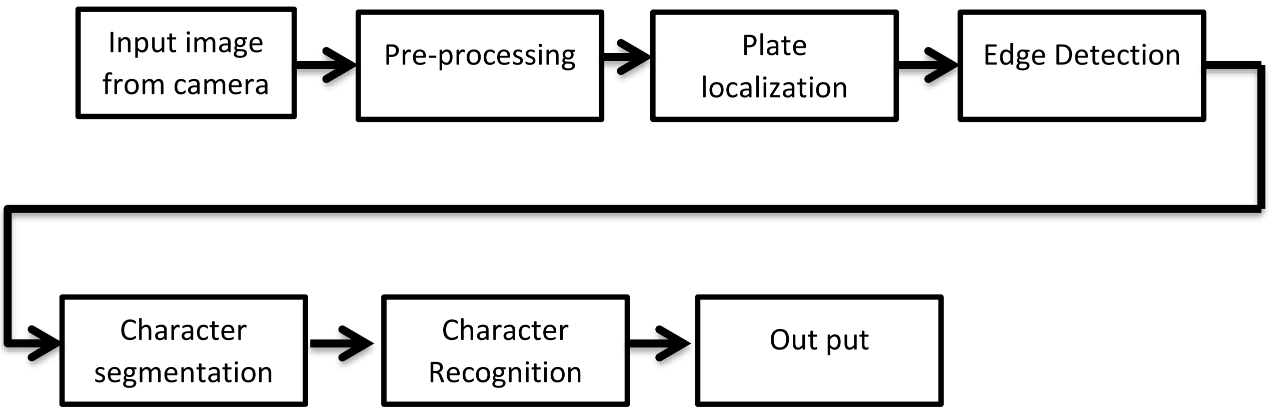
The main purpose of the project is to develop vehicles license plate recognition system that able to detect and recognize vehicles plates from visual sources using python, opencv and tesseract.

Specific Objectives

The specific objectives are to:
Detect or filter out only the plate from the input image
Detect and recognize plate characters in the detected plate

Methodology

OpenCV is the huge open-source library for the computer vision, machine learning, and image processing and now it plays a major role in real-time operation which is very important in today's systems. By using it, one can process images and videos to identify objects, faces, or even handwriting of a human. When it integrated with various libraries, such as Numpuy, python is capable of processing the OpenCV array structure for analysis. To Identify image pattern and its various features we use vector space and perform mathematical operations on these features.
Steps involved in License Plate Recognition



Input image from camera

The image of the vehicle is captured using a high resolution camera. Character recognition is generally very sensitive to the skew. The readable characters can become distorted due to the inclination of the camera. Using a better camera with more definition and resolution will increase the success ratio of the system.

Pre-process

Pre-processing is the set algorithms applied on the image to enhance the quality. It is an important and common phase in any computer vision system. For the present system preprocessing involves two processes:

Resize:-The image size from the camera might be large and with bigger resolution this can drive the system slow. It is to be resized to a feasible aspect ratio.

Convert Colour Space– Images captured using cameras will be either in raw format or encoded into some multimedia standards. Normally, these images will be in RGB mode, with three channels (red, green and blue).

Number of channels defines the amount colour information available on the image. The image has to be converted to grayscale which is common in all image processing steps this speeds up other following process since we no longer have to deal with the color details when processing an image.

Plate localization

Our image will have useful and useless information, in this case for us only the license plate is the useful information the rest are pretty much useless for our program. This useless information is called noise. Normally using a bilateral filter (Blurring) will remove the unwanted details from an image. The code for the same is blurred

Character Segmentation

Segment the license plate out of the image by cropping it and saving it as a new image. We can then use this image to detect the character in it. crop the roi (Region of interest) image form the main image. To find the region of interest in our case the rectangle box that holds the plate number we use contours

Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition.

