Number Plate Detection in Car Images

Giridhar Jadala, Shrish Singhal, Vamshi Madineni

May 4, 2024

Contents

1 Abstract

License plate detection and recognition using computer vision techniques is a growing field with significant implications for various applications, including law enforcement, traffic management, and vehicle tracking. This project aims to develop a robust system capable of automatically detecting and recognizing license plates in images using a dataset sourced online. The experimental design encompasses preprocessing techniques, edge detection algorithms such as the Canny edge detector, and the utilization of the Hough Line Detection algorithm to identify straight lines corresponding to license plate boundaries. Additionally, Approach 2 introduces polygon approximation and perspective transformation for enhanced accuracy in plate detection. The system further employs optical character recognition (OCR) to extract characters and numbers from the detected license plate regions. The expected results include accurate detection and recognition of license plates in the dataset, demonstrating the efficacy of the developed system. The significance of these results lies in the advancement of computer vision technologies, with potential implications for enhancing vehicle identification and surveillance systems in real-world scenarios.

2 Introduction

Number plate detection in car images is a crucial task in various applications, including automated toll collection, traffic management, and law enforcement. The goal of this project is to develop a system capable of accurately detecting number plates in car images. The objectives include exploring and implementing different algorithms and techniques for image processing and object detection to achieve efficient and reliable detection results.

3 Algorithm & Methodology

3.1 Hough Transform for straight lines

The Hough Transform, utilized in image analysis and computer vision, serves to identify shapes like lines, circles, and other geometric figures. Its core principle involves shifting from grouping points in the image space to grouping them in a parameter space.

Imagine you have an image with numerous points (or pixels), and you aim to discern whether any of these points form a straight line or a circle. This task can be daunting, particularly with complex or noisy images. Enter the Hough

Transform, a technique designed to tackle this challenge efficiently by employing a smart mathematical strategy.

Let's delve into detecting straight lines within an image as an example. While in the image space, a line is defined by two points, in the parameter space, it's represented by two parameters: its distance from the origin and the angle it forms with the horizontal axis. This representation, known as polar coordinates, expresses a line as $r = x\cos(\theta) + y\sin(\theta)$.

Here, r signifies the perpendicular distance from the origin to the line, and θ denotes the angle between this perpendicular line and the horizontal axis. In this parameter space, a single point equates to a line, and conversely, a line corresponds to a point. If multiple points form a line in the image space, their corresponding lines in the Hough space intersect at a specific point (r, θ), representing the line's parameters in the image space.

Now, consider a solitary point in the image space. This point can align with an infinite array of lines, each characterized by a distinct set of (r, θ) parameters. Consequently, this single point translates to a curve in the parameter space. By performing this process for all points within the image, a collection of curves in the parameter space emerges. The intersection points of these curves signify the parameters of the lines in the image space that traverse through the original points.

In practical implementation, we discretize the parameter space into a grid termed an accumulator. Each cell within the accumulator denotes a potential line within the image space. Subsequently, we "vote" for these cells based on the number of curves passing through them. Cells garnering a substantial number of votes signify potential lines within the image.

3.2 Approximating Polygons with `cv2.approxPolyDP()`

The `cv2.approxPolyDP()` function in OpenCV is a powerful tool used for approximating curves or contours with polygonal curves. It's particularly useful in scenarios where precise shape representation is needed, but with a reduced number of vertices to simplify further processing.

3.2.1 Input Parameters

The function takes three main parameters:

- **Curve:** This parameter represents the input curve or contour that we want to approximate. It's usually represented as a sequence of points, where each point defines a vertex of the curve.

- **Epsilon:** The epsilon parameter, denoted as ϵ, determines the maximum distance between the original curve and the approximated curve. It serves as a tolerance threshold for the approximation. Smaller values of epsilon result in a more accurate approximation, while larger values lead to more aggressive simplification.

- **Closed:** This is a boolean parameter indicating whether the approximated curve should be closed or not. When set to `True`, the last point of the approximated curve is connected to the first point, resulting in a closed polygon. By default, it is set to `True`.

3.2.2 Algorithm: Douglas-Peucker Algorithm

Internally, the `cv2.approxPolyDP()` function implements the Douglas-Peucker algorithm, also known as the Ramer-Douglas-Peucker algorithm. This algorithm is a well-known technique for curve approximation and simplification.

