Number plate localization using edge detection

PROJECT REPORT

Submitted By
VIRIYALA SAI VARALAKSHMI
Registration No. 20010104

in partial fulfillment of the requirements for the degree of
BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
SENAPATI, MANIPUR-795002, INDIA
MAY 2023

Abstract

Number plate detection and localization is an important task in computer vision and can be implemented using OpenCV. The goal is to locate and extract the license plate region from an input image or video stream. The process typically involves several steps, including image preprocessing, edge detection, contour detection, and character segmentation.

In the preprocessing step, the input image is first converted to grayscale and then filtered to remove noise and enhance the edges. Next, the edges are detected using techniques such as the Canny edge detector or Sobel filter.

Once the edges are detected, the contours are identified using OpenCV's findContours function. These contours are then filtered based on their area, aspect ratio, and position to extract the license plate region.

Finally, the license plate region is segmented into individual characters using techniques such as thresholding and morphological operations. This allows for optical character recognition (OCR) to be performed on the segmented characters to extract the license plate number.

Overall, the number plate detection and localization process in OpenCV involves a combination of preprocessing, edge and contour detection, and character segmentation to extract the license plate region from an input image or video stream.

Declaration

In this submission, I have expressed my idea in my own words, and I have adequately cited and referenced any ideas or words that were taken from another source. I also declare that I adhere to all principles of academic honesty and integrity and that I have not misrepresented or falsified any ideas, data, facts, or sources in this submission. If any violation of the above is made, I understand that the institute may take disciplinary action. Such a violation may also engender disciplinary action from the sources which are not properly cited or permission not taken when needed.

-Viriyala Sai Varalakshmi

20010104

Date: 19 November

Acknowledgement

I would like to express my sincere gratitude to several individuals for supporting me throughout my Project. First, I wish to express my sincere gratitude to my supervisor, Dr Kaushal Bhardwaj, for his enthusiasm, patience, insightful comments, helpful information, practical advice and unceasing ideas that have helped me tremendously at all times in my project and writing of this thesis. His immense knowledge, profound experience and professional expertise has enabled me to complete this project successfully. Without his support and guidance, this project would not have been possible. I could not have imagined having a better supervisor in my study.

-Viriyala Sai Varalakshmi

Contents

1 Introduction

Number plate detection by using OpenCV is developed to improve performance of detecting a numberplate by reducing its Complexity.This Project is classified into two maijor parts namely,

- Localization of Number plate
- Image Proccessing

Localization involves detection of numberplate in the given image using OpenCV.Localization by name gives locating a object from universe.Here we consider given image as universe,and the object to be located will be numberplate.
Image Processing meanly the extraction of data from the result of Localization part i.e to extract the text on the numberplate using EasyOCR.

1.1 Purpose

The main objective is to use OpenCV to detect numberplate instead of using ML train and test sets of data or using XML sheet fitting Process.OpenCV used in order to get rid the complexity in detection Process.

OpenCV

OpenCV is a high performance library for digital image processing and computer vision, which is free and open source. It is equipped with a large set of functions and algorithms for real-time computer vision and predictive mining. OpenCV was devised and developed by Intel, and the current instances are supported by W. Garage and Itseez. It was developed so that real-time analytics of images and recognition can be done for assorted applications. The statistical machine learning libraries used by OpenCV are:

- Deep neural networks (DNN)
- Convolutional neural networks (CNN)
- Boosting (meta-algorithm)
- Decision tree learning
- Gradient boosting trees
- Expectation-maximisation algorithm
- K-nearest neighbour algorithm
- Naive Bayes classifier
- Artificial neural networks
- Support vector machine (SVM)

1.2 Existing System

There are many existing vehicle number plate detection algorithms. Here are some popular ones:

1. Haar Cascades: Haar Cascades is a machine learning-based approach for object detection that uses a classifier trained on thousands of positive and negative images to detect objects in images. It is commonly used for face detection, but it can also be used for license plate detection.

2. YOLO (You Only Look Once): YOLO is a real-time object detection system that uses a single neural network to predict the bounding boxes and class probabilities for objects in an image. It is a popular algorithm for license plate detection due to its speed and accuracy.

3. SSD (Single Shot Detector): SSD is another real-time object detection algorithm that uses a single neural network to predict the bounding boxes and class probabilities for objects in an image. It is similar to YOLO in many ways but uses a different network architecture.

4. Faster R-CNN: Faster R-CNN is a two-stage object detection algorithm that first proposes regions of interest and then classifies them. It is slower than YOLO and SSD but is more accurate and can detect smaller objects more reliably.
Fast R-CNN is an object detection algorithm that is an improvement over the previous R-CNN (Region-based Convolutional Neural Networks) algorithm. The main problem with R-CNN was that it was very slow because it required running a separate Convolutional Neural Network (CNN) for each region proposal in an image.Fast R-CNN addressed this issue by introducing a single, shared CNN that can be used to process the entire image. Instead of running the CNN on each region proposal separately, Fast R-CNN uses a Region of Interest (RoI) pooling layer to extract a fixed-length feature vector from each proposal, which is then fed into a fully connected layer for classification and bounding box regression.Overall, Fast R-CNN is much faster than R-CNN, as it allows for the use of a single, shared CNN instead of running separate CNNs for each proposal. It is also more accurate than R-CNN, as it uses the RoI pooling layer to extract features from each proposal more efficiently.

5. Mask R-CNN: Mask R-CNN is an extension of Faster R-CNN that can also generate segmentation masks for objects in an image. It is used for more advanced applications such as instance segmentation, where the goal is to identify and segment individual objects within an image.

These are just a few of the many algorithms that can be used for license plate detection. The choice of algorithm depends on the specific requirements of the application, such as speed, accuracy, and complexity.

1.3 Intended Use

The cv2 library in Python, also known as OpenCV (Open Source Computer Vision Library), is not a machine learning library per se. Rather, it is a computer vision library that provides a wide range of functions for image and video processing, including image and video capture, filtering, feature detection, object detection, and more.
That being said, OpenCV does incorporate some machine learning functionality, particularly for object detection and recognition tasks. For example, it includes pre-trained models for object detection using Haar cascades, as well as the ability to train your own object detection models using support vector machines (SVMs), neural networks, and other machine learning techniques.
In addition to OpenCV, there are many other Python libraries that are commonly used for machine learning, such as TensorFlow, PyTorch, Scikit-Learn, and Keras. These libraries provide a wide range of machine learning algorithms and tools for tasks such as classification, regression, clustering, and more.

1.4 Scope

OpenCV is a library of programming functions mainly aimed at real-time computer vision, while Deep Learning is a subset of Machine Learning that uses algorithms to learn from data. Both technologies have their own advantages and disadvantages, and they can be used together to create powerful applications.

Advantages of OpenCV :

OpenCV has several advantages over Deep Learning. It is open source and free to use, so it is accessible to anyone. It is also fast and efficient, and can be used for real-time applications. OpenCV is also well-documented and has a large community of developers who can help with any issues.
OpenCV can be used for image localization through various techniques, including feature-based matching, template matching, and contour detection.
Here are some examples:

1. Feature-based matching: This technique involves detecting key features in an image, such as corners or edges, and then matching them with the corresponding features in another image. OpenCV provides functions such as $cv :: FeatureDetector$, $cv :: DescriptorExtractor$, and $cv :: DescriptorMatcher$ for feature-based matching.

2. Template matching: This technique involves searching for a template image within a larger image by sliding the template across the image and comparing pixel values. OpenCV provides the $cv :: matchTemplate$ function for template matching.

3. Contour detection: This technique involves identifying the boundaries of an object in an image by finding contours or edges. OpenCV provides functions such as $cv :: findContours$ and $cv :: Canny$ for contour detection.

Once an object has been localized, its position and orientation can be determined using geometric transformations such as affine transforms or perspective transforms. OpenCV provides functions such as $cv :: getAffineTransform$ and $cv :: getPerspectiveTransform$ for these transformations.
In summary, OpenCV provides a variety of functions and algorithms for image localization tasks, making it a useful tool for identifying the position and orientation of objects in images.