Character Recognition

The Final step in this Number Plate Recognition is to actually read the number plate information from the segmented image. We will use the pytesseract package to read characters from image

Python-tesseract is an optical character recognition (OCR) tool for python. That is, it will recognize and “read” the text embedded in images. ... Additionally, if used as a script, Python-tesseract will print the recognized text instead of writing it to a file

OCR is a technology that converts printed text into a digital format. The variety of different fonts and ways of writing a single character makes this problem hard to solve.

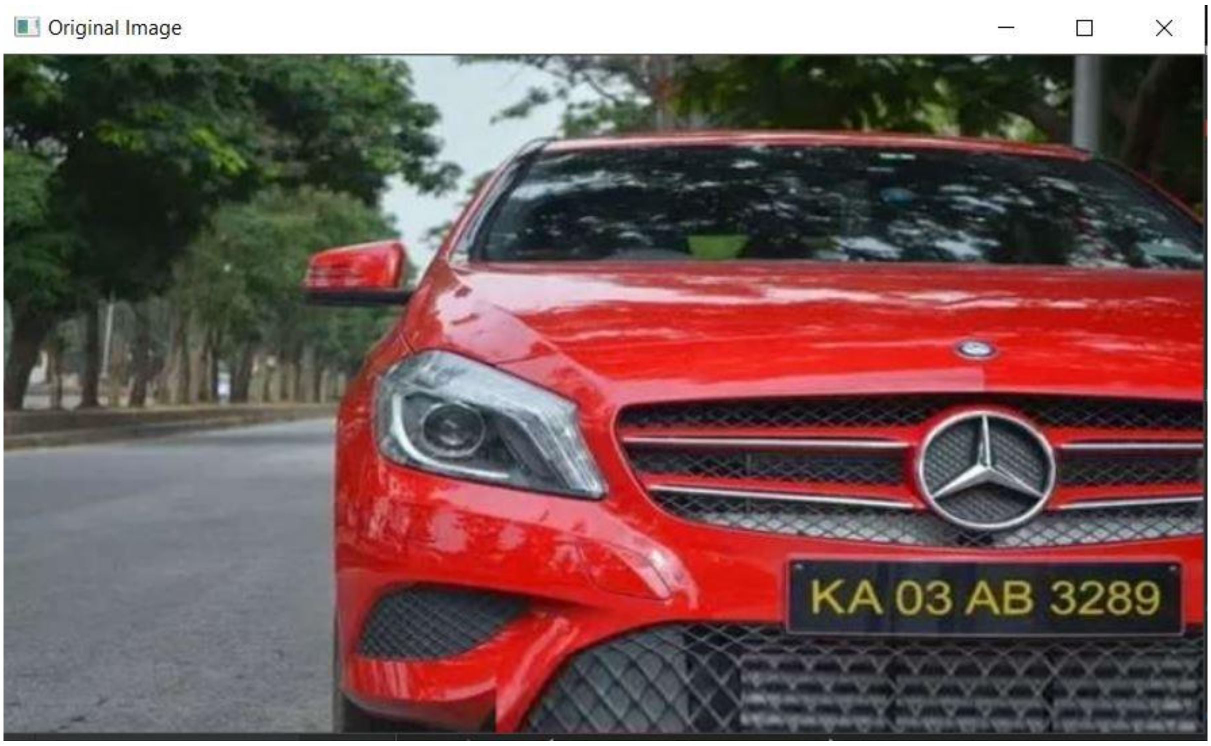
The approach of pattern recognition, works by identifying the character as a whole. We can identify a line of text by looking for rows of white pixels with rows of black pixels in between. In the same way, we can identify where an individual character begins and ends. Next, we convert the image of the character into a binary matrix where white pixels are 0s and black pixels are 1s.Then, by using the distance formula, we can find the distance from the center of the matrix to the farthest one. We then create a circle of that radius and split it up into more granular sections. At this point, the algorithm will compare every single subsection against a database of matrices representing characters with various fonts to find the character it statistically has the most in common with. Doing this for every line and every character makes it easy to bring printed media into the digital world..