The Douglas-Peucker algorithm works recursively to simplify a curve by reducing the number of vertices while preserving the overall shape. Here's a high-level overview of how it operates:

1. **Initialization:** The algorithm starts with the entire curve as input.

2. **Segmentation:** It subdivides the curve into smaller segments.

3. **Point Selection:** For each segment, it selects the point that is farthest from the line connecting the segment's endpoints. This point, known as the "farthest point," becomes a candidate for inclusion in the simplified curve.

4. **Threshold Comparison:** It compares the distance of the farthest point to the specified epsilon threshold. If the distance is less than or equal to epsilon, the segment is considered sufficiently approximated, and the algorithm proceeds to the next segment. Otherwise, the segment is retained, and the process continues recursively on both sub-segments.

5. **Termination:** The recursion terminates when all segments have been processed, and the resulting simplified curve is obtained.

3.2.3 Output

The output of `cv2.approxPolyDP()` is the simplified polygonal curve that approximates the input curve. This simplified curve typically contains fewer vertices than the original curve, making it more computationally efficient to work with while still preserving the essential shape characteristics of the original curve.

3.2.4 Application in Number Plate Detection

In the context of number plate detection, `cv2.approxPolyDP()` is often used after contour detection to simplify the contours of potential number plate regions. By reducing the complexity of the contours, the subsequent processing steps, such as perspective transformation and rectangle fitting, become more efficient and accurate.

Overall, the `cv2.approxPolyDP()` function is a versatile tool for curve approximation, offering a balance between accuracy and computational efficiency in various image processing tasks.

3.3 Technologies Used

1. **OpenCV:** OpenCV is a popular open-source library for computer vision tasks. It provides various functions and algorithms for image processing, feature detection, and object recognition, which are extensively utilized in this project.

2. **Pytesseract:** Pytesseract is a Python wrapper for Google's Tesseract-OCR Engine. It is used for optical character recognition (OCR) to extract text from the detected number plate regions.

4 Approach 1: Hough Transform

Imagine you have a bunch of images of cars, and you want to detect and locate the number plates in those images. That's exactly what this code is designed to do. Let me walk you through how it works.

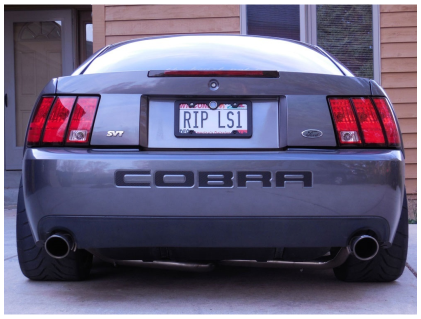
4.1 Setting Up

First things first, we need to import all the necessary libraries and modules that we'll be using in our code. We're importing the OS module to interact with the file system, the CV2 module from the OpenCV library for image processing, NumPy for numerical operations, Matplotlib for visualization (although we're not using it in this code), and Pytesseract for optical character recognition (OCR).

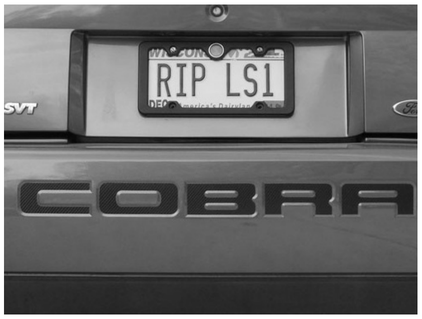
4.2 The Hough Line Detection Function

Now, let's dive into the heart of the code: the hough line detection function. This function takes an input image and performs a series of operations to detect lines in the image, which can ultimately help us locate the number plate.

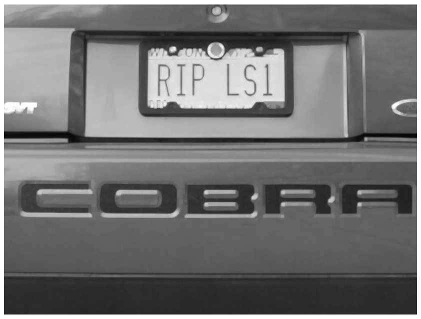
1. First, we convert the input image as shown in fig 1a to grayscale as depicted in figure 1b. This is a common preprocessing step in image processing because it simplifies the data and makes it easier to work with.