1.5 Requirements Engineering Software Model

Requirements engineering (RE) refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirement engineering is the disciplined application of proven principles, methods, tools, and notation to describe a proposed system's intended behavior and its associated constraints.

Requirement Engineering Process,It is a four-step process, which includes -

- Feasibility Study
- Requirement Elicitation and Analysis
- Software Requirement Specification
- Software Requirement Validation
- Software Requirement Management

2 System Design

Recognization of number plate is done by using OpenCV. This system has a python main code with CV2 library which used to get our localization part done.

2.1 System Analysis

Firstly it will take a image input and then it generate a localized output.This generation involves some functions and algorithms to detect the number plate.

2.1.1 Process Flow

In this section we will discuss the main idea behind the detection process in OpenCV.Firstly we will take input image of vehicle and then by edge detection process we will detect the edges by converting that image to grayscale and edge detection algorithm consists is the idea behind the canny edge detection algorithm.After detection of edges we will find contours to the image.Then by a algoritm we will detect numberplate. Process involved,

- Edge Detection
- Contours Finding
- Number Plate Detection

Edge Detection :

A Canny edge detector is a multi-step algorithm to detect the edges for any input image. It involves the below-mentioned steps to be followed while detecting edges of an image.

1. Removal of noise in input image using a Gaussian filter.

2. Computing the derivative of Gaussian filter to calculate the gradient of image pixels to obtain magnitude along x and y dimension.

3. Considering a group of neighbors for any curve in a direction perpendicular to the given edge, suppress the non-max edge contributor pixel points.

4. Lastly, use the Hysteresis Thresholding method to preserve the pixels higher than the gradient magnitude and neglect the ones lower than the low threshold value.

Before deep-diving into the steps below are the three conclusions that J.K Canny who derived the algorithm :

– Good Detection: The optimal detector must eliminate the possibility of getting false positives and false negatives.

– Good Localisation: The detected edges must be as close to true edges.

– Single Response Constraint: The detector must return one point only for each edge point.

Canny edge detection is a popular technique in computer vision for detecting edges in images. The steps of the Canny edge detection algorithm are as follows:

1. **Grayscale conversion**: Convert the input image to grayscale. This step is necessary because the algorithm works with intensity values rather than color.

2. **Gaussian blur**: Apply a Gaussian filter to the image to reduce noise and smooth out any irregularities.

3. **Gradient calculation**: Calculate the gradient of the image using a Sobel operator. This step involves computing the gradient in the x and y directions and then combining the two to get the gradient magnitude and direction.

4. **Non-maximum suppression**: Suppress all the gradient values that are not maximum in the gradient direction. This step helps to thin out the edges and remove any spurious responses.

5. **Double thresholding**: Set two thresholds, a high and a low threshold. All gradient values above the high threshold are considered as edges, and all values below the low threshold are not edges. Values between the two thresholds are considered edges only if they are connected to strong edges.

6. **Edge tracking by hysteresis**: In this final step, weak edges that are connected to strong edges are considered edges, and all others are suppressed. This step helps to complete edges that were interrupted by noise or other factors.

By combining these steps, the Canny edge detection algorithm can produce high-quality edge maps that are useful in a wide range of computer vision applications.

Contours Finding

Countour finding is a common image processing task in OpenCV. Contours are the boundaries of objects in an image and can be used for various tasks like object detection, recognition, and tracking. Here are the general steps to find contours in an image using OpenCV:

1. **Load the image**: Read the image from a file or capture it from a camera.

2. **Preprocessing**: Preprocess the image as needed, such as converting it to grayscale, blurring, or thresholding.

3. **Find contours**: Use the 'findContours()' function in OpenCV to detect the contours in the preprocessed image. This function takes the input image, the mode of contour retrieval (e.g., external, all, etc.), and the method of contour approximation (e.g., simple, complex, etc.) as parameters.

4. **Draw contours**: After finding the contours, you can draw them on the image using the 'drawContours()' function in OpenCV. This function takes the input image, the list of contours, the index of the contour to draw (or -1 for all contours), the color of the contour, and the thickness of the line as parameters.

5. **Analyze the contours**: You can analyze the contours to extract various information about the objects in the image, such as their size, shape, position, and orientation.

Note that the 'findContours()' function modifies the input image, so you may want to make a copy of the image before calling it if you need to use the original image later.
Overall, contour finding is a powerful tool in OpenCV for various computer vision tasks and can be easily implemented using the functions provided in the OpenCV library.

Number Plate Detection

Once you have found the contours in an image using OpenCV, you can analyze the contours to detect rectangular shapes. Here are the general steps to detect a rectangle from the contour results:

1. **Filter contours**: Depending on the application, you may want to filter the contours based on their size, shape, or other characteristics. For example, you can remove contours that are too small or too large to be rectangles.

2. **Approximate contours**: Use the ‘approxPolyDP()‘ function in OpenCV to approximate the contours with simpler shapes, such as polygons. This function takes the contour points and the maximum distance from the contour to the polygon as parameters. For a rectangle, you can set the maximum distance to a small value.

3. **Find rectangular contours**: After approximating the contours, you can filter them further to only keep those that are rectangular. One way to do this is to check the number of vertices in the polygon approximation. A rectangle has four vertices, so you can keep only the contours with four vertices.

4. **Check aspect ratio**: Another way to filter rectangular contours is to check their aspect ratio. A rectangle has a width-to-height ratio of approximately 1, so you can keep only the contours with a similar aspect ratio.

5. **Draw rectangles**: After detecting the rectangular contours, you can draw rectangles on the image using the ‘rectangle()‘ function in OpenCV. This function takes the input image, the coordinates of the top-left and bottom-right corners of the rectangle, the color of the rectangle, and the thickness of the line as parameters.

Overall, detecting rectangular shapes from contours in an image is a powerful tool in OpenCV for various computer vision tasks, such as object detection and recognition. The OpenCV library provides many functions to implement these steps easily.

2.2 System Requirements

The edge detection algorithm is a well-known and widely used technique for detecting edges in images. Here are the system requirements for running the edge detection algorithm:

1. Hardware requirements:

- CPU: The algorithm is computationally intensive and may require a powerful CPU for processing large images or real-time video streams.

- Memory: Sufficient RAM is required to store the input image and intermediate results during processing.

- GPU: Some implementations of the Canny edge detection algorithm can be accelerated using a GPU.

2. Software requirements:

- Operating system: The algorithm can be run on various operating systems such as Windows, Linux, or macOS.

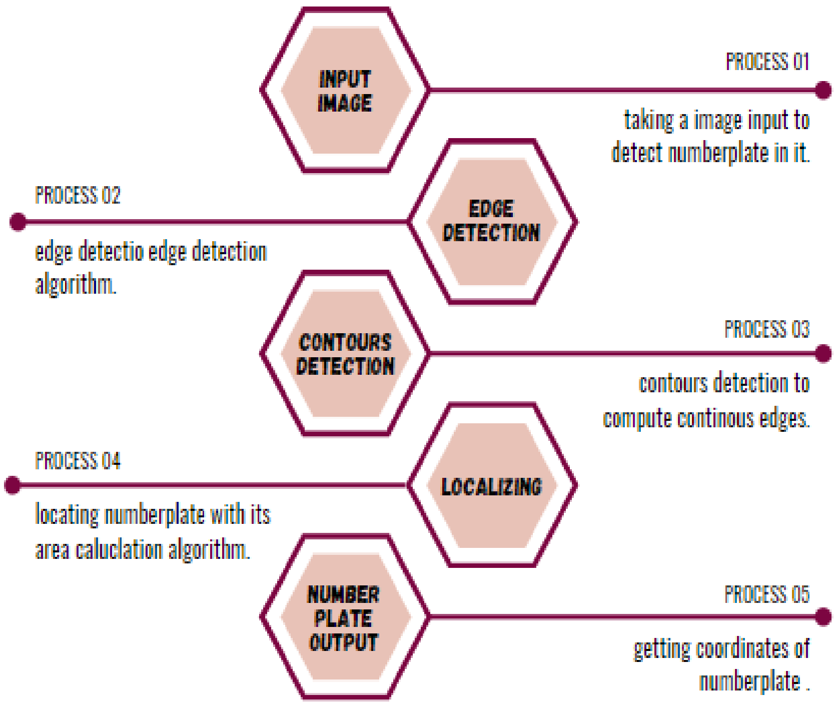
- Programming language: The Canny edge detection algorithm can be implemented in various programming languages such as Python, C++, Java, and others.