Simulation and discussion

Input image from camera

Load an image using imread. The following code explains how to capture the image from camera and load source image into the system.

```
img = cv2.imread('4.jpg', cv2.IMREAD_COLOR)
cv2.imshow("Original Image", img)
```



Pre-process

As seen before, preprocessing involves changing the size using resize and color spaces of the source image Transform an image from BGR to Grayscale format by using cvtColor

```
img = cv2.resize(img, (620, 480))
cv2.imshow("Resized Image", img)

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow("Gray Image", gray)
```

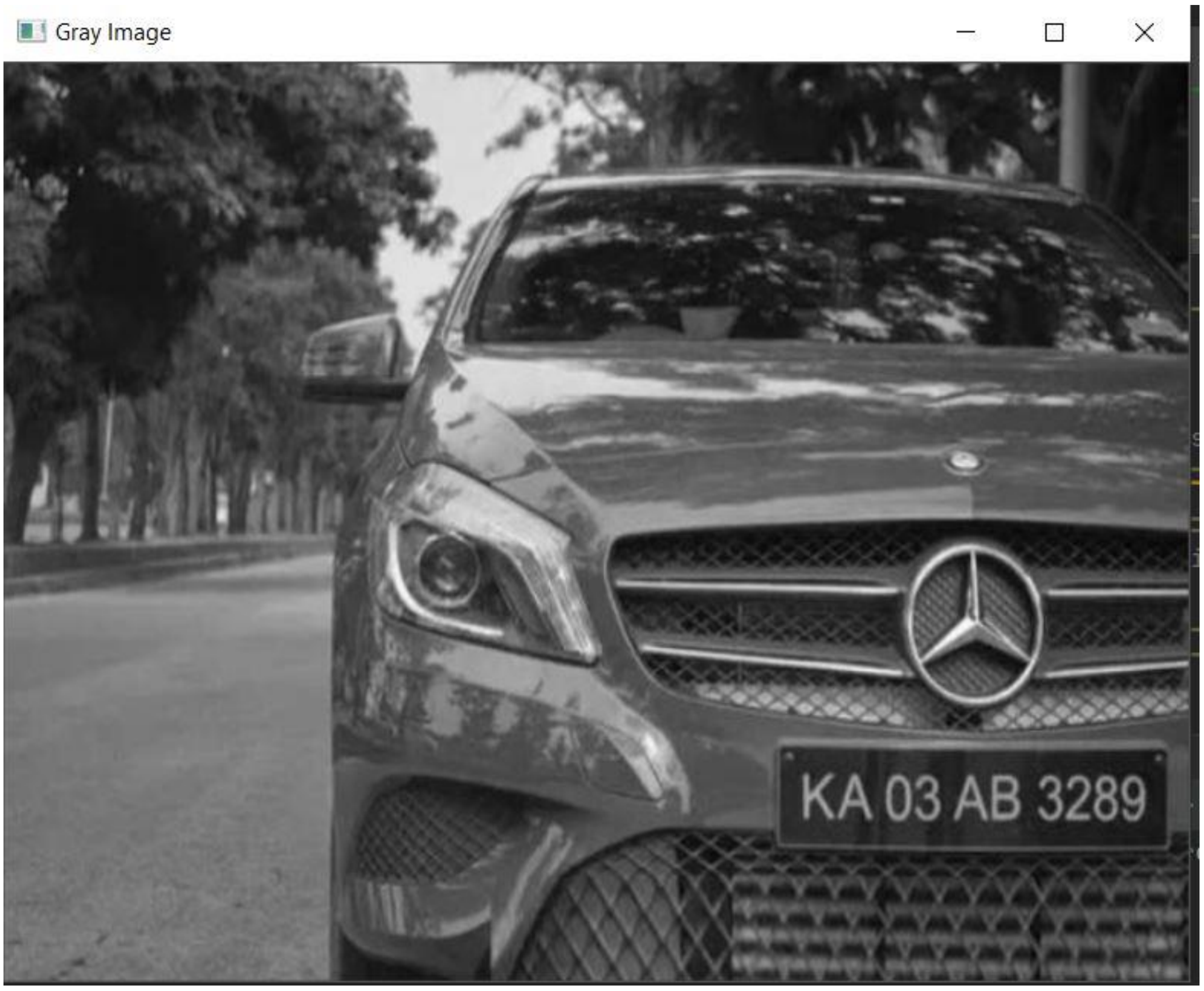
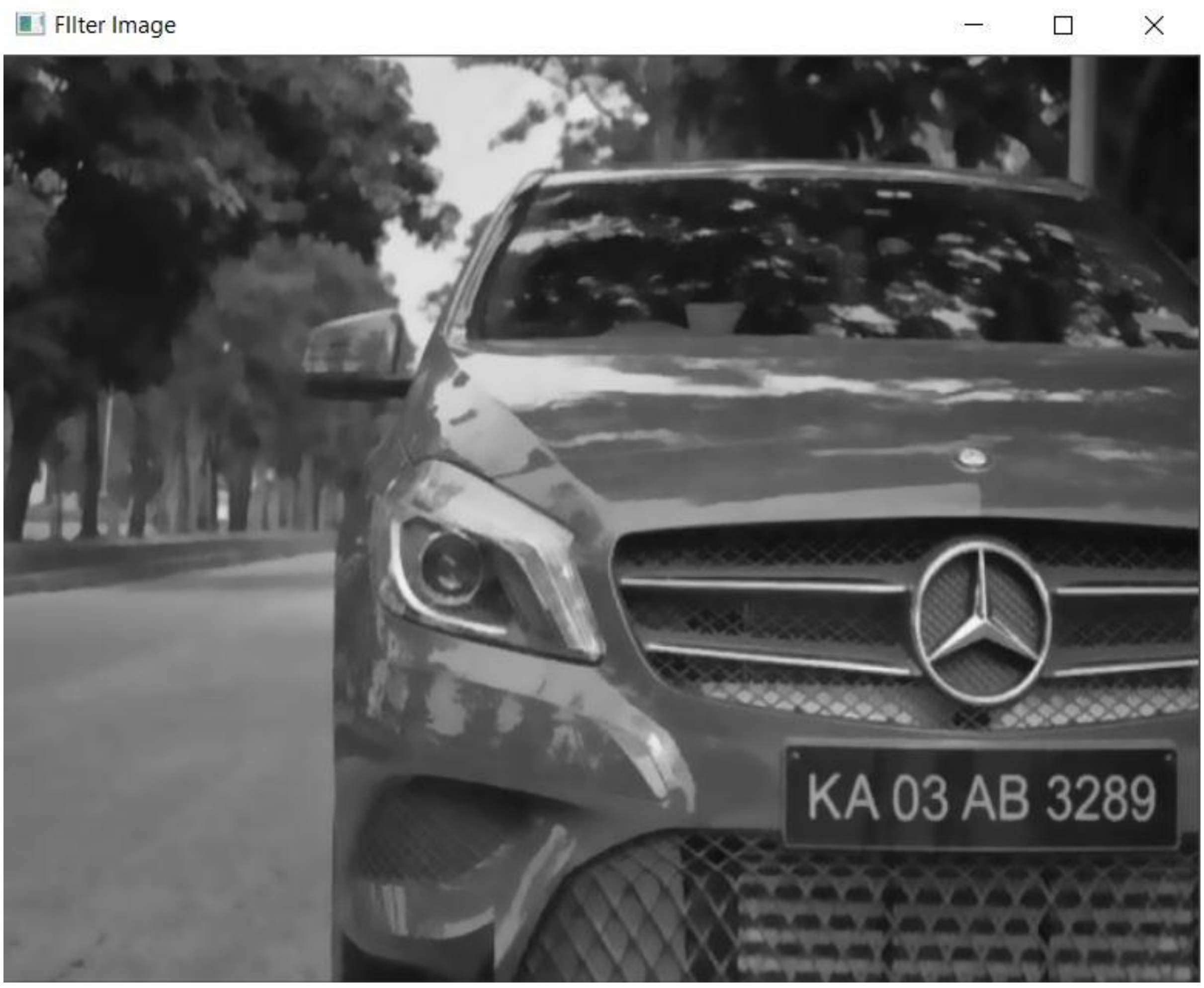