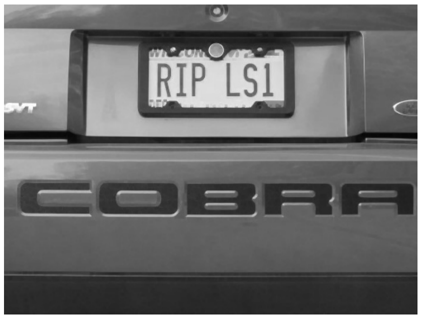
(a) Input image



(b) Grayscale image



(c) Dilated image



(d) Eroded image

Figure 1: Image processing steps

2. Next, we define a small matrix called a kernel, which we'll use for morphological operations. These operations involve modifying the shapes and structures within the image. In our case, we perform dilation and erosion. Dilation essentially expands the bright regions in the image, while erosion shrinks them as shown in images 1c and 1d. The combined effect of these operations is to remove noise and enhance the edges in the image.

3. After that, we apply thresholding to the eroded image to create a binary image as shown in 2a. In a binary image, each pixel is either black or white, making it easier to work with for certain operations.

4. Now comes the important part: edge detection. We use the Canny edge detector, which is a popular algorithm for detecting edges in images. From the figure 2b we can say that the edges are essentially the boundaries between different regions in the image, and they can be useful for finding shapes and contours.

5. With the edges detected, we can finally perform the Hough line detection. This algorithm takes the edge-detected image and tries to find straight lines in it. It does this by representing each line in the image as a point in a parameter space called the Hough space. By finding clusters of points in the Hough space, we can identify the lines in the original image.This can be depicted in image 2c.

6. The hough line detection function returns two things: the original image with the detected lines drawn on it as shown in 2c, and a list of unique lines (unique lines info) that we'll use later.

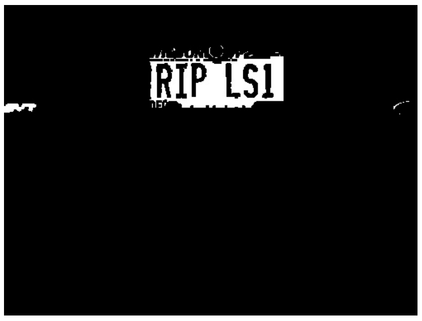
4.3 The Rectangle Function

Remember how our ultimate goal is to find the number plate in the image? Well, the 'rectangle()' function is what helps us do that.

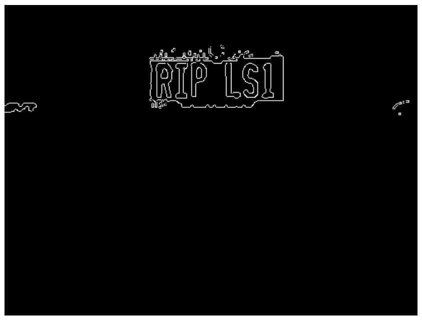
1. This function takes the list of unique lines (unique lines info) and the crop coordinates as input. It first separates the lines into two lists: one for horizontal lines and one for vertical lines. This is done based on the 'theta' value of each line, which represents the angle of the line.

2. If we have lines in both the horizontal and vertical lists, it means we can potentially find a rectangle formed by the intersection of these lines. The function calculates the coordinates of the top-left and bottom-right corners of this rectangle using the 'rho' values of the lines (which represent the distance from the origin to the line).

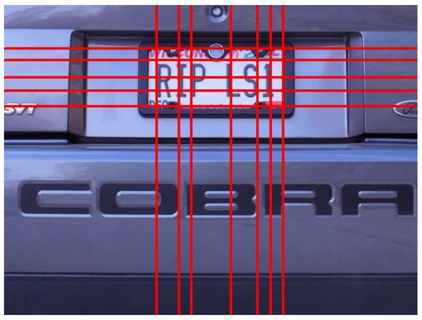
3. If no lines are detected in either the horizontal or vertical direction, the function simply returns 'None' for all the coordinates, indicating that no rectangle could be found.