- Image processing libraries: The Canny edge detection algorithm can be implemented using various image processing libraries, including OpenCV, MATLAB, and others.

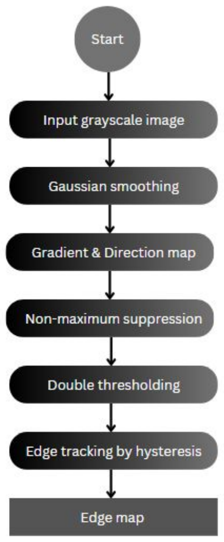
In summary, running the edge detection algorithm requires a computer with sufficient processing power, memory, and software libraries. The exact system requirements will depend on the size and complexity of the input images, the desired performance, and the implementation used.

2.2.1 UML Diagrams

SYSTEM DESIGN


INPUT IMAGE
PROCESS 01
taking a image input to detect numberplate in it.
EDGE DETECTION
PROCESS 02
edge detectio edge detection algorithm.
CONTOURS DETECTION
PROCESS 03
contours detection to compute continous edges.
LOCALIZING
PROCESS 04
locating numberplate with its area caluclation algorithm.
NUMBER PLATE OUTPUT
PROCESS 05
getting coordinates of numberplate .

Edge detection Process Flow Diagram



3 Implementation and Design Analysis

Edge detection is a fundamental technique in image processing that involves identifying and locating edges or boundaries within an image. An edge can be defined as the boundary between two distinct regions within an image, where the regions differ in intensity or color.

Edge detection is an important step in many computer vision and image processing applications such as object detection, image segmentation, and feature extraction. There are various techniques for edge detection, including gradient-based methods, Laplacian-based methods, and machine learning-based methods.

Gradient-based methods involve calculating the gradient of an image, which represents the rate of change of intensity or color with respect to the position. Common gradient-based edge detection operators include the Sobel operator, Prewitt operator, and Roberts operator.
Laplacian-based methods involve computing the second derivative of an image to detect edges. These methods are more sensitive to noise and require a smoothing step to improve the results.

Machine learning-based methods involve training a machine learning model to detect edges in an image. These methods require large amounts of training data and may not perform well on images outside of the training data.
After detecting the edges in an image, post-processing steps like thresholding, non-maximum suppression, and hysteresis thresholding may be used to refine the edge detection results.

Sure, here's a step-by-step process for implementing the edge detection algorithm:

1. Load the input image: Use an image processing library like OpenCV to load the input image in a suitable format.

2. Convert the image to grayscale: Convert the color image to a grayscale image using a function like $cv2.cvtColor()$ in OpenCV. This reduces the computation required for edge detection and simplifies the algorithm.

3. Apply Gaussian blur: Apply a Gaussian blur to the grayscale image using a function like $cv2.GaussianBlur()$ in OpenCV. This helps to smooth out any noise in the image, which can help produce cleaner edge detection results.

4. Calculate gradients: Use a gradient operator like the Sobel operator to calculate the gradients (derivatives) of the grayscale image in both the x and y directions. This is done using a function like $cv2.Sobel()$ in OpenCV.

5. Non-maximum suppression: Perform non-maximum suppression on the gra-

dient image to thin the edges and ensure that only the most prominent edges are detected. This involves comparing the magnitude of the gradient at each pixel to the magnitudes of the two neighboring pixels in the direction of the gradient.

6. Double thresholding: Apply double thresholding to the edge map to separate strong edges from weak edges. This involves setting two threshold values, a low threshold and a high threshold. Any pixel with a gradient magnitude above the high threshold is considered a strong edge, while any pixel with a gradient magnitude below the low threshold is considered a non-edge. Pixels with gradient magnitudes between the low and high thresholds are considered weak edges.

7. Edge tracking by hysteresis: Finally, perform edge tracking by hysteresis to link weak edges to strong edges and produce the final edge map. This involves starting at a strong edge pixel and tracing along connected weak edge pixels until the edge is no longer followed. This is repeated until all possible edges have been tracked.

8. Display the output: Finally, display the resulting edge map using a function like $cv2.imshow()$ in OpenCV.

Note that some of these steps can be combined or optimized based on the specific implementation and requirements of the application. Additionally, there may be some additional steps involved depending on the complexity of the input image or the desired edge detection results.

3.1 Functions Libraries

This section denotes detailed information about the functions used and the way of usage too.

Graysacle function :

The $cvtColor()$ function in OpenCV is a versatile function that can be used to convert an image from one color space to another. In the case of grayscale conversion, $cvtColor()$ is used to convert a color image to a grayscale image. Here's a brief explanation of how $cvtColor()$ works for grayscale conversion:

$gray_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)$

In this example, $color_image$ is the input color image, and $gray_image$ is the output grayscale image.
The second argument $cv2.COLOR_BGR2GRAY$ specifies the conversion code for the $cvtColor()$ function. This code tells OpenCV which color space con-

version to perform. In this case, the conversion code $cv2.COLOR_BGR2GRAY$ means that the function should convert the input image from the BGR color space to grayscale.

The $cvtColor()$ function converts the color image to grayscale by taking the average value of the three color channels (red, green, and blue) at each pixel. This results in a single grayscale value for each pixel that represents the brightness of that pixel. The resulting grayscale image has only one channel (compared to the three channels of the input color image) and contains values ranging from 0 (black) to 255 (white).
In summary, the $cvtColor()$ function in OpenCV is a powerful tool for converting images between different color spaces, including grayscale conversion. The $cv2.COLOR_BGR2GRAY$ parameter tells the function to convert from the BGR color space to grayscale by averaging the color channels.

Working of cv2.Color() :

In OpenCV, the $cv2.cvtColor()$ function is used to convert an image from one color space to another. The function takes two arguments: the input image and the target color space. The function works by applying a linear or nonlinear transformation to each pixel in the input image to convert it to the target color space. The transformation is based on the properties of the color spaces being converted between.

For example, to convert an RGB image to grayscale, the $cv2.cvtColor()$ function applies the following linear transformation to each pixel:
$gray = 0.299 * R + 0.587 * G + 0.114 * B$
where 'R', 'G', and 'B' are the red, green, and blue color channels of the input image, respectively.
Similarly, to convert an RGB image to the HSV (hue, saturation, value) color space, the $cv2.cvtColor()$ function applies a nonlinear transformation to each pixel based on the color space's properties.

The $cv2.cvtColor()$ function supports various color spaces, including RGB, BGR, grayscale, HSV, LAB, and YUV, among others.
It's important to note that not all color spaces are suitable for all image processing tasks. The choice of color space depends on the specific task and the properties of the input images.

Noise reduction by gaussian blur :

cv2.GaussianBlur() is a function in the OpenCV library that applies Gaussian blur to an image. Gaussian blur is a type of image-blurring algorithm that is commonly used to reduce noise and smooth out the image.
The function takes the following arguments:

src: The input image.
ksize: The size of the Gaussian kernel. This should be an odd number in both dimensions (e.g., (5,5), (7,7), etc.).
sigmaX: The standard deviation in the X direction.
sigmaY: The standard deviation in the Y direction. If this value is not specified, it is set to be the same as sigmaX.
The function returns the Gaussian-blurred image.

The Gaussian kernel is a two-dimensional matrix that is used to convolve the image. It is calculated based on the values of the standard deviation (sigma) and the kernel size. The larger the value of sigma, the more blur is applied to the image. The kernel size controls the extent of the blur, and larger kernel sizes result in more blurring.

Gradient caluclation by sobel :

Sobel:
The Sobel operator is a common method for calculating the gradient of an image. It is often used in computer vision and image processing applications to detect edges in images.
The Sobel operator uses two kernels, one for the x-axis and one for the y-axis, to calculate the gradient. The kernels are small matrices that are convolved with the image.