Plate localization

Syntax is destination_image = cv2.bilateralFilter(source_image, diameter of pixel, sigmaColor, sigmaSpace). we can increase the sigma color and sigma space from 15 to higher values to blur out more background information, but we should be careful that the useful part does not get blurred. The output image is shown below, as you can see the background details (tree and building) are blurred in this image. This way we can avoid the program from concentrating on these regions later.

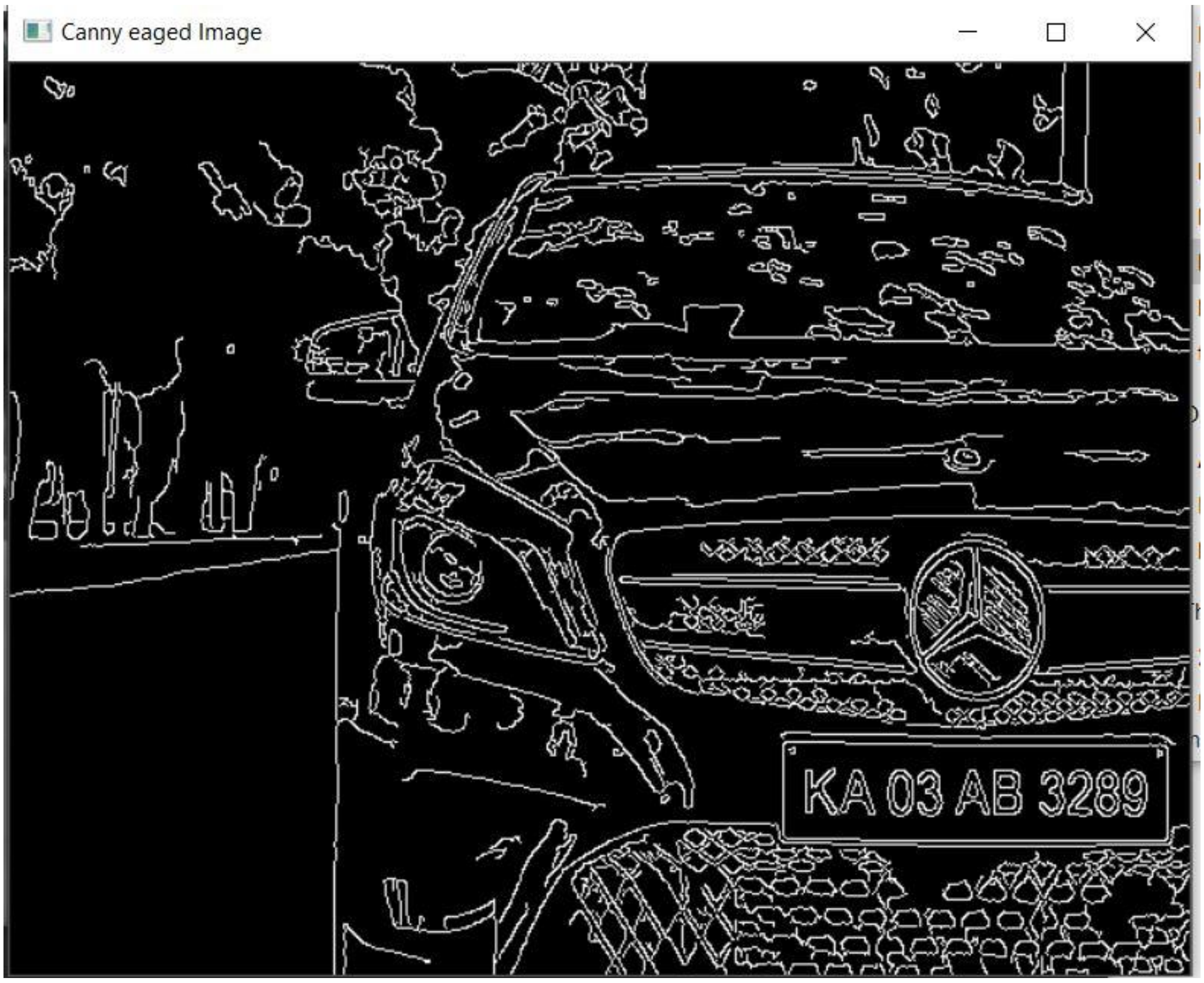
```
gray = cv2.bilateralFilter(gray, 11, 15, 15)
cv2.imshow("FIlter Image", gray)
```