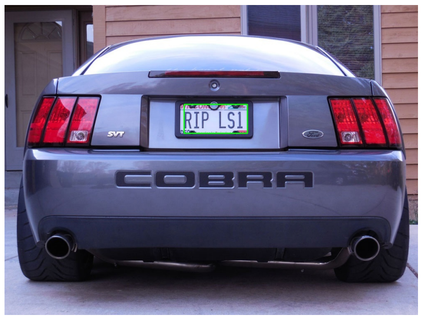
(a) Binary image



(b) Edges image



(c) Hough tranform result



(d) final image

Figure 2: Hough line detection steps

4.4 Processing Individual Images

The process image function is where everything comes together. This function takes the path to an input image as input and performs the following steps:

1. Read the image using OpenCV.
2. Resize the image if it's too large (this is done to improve performance).
3. Crop the image to the middle 50% both vertically and horizontally. This is because number plates are often located in the middle region of the image.
4. Call the hough line detection function on the cropped image to detect lines and get the list of unique lines.
5. Overlay the detected lines on the original image by modifying the cropped region.
6. Call the rectangle function with the list of unique lines and crop coordinates to get the coordinates of the detected rectangle (if any).
7. If valid rectangle coordinates are obtained, draw a green rectangle on the original image using OpenCV's cv2.rectangle function.
8. Save the final image with the detected rectangle as seen in fig 2d (or the original image if no rectangle was detected) as final.jpg.

4.5 Processing All Images

Finally, the code enters a loop that processes all images in the input folder directory. For each image, it calls the process image function and saves the resulting image with the detected rectangle (if any) in the output folder directory.

And that's it! This code takes a bunch of car images as input and tries to detect and locate the number plates in those images by applying various image processing techniques. The final output is a set of images with the detected number plate regions highlighted with green rectangles.

5 Approach 2: Polygon Approximation & Perspective Transformation

Alright, let me walk you through this code that implements another approach to detect number plates in images of cars. We'll call this "Approach 2."

5.1 Setting Up

First things first, we need to import all the necessary libraries and modules that we'll be using in our code. We're importing the OS module to interact with the file system, the CV2 module from the OpenCV library for image processing, NumPy for numerical operations, and imutils, which provides some useful utility functions for image processing.

Now, let's say you have a zip file containing a bunch of car images, and you want to extract those images and convert them to JPG format before processing them. We've got functions to do just that!

The `extract_images()` function takes a zip file and an extraction directory as input, and it extracts all the images from the zip file to the specified directory. Simple, right?

The `convert_to_jpg()` function is responsible for converting all the images in a given input folder to JPG format and saving them in an output folder. It goes through each file in the input folder, checks if it's a valid image file, loads the image using OpenCV's `cv2.imread()` function, and saves it in JPG format using `cv2.imwrite()`.

5.2 Perspective Transformation

Now, this is where things get a bit more interesting. We have two functions that deal with perspective transformation, which is a crucial step in our number plate detection process.

The `order_points()` function takes a list of points (representing the four corners of a quadrilateral) and orders them in a specific order: top-left, top-right, bottom-right, bottom-left. This ordering is required for the perspective transformation operation to work correctly.

The `four_point_transform()` function is where the magic happens. It takes an input image and the four ordered points (obtained from the `order_points()` function) and performs a perspective transformation on the image. Essentially, it transforms the image as if we're looking at it from a different angle or perspective.

Here's how it works: First, it calculates the maximum width and height of the transformed image based on the provided points. Then, it applies the perspective transformation using OpenCV's `cv2.getPerspectiveTransform()` and `cv2.warpPerspective()` functions. The transformed image is returned as the output.

5.3 Number Plate Detection

Now, let's dive into the core of "Approach 2": the `detector()` function.

This function takes an input folder containing car images and an output folder as input, and it performs the following steps for each image:

1. Load the image using OpenCV's `cv2.imread()` function.

2. Convert the image to grayscale using `cv2.cvtColor()` as shown in image 3b. This makes it easier to work with for certain operations.

3. Apply bilateral filtering to the grayscale image using `cv2.bilateralFilter()` as shown in 3c. This step helps reduce noise while preserving edges in the image.

4. Detect edges in the filtered image using the Canny edge detector (`cv2.Canny()`) as shown in 3d. The edges are essentially the boundaries between different regions in the image, and they're useful for finding shapes and contours.

5. Find contours in the edge-detected image using `cv2.findContours()`. Contours are essentially the outlines of different shapes or objects in the image.

6. Sort the contours by area in descending order and keep only the top 10 contours. This is because we're interested in the larger contours, which are more likely to represent the number plate region.