Here are the kernels for the x and y directions:

$$\text{Gx} =$$

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$$\text{Gy} =$$

$$\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

To apply the Sobel operator to an image, you first convert the image to grayscale. Then, you convolve the image with the Gx and Gy kernels separately, to obtain the x and y gradients, respectively. Finally, you can calculate the gradient magnitude by combining the x and y gradients using the following formula:

$\text{magnitude} = \text{sqrt}(\text{Gx}^2 + Gy^2)$

The resulting magnitude image will have high values at edges and low values elsewhere. You can then threshold the magnitude image to obtain a binary edge map, where edges are white and the background is black.
we load an image in grayscale and define the Sobel kernels. We then use

cv2.filter2D to perform a convolution on the input image with each kernel and obtain the x and y gradients. The output images Ix and Iy will have the same size and depth as the input image, and will contain the filtered output.

Non-maximum supression :

Non-maximum suppression (NMS) is a technique used in computer vision to suppress the non-maximal values in an image or feature map, while preserving the maximum values. This technique is commonly used to reduce the number of overlapping detections in object detection algorithms.

Here is the implementation logic for non-maximum suppression:

1. Define the input parameters
- ‘boxes‘: A list of bounding boxes to perform NMS on.
- ‘scores‘: A list of scores associated with each bounding box.
- ‘threshold‘: The overlap threshold used to decide which boxes to suppress.

- ‘mode‘: The NMS mode used to determine which boxes to suppress. The two most common modes are ‘union‘ and ‘min‘.

2. Create two lists, ‘x1‘, ‘y1‘, ‘x2‘, and ‘y2‘, to store the coordinates of the bounding boxes.

3. Calculate the area of each bounding box.

4. Sort the boxes by their scores in descending order.

5. Initialize an empty list to store the boxes to keep.

6. Loop over the boxes in the sorted order, This loop goes through each bounding box in order of its score, starting from the one with the highest score. It adds the current box to the list of boxes to keep, and then calculates the overlap of this box with all the remaining boxes. If the overlap is above the threshold, the overlapping box is removed from the list of remaining boxes, and the loop continues.

7. Return the list of boxes to keep.

This implementation logic performs non-maximum suppression on a list of bounding boxes and their associated scores, by calculating the overlap between the boxes and removing the boxes that have overlap above a certain threshold. The remaining boxes are the ones with the highest scores and non-overlapping with other boxes.

Double thresholding :

Double thresholding is a technique used in image processing to identify edges in an image. It is commonly used as a pre-processing step in edge detection algorithms.The Canny edge detection algorithm is a popular technique used to identify edges in images. It works by detecting the gradient of the image intensity and then applying a series of operations to highlight the edges.One important step in the Canny algorithm is double thresholding. This step is used to filter out weak edges and noise from the final edge map.

Double thresholding involves setting two thresholds: a high threshold and a low threshold. Any edge pixels with gradient magnitude above the high threshold are considered strong edges, while edge pixels with gradient magnitude below the low threshold are considered non-edges. Edge pixels with gradient magnitude between the high and low thresholds are considered weak edges.The weak edges are then further processed by checking if they are connected to strong edges. If a weak edge is connected to a strong edge, it is also considered a strong edge and included in the final edge map. If it is not connected to any strong edges, it is discarded as noise.By setting two thresholds, double thresholding allows for greater control over the sensitivity of the edge detection algorithm and helps to reduce false positives and noise in the final output.

Here is the implementation logic for double thresholding:

1. Calculate the gradient magnitude and direction of the image using a derivative filter, such as the Sobel operator.

2. Apply non-maximum suppression to thin the edges and obtain a one-pixel-wide edge map.

3. Set two thresholds: a high threshold and a low threshold. The high threshold is used to identify strong edges, while the low threshold is used to identify weak edges.

4. Identify strong edges by checking which pixels have a gradient magnitude greater than the high threshold. These pixels are marked as strong edges.

5. Identify weak edges by checking which pixels have a gradient magnitude between the high and low thresholds. These pixels are marked as weak edges.

6. Discard all pixels below the low threshold.

7. Perform hysteresis thresholding by connecting weak edges to strong edges. This involves checking if each weak edge is connected to a strong edge, either directly or indirectly. If it is, it is marked as a strong edge. If it is not, it is discarded as noise.

The high and low thresholds are usually set based on the signal-to-noise ratio of the image. If the image has high levels of noise, the high threshold should be set lower to capture more edges. Conversely, if the image has low levels of noise, the high threshold can be set higher to capture only strong edges. The low threshold is typically set to be a fraction of the high threshold.Overall, double thresholding in Canny edge detection provides a way to fine-tune the edge detection process and reduce false positives and noise in the final output.

Edge tracking by hysteresis:

Edge hysteresis is the final step in the Canny algorithm and is used to connect weak edges to strong edges. This step helps to form continuous edges by linking edge pixels that belong to the same edge but may have been broken up by noise or other factors.

Here are the steps involved in edge hysteresis:

1. Starting with the strong edges identified in the double thresholding step, trace the edge by following the gradient direction and connecting any adjacent weak edges that are above the low threshold.

2. Continue tracing the edge until no more weak edges can be found.

3. Repeat this process for all strong edges.

This process of tracing and connecting weak edges to strong edges helps to form continuous edges and fill in any gaps that may have been created by noise or other factors.
The threshold values used in edge hysteresis play an important role in the final output. If the low threshold is too low, it can lead to a large number of false positives and noise in the final output. On the other hand, if the low threshold is too high, it may miss weak edges that are part of the overall edge structure. Overall, edge hysteresis is an important step in the Canny edge detection algorithm that helps to improve the accuracy and continuity of the detected edges.

Contour Detetction:

When we join all the points on the boundary of an object, we get a contour. Typically, a specific contour refers to boundary pixels that have the same color and intensity. OpenCV makes it really easy to find and draw contours in images. It provides two simple functions:

1.findContours()
2.drawContours()

Also, it has two different algorithms for contour detection:

1.$CHAIN\_APPROX\_SIMPLE$
2.$CHAIN_APPROX_NONE$

Steps for Detecting and Drawing Contours in OpenCV :

OpenCV makes this a fairly simple task with following steps,

1.Read the Image and convert it to Grayscale Format
Read the image and convert the image to grayscale format. Converting the image to grayscale is very important as it prepares the image for the next step. Converting the image to a single channel grayscale image is important for thresholding, which in turn is necessary for the contour detection algorithm to work properly.

2.Apply Binary Thresholding
While finding contours, first always apply binary thresholding or Canny edge detection to the grayscale image. Here, we will apply binary thresholding.
This converts the image to black and white, highlighting the objects-of-interest to make things easy for the contour-detection algorithm. Thresholding turns the border of the object in the image completely white, with all pixels having the same intensity. The algorithm can now detect the borders of the objects from these white pixels. Note that black pixels, having value 0, are perceived as background pixels and ignored. At this point, one question may arise. What if we use single channels like R (red), G (green), or B (blue) instead of grayscale (thresholded) images? In such a case, the contour detection algorithm will not work well. As we discussed previously, the algorithm looks for borders, and similar intensity pixels to detect the contours. A binary image provides this information much better than a single (RGB) color channel image. In a later portion of the blog, we have resultant images when using only a single R, G, or B channel instead of grayscale and thresholded images.

3.Find the Contours
Use the findContours() function to detect the contours in the image.

4.Draw Contours on the Original RGB Image.
Once contours have been identified, use the drawContours() function to overlay the contours on the original RGB image.

findContours() function. It has three required arguments,

image: The binary input image obtained in the previous step.

mode: This is the contour-retrieval mode. We provided this as $RETR\_TREE$, which means the algorithm will retrieve all possible contours from the binary image. More contour retrieval modes are available, we will be discussing them too. You can learn more details on these options here.

method: This defines the contour-approximation method. In this example, we will use $CHAIN\_APPROX\_NONE$.Though slightly slower than $CHAIN\_APPROX\_SIMPLE$, we will use this method here tol store ALL contour points.