Edge Detection

The syntax will be destination image = cv2.Canny(source_image, thresholdValue 1, thresholdValue 2). The Threshold Vale 1 and Threshold Value 2 are the minimum and maximum threshold values. Only the edges that have an intensity gradient more than the minimum threshold value and less than the maximum threshold value will be displayed

```
edged = cv2.Canny(gray, 30, 200)
cv2.imshow("Canny eaged Image", edged)
```



Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition.

In OpenCV, finding contours is like finding white object from black background. the boundaries of a shape with same intensity. It stores the (x,y) coordinates of the boundary of a shape

```
keypoints = cv2.findContours(edged.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
contours = imutils.grab_contours(keypoints)
contours= sorted(contours, key=cv2.contourArea, reverse=True)[:10]
```

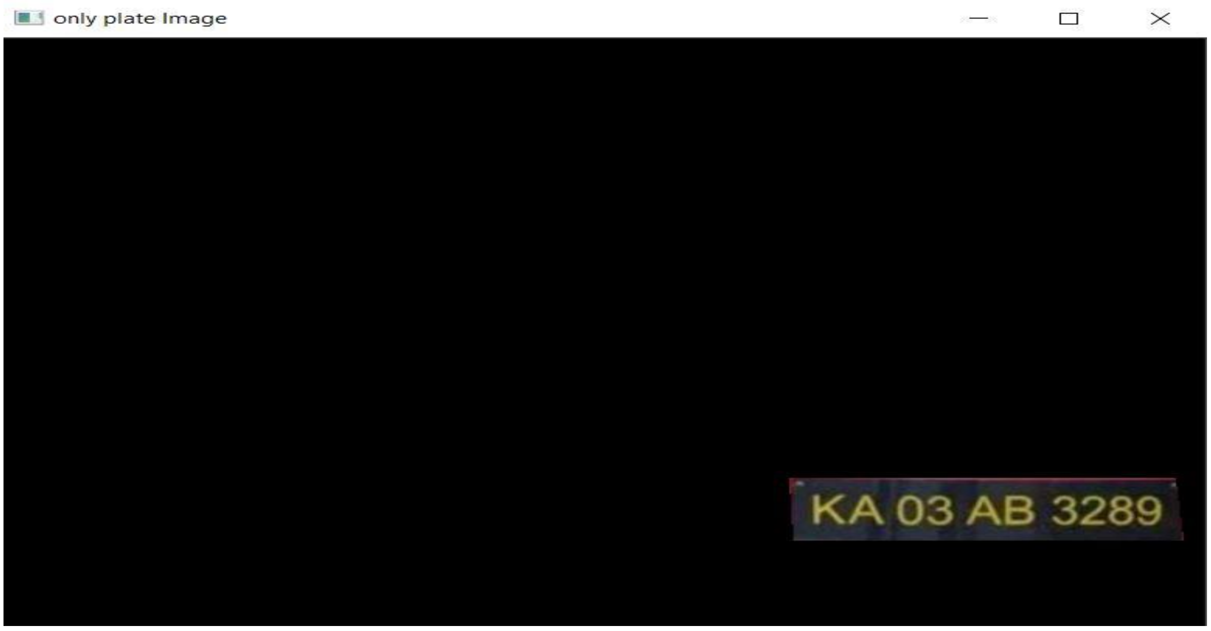
Once the counters have been detected we sort them from big to small and consider only the first 10 results ignoring the others. In our image the counter could be anything that has a closed surface but of all the obtained results the license plate number will also be there since it is also a closed surface.