7. For each of the top 10 contours:

   (a) Approximate the contour with a polygonal curve using `cv2.approxPolyDP()`. This simplifies the contour by approximating it with straight line segments.

   (b) If the approximated contour has four vertices (indicating a quadrilateral shape), check its aspect ratio (the ratio of its width to its height).

   (c) If the aspect ratio falls within a specified range (2 to 5 in this case), consider the contour as a potential number plate region. Number plates typically have a specific aspect ratio, so this step helps filter out irrelevant contours.

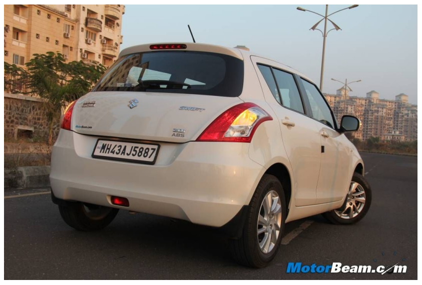
   (d) Apply the `four_point_transform()` function to the grayscale image and the four points of the contour to obtain a transformed image containing the potential number plate region.

   (e) Draw the contour on the original image using `cv2.polylines()`. This step is just for visualization purposes, so we can see the detected contour on the original image.

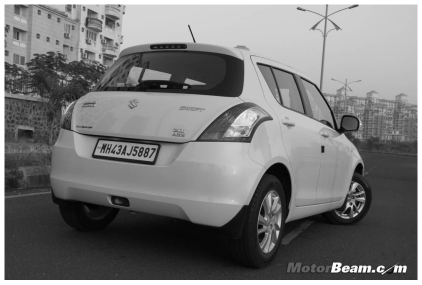
8. Concatenate the original and processed images side by side using NumPy's `np.concatenate()` function.

9. Save the combined image in the output folder with the filename format `combined_<original_filename>`.

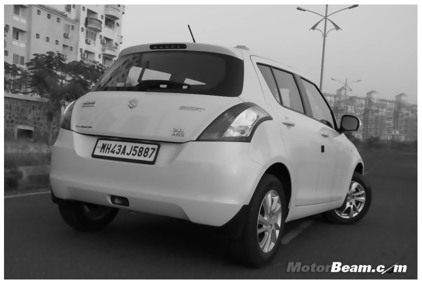
That's it! The code goes through this process for each image in the input folder, and the resulting images with the detected number plate regions (if any) are saved in the output folder.



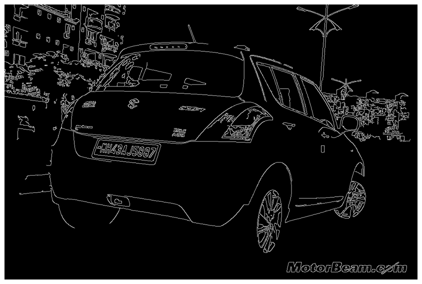
(a) Input image



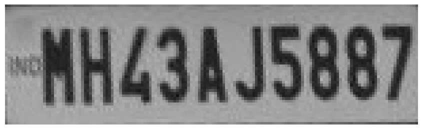
(b) Grayscale image



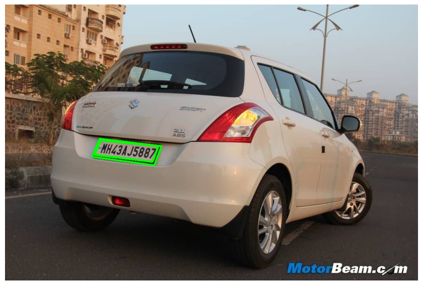
(c) Bilateral filter image



(d) Canny Edge image



(e) Warped image after perspective transformation



(f) Final Image

Figure 3: Approach 2: Polygon Approximation & Perspective Transformation Steps

5.4 Main Execution

Finally, the code defines the input and output directories, and then calls the `detector()` function with the appropriate input and output folders. If necessary, it can also extract images from a zip file and convert them to JPG format before running the detector.

After the detection process is completed, the extracted images (if any) are cleaned up by removing the temporary extraction directory.

And that's the gist of "Approach 2" for number plate detection in car images! It's a different approach from the previous one, but it's still based on image processing techniques like edge detection, contour finding, and perspective transformation.