DrawContours() function to overlay the contours on the RGB image. This function has four required and several optional arguments. The first four arguments below are required.

image: This is the input RGB image on which you want to draw the contour.
contours: Indicates the contours obtained from the findContours() function.
contourIdx: The pixel coordinates of the contour points are listed in the obtained contours. Using this argument, you can specify the index position from this list, indicating exactly which contour point you want to draw. Providing a negative value will draw all the contour points.
color: This indicates the color of the contour points you want to draw. We are drawing the points in green.
thickness: This is the thickness of contour points.

The $CHAIN\_APPROX\_SIMPLE$ algorithm compresses horizontal, vertical, and diagonal segments along the contour and leaves only their end points. This means that any of the points along the straight paths will be dismissed, and we will be left with only the end points. For example, consider a contour, along a rectangle. All the contour points, except the four corner points will be dismissed. This method is faster than the $CHAIN\_APPROX\_NONE$ because the algorithm does not store all the points, uses less memory, and therefore, takes less time to execute.

Contour Hierarchies :

Hierarchies denote the parent-child relationship between contours. You will see how each contour-retrieval mode affects contour detection in images, and produces hierarchical results.

Parent-Child Relationship:

Objects detected by contour-detection algorithms in an image could be:
1.Single objects scattered around in an image or
2.Objects and shapes inside one another
In most cases, where a shape contains more shapes, we can safely conclude that the outer shape is a parent of the inner shape.

Different Contour Retrieval Techniques :
one specific retrieval technique, *RETR_TREE* to find and draw contours, but there are three more contour-retrieval techniques in OpenCV, namely, *RETR_LIST*, *RETR_EXTERNAL* and *RETR_CCOMP*.

RETR_LIST :
The *RETR_LIST* contour retrieval method does not create any parent child relationship between the extracted contours. So, for all the contour areas that are detected, the First Child and Parent index position values are always -1.All the contours will have their corresponding Previous and Next contours.

RETR_EXTERNAL :
The *RETR_EXTERNAL* contour retrieval method is a really interesting one. It only detects the parent contours, and ignores any child contours.

RETR_CCOMP :
Unlike *RETR_EXTERNAL*,*RETR_CCOMP* retrieves all the contours in an image. Along with that, it also applies a 2-level hierarchy to all the shapes or objects in the image. This means:

1.All the outer contours will have hierarchy level 1
2.All the inner contours will have hierarchy level 2

RETR_TREE :
Just like *RETR_CCOMP*, *RETR_TREE* also retrieves all the contours. It also creates a complete hierarchy, with the levels not restricted to 1 or 2. Each contour can have its own hierarchy, in line with the level it is on, and the corresponding parent-child relationship that it has.

RETR_LIST and *RETR_EXTERNAL* take the least amount of time to execute, since *RETR_LIST* does not define any hierarchy and *RETR_EXTERNAL* only retrieves the parent contours. *RETR_CCOMP* takes the second highest time to execute. It retrieves all the contours and defines a two-level hierarchy. *RETR_TREE* takes the maximum time to execute for it retrieves all the contours, and defines the independent hierarchy level for each parent-child relationship as well.

approxpolydp function:

This is a function in OpenCV that approximates a polygonal curve with a specified precision. It takes a contour as input and returns a new contour that has fewer vertices and approximates the original contour with a specified maximum distance (epsilon) between the original and the approximated contours.
The function works by iteratively removing the vertices that do not contribute significantly to the overall shape of the contour. This is done by computing the distance between the original contour and its approximated version and remov-

ing any vertices that fall within a distance threshold. The threshold is controlled by the 'epsilon' parameter, which is specified as a percentage of the contour's perimeter.
The *approxPolyDP* function is commonly used in shape analysis and recognition applications where the number of vertices in a contour needs to be reduced while preserving the overall shape of the contour. For example, it can be used to approximate the shape of a polygonal object detected in an image or to extract the edges of a shape for further processing or analysis. Overall, *approxPolyDP* is a useful function in OpenCV for approximating polygonal curves with a specified precision, and it can be applied to a variety of applications in computer vision and image processing.

approxPolyDP is a function in OpenCV that approximates a given polygonal curve with a new curve that has fewer vertices but still closely resembles the original curve. The function takes as input a contour represented as a vector of points, and returns a new contour represented as a vector of points that approximates the original contour with a specified maximum distance between the original and approximated contours.
The function works by iteratively simplifying the contour using the Douglas-Peucker algorithm, which is a recursive method for reducing the number of vertices in a curve. The Douglas-Peucker algorithm works by finding the point on the curve that is farthest from the line segment connecting the endpoints of the curve. If this point is farther than the specified maximum distance (epsilon) from the line segment, it is added to the new approximated curve. Otherwise, the curve is split into two sub-curves, and the Douglas-Peucker algorithm is applied recursively to each sub-curve.
The *approxPolyDP* function allows for the input contour to be approximated with a high degree of accuracy by controlling the value of epsilon. A smaller value of epsilon will result in a more accurate approximation of the original contour, but will also result in more vertices in the approximated contour. Conversely, a larger value of epsilon will result in fewer vertices in the approximated contour, but will result in a less accurate approximation of the original contour.

Overall, *approxPolyDP* is a useful function in OpenCV for simplifying polygonal curves while preserving their shape and reducing the number of vertices. It can be used in a variety of applications such as shape recognition, contour extraction, and image segmentation.

3.2 Design Algorithm

edge detection process :

```
def Canny_detector(img, weak_th = None, strong_th = None):

# conversion of image to grayscale
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Noise reduction step
img = cv2.GaussianBlur(img, (5, 5), 1.4)
# Calculating the gradients
gx = cv2.Sobel(np.float32(img), cv2.CV_64F, 1, 0, 3)
gy = cv2.Sobel(np.float32(img), cv2.CV_64F, 0, 1, 3)
        # Computed magnitude and orientation of gradients
# Conversion of Cartesian coordinates to polar
mag, ang = cv2.cartToPolar(gx, gy, angleInDegrees = True)

# setting the minimum and maximum thresholds
# for double thresholding
mag_max = np.max(mag)
if not weak_th:weak_th = mag_max * 0.1
if not strong_th:strong_th = mag_max * 0.5

# getting the dimensions of the input image
height, width = img.shape

# Looping through every pixel of the grayscale
# image
for i_x in range(width):
for i_y in range(height):

grad_ang = ang[i_y, i_x]
grad_ang = abs(grad_ang-180) if abs(grad_ang)>180 else abs(grad_ang)

# selecting the neighbours of the target pixel
# according to the gradient direction
# In the x axis direction
if grad_ang<= 22.5:
neighb_1_x, neighb_1_y = i_x-1, i_y
neighb_2_x, neighb_2_y = i_x + 1, i_y

# top right (diagonal-1) direction
elif grad_ang>22.5 and grad_ang<=(22.5 + 45):
neighb_1_x, neighb_1_y = i_x-1, i_y-1
```

```
neighb_2_x, neighb_2_y = i_x + 1, i_y + 1

# In y-axis direction
elif grad_ang>(22.5 + 45) and grad_ang<=(22.5 + 90):
neighb_1_x, neighb_1_y = i_x, i_y-1
neighb_2_x, neighb_2_y = i_x, i_y + 1

# top left (diagonal-2) direction
elif grad_ang>(22.5 + 90) and grad_ang<=(22.5 + 135):
neighb_1_x, neighb_1_y = i_x-1, i_y + 1
neighb_2_x, neighb_2_y = i_x + 1, i_y-1

# Now it restarts the cycle
elif grad_ang>(22.5 + 135) and grad_ang<=(22.5 + 180):
neighb_1_x, neighb_1_y = i_x-1, i_y
neighb_2_x, neighb_2_y = i_x + 1, i_y

# Non-maximum suppression step
if width>neighb_1_x>= 0 and height>neighb_1_y>= 0:
if mag[i_y, i_x]<mag[neighb_1_y, neighb_1_x]:
mag[i_y, i_x]= 0
continue

if width>neighb_2_x>= 0 and height>neighb_2_y>= 0:
if mag[i_y, i_x]<mag[neighb_2_y, neighb_2_x]:
mag[i_y, i_x]= 0

weak_ids = np.zeros_like(img)
strong_ids = np.zeros_like(img)
ids = np.zeros_like(img)

# double thresholding step
for i_x in range(width):
for i_y in range(height):

grad_mag = mag[i_y, i_x]

if grad_mag<weak_th:
mag[i_y, i_x]= 0
elif strong_th>grad_mag>= weak_th:
ids[i_y, i_x]= 1
else:
ids[i_y, i_x]= 2


# finally returning the magnitude of
# gradients of edges
return mag
```

in edge detection to explain clearly the crutial step is non-max supression in which we use the output of gradient caluclated with thw sliding window protocol and angle involved.