To filter the license plate image among the obtained results, we will loop though all the results and check which has a rectangle shape contour with four sides and closed figure. Since a license plate would definitely be a rectangle four sided figure.

```
for contour in contours:
    approx = cv2.approxPolyDP(contour, 10, True)
    if len(approx) == 4:
        location = approx
        break
```

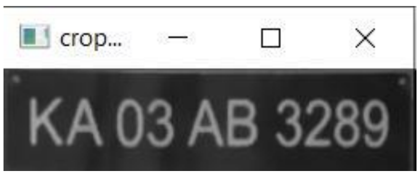
since we find the region of our plate the rest of the image is useless information and it can be masked out the code for that is:

```
mask = np.zeros(gray.shape, np.uint8)
new_image = cv2.drawContours(mask, [location], 0,255, -1)
new_image = cv2.bitwise_and(img, img, mask=mask)
```



Character Segmentation

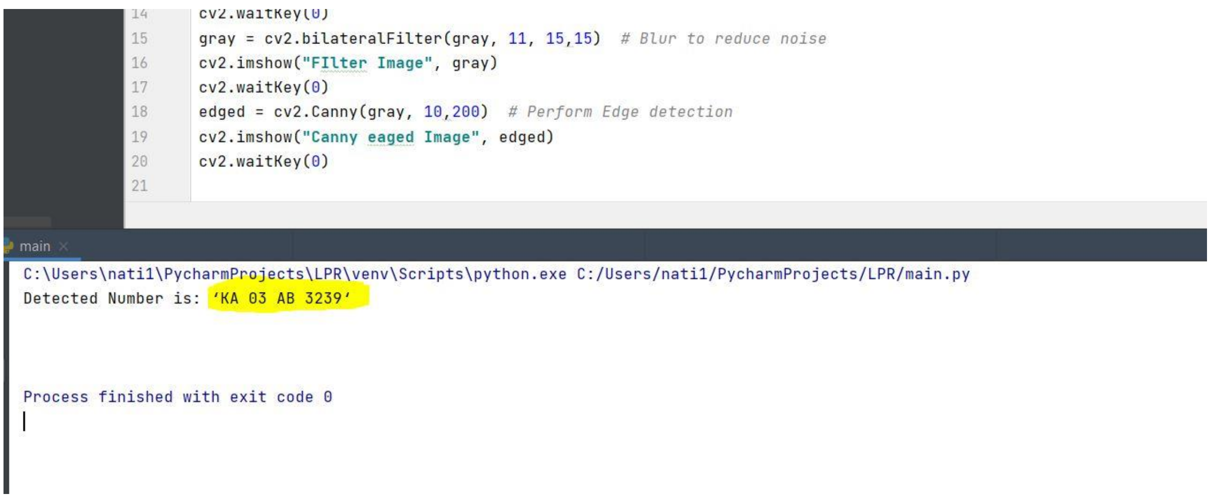
```
#for croping (character segmentation)
(x,y) = np.where(mask==255)
(x1, y1) = (np.min(x), np.min(y))
(x2, y2) = (np.max(x), np.max(y))
cropped_image = gray[x1:x2+1, y1:y2+1]
```



Character Recognition

The licence plate characters are recognized using pytesseract

```
plate = pytesseract.image_to_string(cropped_image, config='')
print("Detected Number is:",plate)
```



```
cv2.waitKey(0)
gray = cv2.bilateralFilter(gray, 11, 15,15)  # Blur to reduce noise
cv2.imshow("FIlter Image", gray)
cv2.waitKey(0)
edged = cv2.Canny(gray, 10,200)  # Perform Edge detection
cv2.imshow("Canny eaged Image", edged)
cv2.waitKey(0)

```

```
C:\Users\nati1\PycharmProjects\LPR\venv\Scripts\python.exe C:/Users/nati1/PycharmProjects/LPR/main.py
Detected Number is: 'KA 03 AB 3239'


Process finished with exit code 0
```

Conclusion

The overall aim of this project is to develop car license plate recognition system that able to detect and recognize car plates from visual sources using Digital signal processing and free and open source technologies like Python and OpenCV .