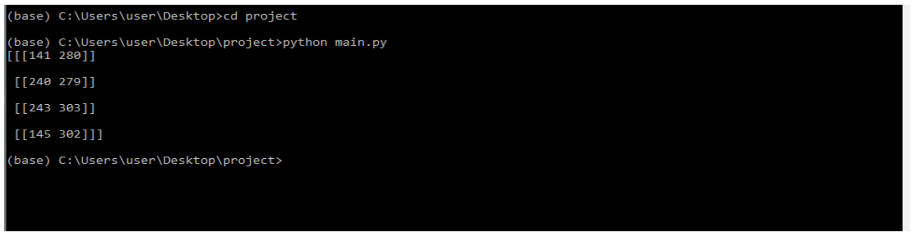
4 Testing

This sectoion defines the actual working of designed model.This also gives the accuracy of working of desgined model.

4.1 Usecases

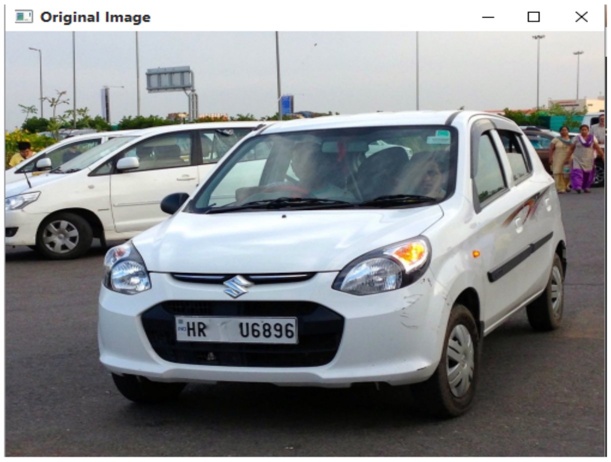
TEST CASE - 1 :

COMMAND PROMPT OUTPUT FOR THE EDGE POINTS THAT ARE DETECTED :

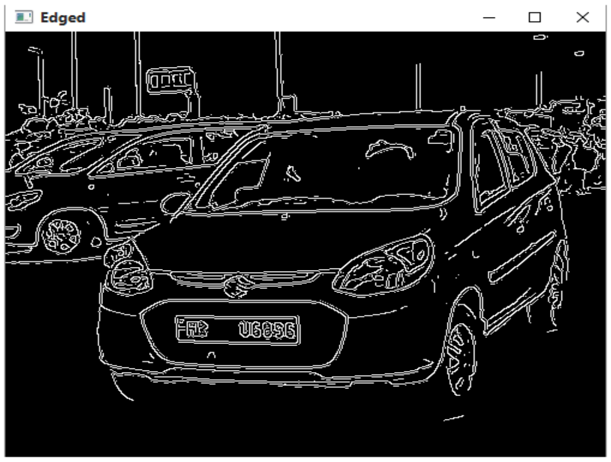


```
(base) C:\Users\user\Desktop>cd project

(base) C:\Users\user\Desktop\project>python main.py
[[[141 280]]

 [[240 279]]

 [[243 303]]

 [[145 302]]]

(base) C:\Users\user\Desktop\project>
```

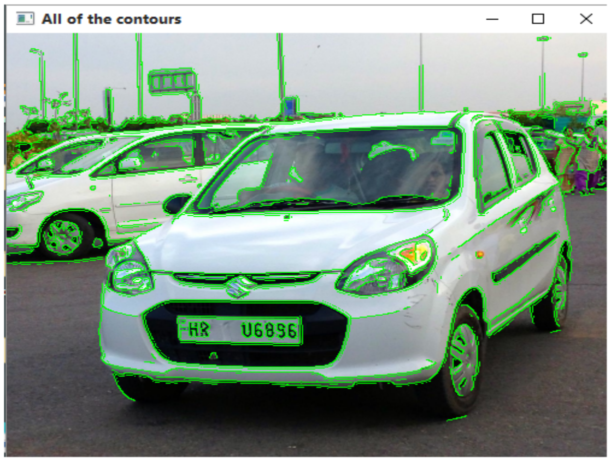
ORIGINAL IMAGE:



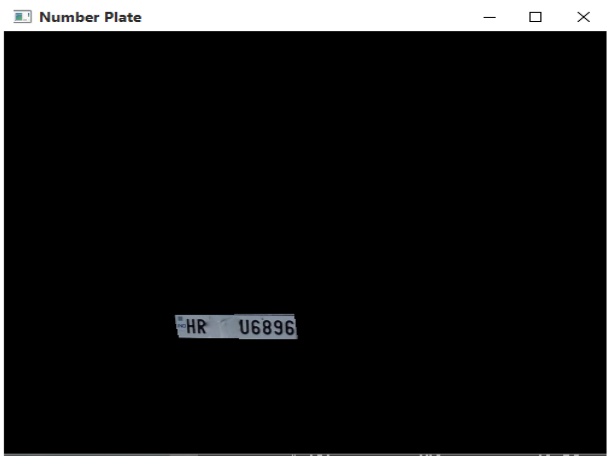
EDGES DETECTED IMAGE :



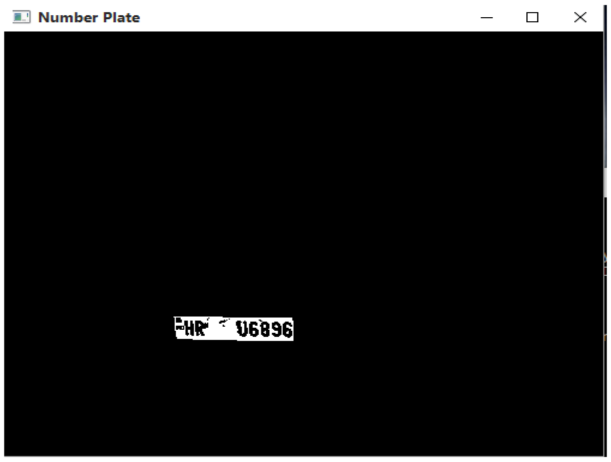
CONTOURS DETECTED IMAGE :



NUMBER PLATE DETECTED IMAGE :

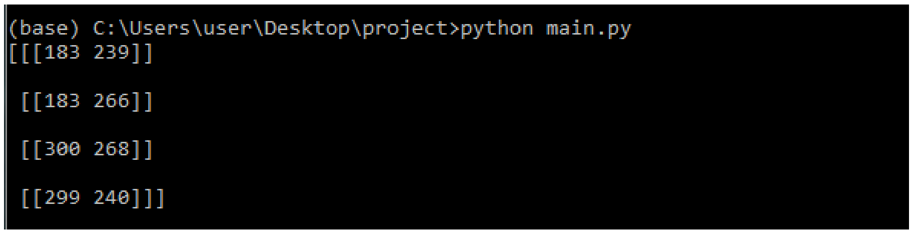


GRAYSCALE IMAGE OF NUMBERPLATE DETECTED :