The accuracy depends on the clarity of image, orientation, light exposure etc. To get better results we can try implementing Machine learning algorithms. Most of the times of the image quality and orientation is correct, the program was able to identify the license plate and read the number from it.

References

[1]K. M. Babu and M. V. Raghunadh. Vehicle number plate de-
tection and recognition using bounding box method. In 2016
International Conference on Advanced Communication Con-
trol and Computing Technologies (ICACCCT), pages 106–110,
May 2016
[2]J. Canny. A computational approach to edge detection. IEEE
Transactions on Pattern Analysis and Machine Intelligence,
PAMI-8, 1986
[3] Andrew S. Agbemenu, Kumasi, Ghana Ernest O. Addo
Dept. of Computer Eng. KNUST
International Journal of Computer Applications (0975 - 8887)
Volume 180 - No.43, May 2018
An Automatic Number Plate Recognition System using
OpenCV and Tesseract OCR Engine
[4] Sweta Kumari,Leeza Gupta,Prena Gupta Department of Computer Science and
EngineeringDr. A.P.J Abdul Kalam Technical University, LucknowUttar Pradesh–India
Automatic License Plate Recognition Using OpenCV and Neural Network International
Journal of Computer Science Trends and Technology (IJCST) –Volume 5 Issue 3, May –Jun
2017
Links
https://opencv.org/
https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html
https://stackabuse.com/introduction-to-image-processing-in-python-with-opencv/
https://circuitdigest.com
https://medium.com/programming-fever/license-plate-recognition-using-opencv-python-
7611f85cdd6c

Appendixes

```
import cv2
import imutils
import numpy as np
import pytesseract

img = cv2.imread('3.jpg', cv2.IMREAD_COLOR)  # import our image
cv2.imshow("Original Image", img)        # display imported image
cv2.waitKey(0)
img = cv2.resize(img, (620, 480))        # resize the image
cv2.imshow("Resized Image", img)
cv2.waitKey(0)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # convert to grey scale
cv2.imshow("Gray Image", gray)
cv2.waitKey(0)
gray = cv2.bilateralFilter(gray, 11, 15,15)  # Blur to reduce noise
cv2.imshow("FIlter Image", gray)
cv2.waitKey(0)
edged = cv2.Canny(gray, 10,200)  # Perform Edge detection
cv2.imshow("Canny eaged Image", edged)
cv2.waitKey(0)

keypoints = cv2.findContours(edged.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
contours = imutils.grab_contours(keypoints)
contours= sorted(contours, key=cv2.contourArea, reverse=True)[:10]
location = None

for contour in contours:
    # approximate the contour
    approx = cv2.approxPolyDP(contour, 10, True)
    # if our approximated contour has four points, then
    # we can assume that we have found our screen
    if len(approx) == 4:
        location = approx
        break

mask = np.zeros(gray.shape, np.uint8)
new_image = cv2.drawContours(mask, [location], 0,255, -1)
new_image = cv2.bitwise_and(img, img, mask=mask)
cv2.imshow("only plate Image", new_image)
cv2.waitKey(0)
#for croping (character segmentation)
(x,y) = np.where(mask==255)
(x1, y1) = (np.min(x), np.min(y))
(x2, y2) = (np.max(x), np.max(y))
cropped_image = gray[x1:x2+1, y1:y2+1]
cv2.imshow("cropped image", cropped_image)
cv2.waitKey(0)


#Read the number plate
plate = pytesseract.image_to_string(cropped_image, config='')
print("Detected Number is:",plate)
```