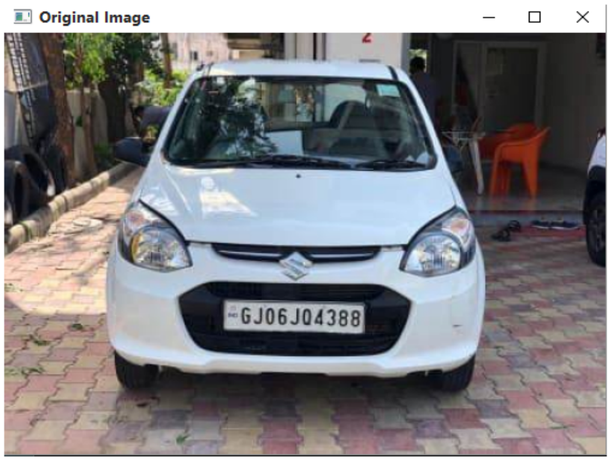
TEST CASE - 2 :

COMMAND PROMPT OUTPUT FOR THE EDGE POINTS THAT ARE DETECTED :

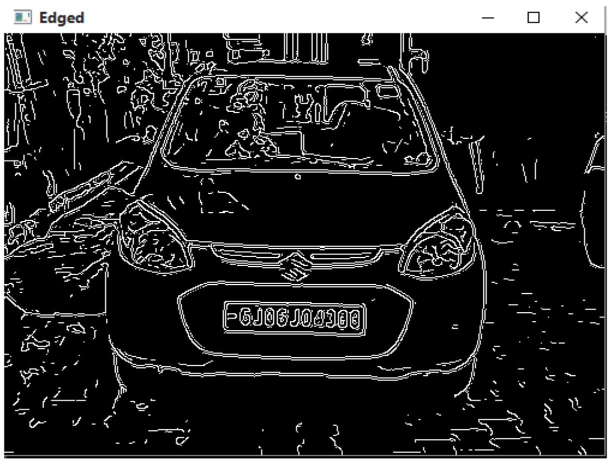


```
(base) C:\Users\user\Desktop\project>python main.py
[[[183 239]]

 [[183 266]]

 [[300 268]]

 [[299 240]]]
```

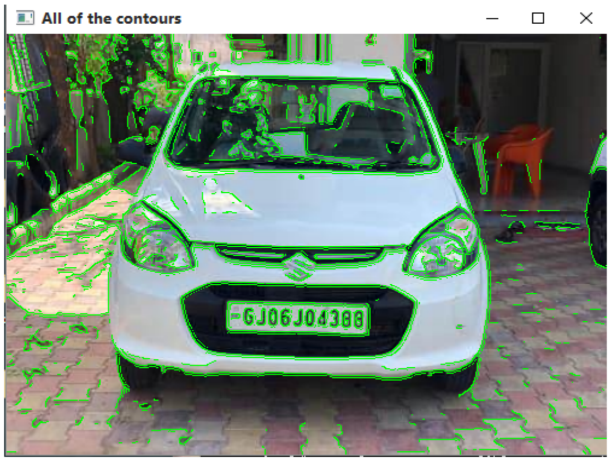
ORIGINAL IMAGE:



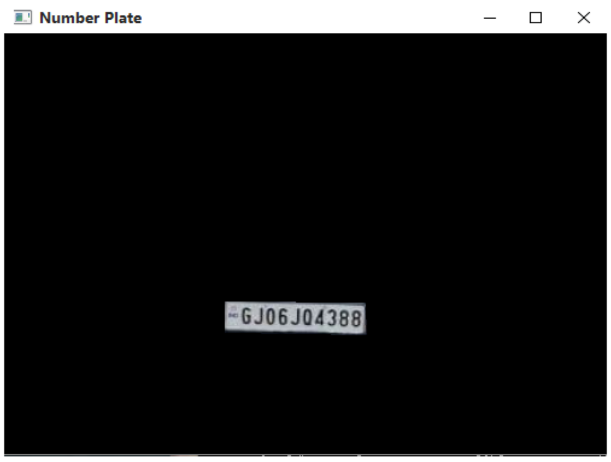
EDGES DETECTED IMAGE :



CONTOURS DETECTED IMAGE :

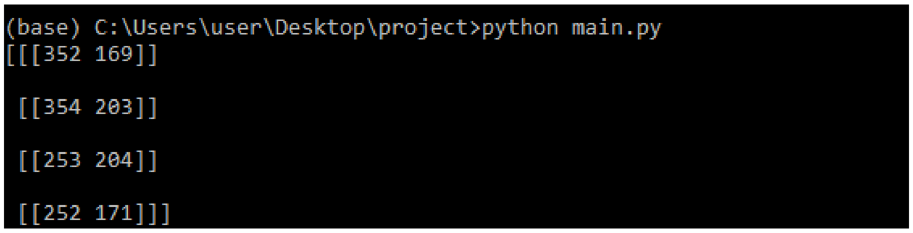


NUMBER PLATE DETECTED IMAGE :


Number Plate
GJ06JO4388

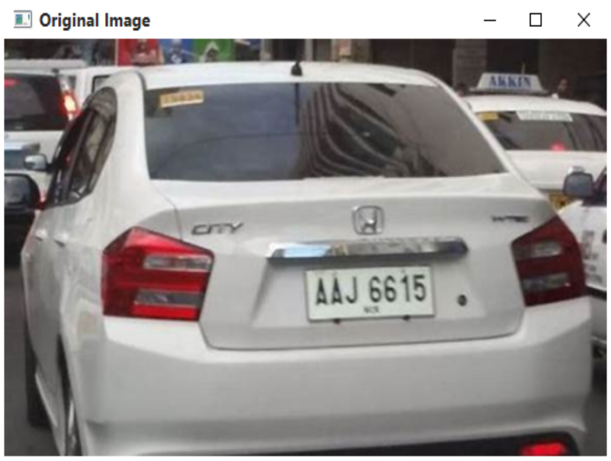
TEST CASE - 3 :

COMMAND PROMPT OUTPUT FOR THE EDGE POINTS THAT ARE DETECTED :

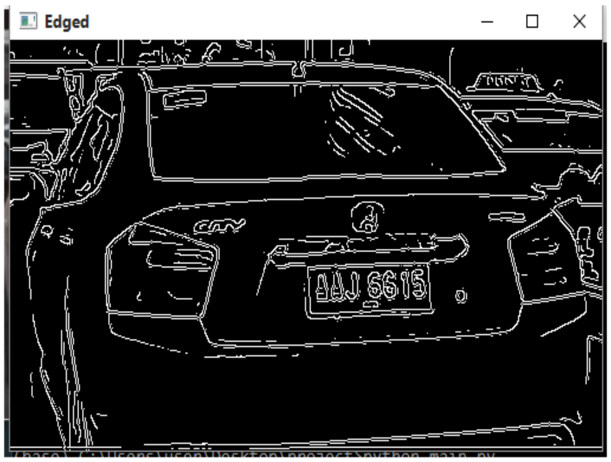


```
(base) C:\Users\user\Desktop\project>python main.py
[[[352 169]]

 [[354 203]]

 [[253 204]]

 [[252 171]]]
```

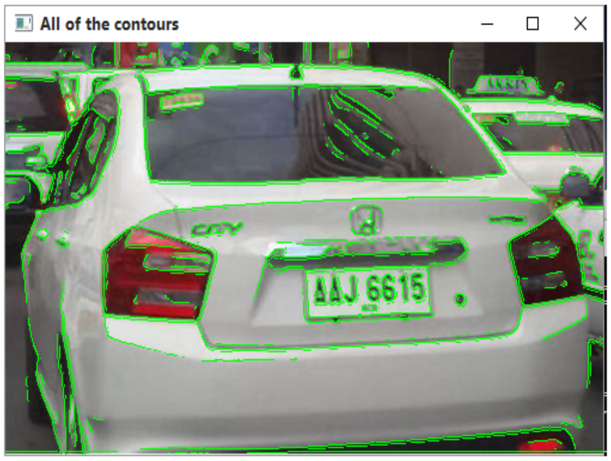
ORIGINAL IMAGE:



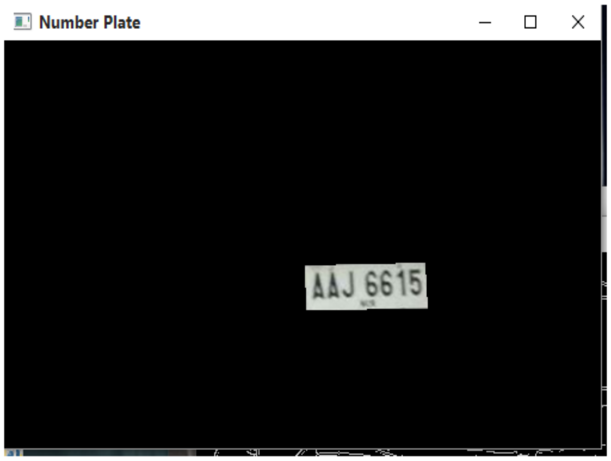
EDGES DETECTED IMAGE :



CONTOURS DETECTED IMAGE :



NUMBER PLATE DETECTED IMAGE :

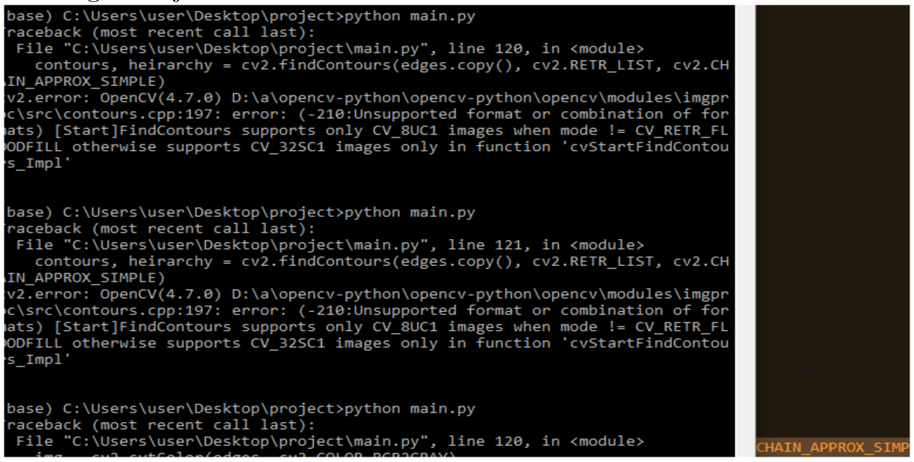


4.2 Acuuracy Measure

This model is executed and tested with many different inputs and resulted a accuracy of 79.82

4.3 Testing Results

It involves the process of development and error handling process.
Error Handling
Defining a maijor error caused and rectified:



```
base) C:\Users\user\Desktop\project>python main.py
raceback (most recent call last):
 File "C:\Users\user\Desktop\project\main.py", line 120, in <module>
   contours, heirarchy = cv2.findContours(edges.copy(), cv2.RETR_LIST, cv2.CH
IN_APPROX_SIMPLE)
v2.error: OpenCV(4.7.0) D:\a\opencv-python\opencv-python\opencv\modules\imgpr
c\src\contours.cpp:197: error: (-210:Unsupported format or combination of for
ats) [Start]FindContours supports only CV_8UC1 images when mode != CV_RETR_FL
ODFILL otherwise supports CV_32SC1 images only in function 'cvStartFindContou
s_Impl'


base) C:\Users\user\Desktop\project>python main.py
raceback (most recent call last):
 File "C:\Users\user\Desktop\project\main.py", line 121, in <module>
   contours, heirarchy = cv2.findContours(edges.copy(), cv2.RETR_LIST, cv2.CH
IN_APPROX_SIMPLE)
v2.error: OpenCV(4.7.0) D:\a\opencv-python\opencv-python\opencv\modules\imgpr
c\src\contours.cpp:197: error: (-210:Unsupported format or combination of for
ats) [Start]FindContours supports only CV_8UC1 images when mode != CV_RETR_FL
ODFILL otherwise supports CV_32SC1 images only in function 'cvStartFindContou
s_Impl'


base) C:\Users\user\Desktop\project>python main.py
raceback (most recent call last):
 File "C:\Users\user\Desktop\project\main.py", line 120, in <module>
```

```
CHAIN_APPROX_SIMP
```

Source, an 8-bit single-channel image. Non-zero pixels are treated as 1's. Zero pixels remain 0's, so the image is treated as binary . You can use compare() , inRange() , threshold() , adaptiveThreshold() , Canny() , and others to create a binary image out of a grayscale or color one. The function modifies the

image while extracting the contours. If mode equals to CV_RETR_CCOMP or $CV_RETR_FLOODFILL$, the input can also be a 32-bit integer image of labels (CV_32SC1).
Rectified by converting the image by ,
$img = np.uint8(edges * 255)$.

5 Conclusion Future Work

Number plate detection using OpenCV is a commonly used computer vision technique for extracting and recognizing license plates from images or video frames. The process involves several steps, including image preprocessing, contour detection, region of interest selection, and character segmentation.
OpenCV provides a set of powerful image processing functions that can be used to implement these steps and extract number plates accurately. Some of the commonly used functions in OpenCV for number plate detection include thresholding, morphological operations, edge detection, and contour detection. The accuracy of number plate detection using OpenCV can be affected by various factors such as lighting conditions, image resolution, and the complexity of the background. However, with proper tuning of the parameters and careful selection of the image processing techniques, OpenCV can achieve high accuracy and reliability in number plate detection.

Edge detection is a widely used image processing technique for detecting edges in an image. It was developed by John F. Canny in 1986 and is considered one of the most effective edge detection algorithms. The technique works by applying a series of image processing steps, including Gaussian smoothing, gradient calculation, non-maximum suppression, and hysteresis thresholding. The output of the Edge detection algorithm is a binary image where pixels are either labeled as edge pixels or non-edge pixels. The algorithm is very effective at detecting edges in images with low noise levels and high contrast.

In conclusion, number plate detection using OpenCV is a powerful and effective tool for automatic license plate recognition, which has numerous applications in areas such as traffic monitoring, law enforcement, and parking management.Edge detection is a powerful tool for detecting edges in images, and its effectiveness can be further enhanced by tuning its parameters to suit the specific application.

5.1 Future Work

There are several potential areas for future work in number plate detection using OpenCV, such as:

1. Improving accuracy: One area of future work could be to focus on improving the accuracy of number plate detection by exploring new image processing techniques or optimizing existing ones. For example, deep learning algorithms could be used in combination with OpenCV for more accurate detection and recognition of license plates.
2. Real-time performance: Another area of future work could be to improve the speed and real-time performance of number plate detection using OpenCV. This could involve optimizing algorithms and techniques to reduce computational complexity and processing time.

3. Adapting to different countries and regions: License plate formats can vary widely across different countries and regions, so another area of future work could be to adapt number plate detection algorithms to work with different formats. This could involve developing new algorithms or adapting existing ones to recognize different characters and formats.
4. Integration with other systems: Number plate detection using OpenCV could be integrated with other systems such as traffic management or law enforcement. Future work could focus on developing systems that integrate number plate detection with other technologies to provide more comprehensive solutions for specific applications.
5. Developing user-friendly interfaces: Another area of future work could be to develop user-friendly interfaces for number plate detection using OpenCV. This could involve developing software applications or mobile apps that allow users to easily capture and process images for license plate detection.

6 Bibiliography

References

[1] M. A. Hossain, S. M. H. Rizvi, and N. H. Noman, "License Plate Recognition Using OpenCV and Tesseract OCR Engine," *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pp. 1-6, 2018.

[2] G. He, F. Zhao, and D. Chen, "Automatic License Plate Recognition System Based on OpenCV," *2019 4th International Conference on Control and Robotics Engineering (ICCRE)*, pp. 212-215, 2019.

[3] R. M. Bhalerao and S. K. Kadam, "Vehicle Number Plate Recognition Using OpenCV and Tesseract OCR Engine," *2016 International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 165-168, 2016.

[4] F. J. Jin, L. X. Chen, and H. W. Chen, "License Plate Recognition System Based on OpenCV," *2017 2nd IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 471-476, 2017.

[5] S. B. Shetty and R. D. Reddy, "Automatic Number Plate Recognition using OpenCV and Haar Cascade Classifier," *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 990-995, 2017.