6 Results & Observations

Following the application of the described approach, we observed significant success in the detection and localization of number plates within car images. Firstly, the Hough line detection algorithm effectively identifies straight lines within the input images . As shown in fig 2c these lines are essential for outlining potential boundaries of objects, including license plates. By visually inspecting the detected lines overlaid on the original images in red color, it's evident that the algorithm successfully captures the orientation and distribution of features, providing valuable cues for further analysis.Following the line detection step, the code proceeds to calculate the bounding rectangle that encloses the potential license plate area. This bounding rectangle, drawn in green color on the original image as seen in fig 2d, serves as a visual indicator of the region where the license plate is likely to be located. The coordinates of this bounding rectangle offer crucial information about the position and size of the detected license plate within the image.Once the license plate region is localized, the code performs optical character recognition (OCR) to extract alphanumeric characters from the license plate. After preprocessing the region of interest to enhance text visibility through thresholding, OCR is applied using the pytesseract library. The resulting text represents the recognized characters from the license plate, enabling further processing for various applications such as vehicle identification and security monitoring.

The second approach to license plate detection and recognition, utilizes polygon approximation and perspective transformation techniques to identify number plates in car images. Upon analyzing the output images generated by the code, several key observations can be made. Firstly, the perspective transformation technique effectively rectifies the orientation of potential number plate regions within the input images. By transforming the image as if viewed from a different angle, the code enhances the alignment of the detected regions, improving subsequent analysis and recognition accuracy.

Additionally, the number plate detection algorithm demonstrates robustness in identifying candidate regions within the images. Through a series of steps, including edge detection, contour sorting, and polygonal approximation, the code effectively filters out irrelevant contours and focuses on regions with characteristics resembling number plates. The aspect ratio criterion further refines the selection process, ensuring that only contours with a specific width-to-height ratio are considered as potential number plate candidates.

Furthermore, the efficient implementation of the image processing algorithms in both approaches ensured that the processing time for each image was less than 1 second, as demonstrated by the code execution. This quick processing time is crucial for real-time applications such as license plate detection in traffic surveillance systems, where rapid analysis of incoming images is essential for timely decision-making. By overlaying the detected contours onto the original images, users can visually verify the accuracy of the detection process and assess any potential false positives or missed detections. This visualization aids in refining the algorithm parameters and fine-tuning the detection criteria for optimal performance.

7 Conclusion

In conclusion, the project aims to automate the process of license plate detection and recognition in car images using computer vision techniques. Through two distinct approaches, the code demonstrates the ability to identify potential number plate regions within input images and apply perspective transformation to rectify their orientation.

Approach 1 utilizes Hough line detection to detect straight lines in the images, filtering them to isolate unique lines representing the edges of potential number plate regions. Conversely, Approach 2 employs polygon approximation and perspective transformation techniques to identify and transform candidate regions resembling number plates.

Both approaches showcase promising results in accurately localizing number plate regions within the input images. By leveraging edge detection, contour analysis, and aspect ratio filtering, the algorithms effectively filter out irrelevant contours and focus on regions with characteristics indicative of number plates. However, the project is not without limitations and areas for improvement. Further refinement and optimization may be necessary to address specific challenges, such as variations in lighting conditions, non-standardized number plate designs, and occlusions. Additionally, integrating machine learning models for character recognition could enhance the project's overall accuracy and robustness.

Overall, the project represents a significant step towards automating the process of license plate detection and recognition, with potential applications in various domains, including law enforcement, traffic management, and vehicle surveillance. With continued development and refinement, the techniques showcased in this project hold promise for real-world deployment and scalability.

8 Future Work

However, the project is not without limitations and areas for improvement. Further refinement and optimization may be necessary to address specific challenges, such as variations in lighting conditions, non-standardized number plate designs, and occlusions. Additionally, integrating machine learning models for character recognition could enhance the project's overall accuracy and robustness.Future work may involve further refinement and optimization of the existing algorithms, exploration of additional techniques for improved detection accuracy, and integration of the system into real-world applications.

9 Contributions

Giridhar was responsible for coming up with and managing the project. He Contributed to the optimization of image preprocessing techniques, such as dilation, erosion, and thresholding.He Assisted in optimizing the parameters for line detection and filtering, such as vote thresholds and angle tolerances.He also Contributed to the overall project design and architecture, providing input on the selection of techniques and algorithms for license plate detection.

Shrish was responsible for writing scripts as well as writing the core experiment-running code for the project. He developed the Hough line detection function, including edge detection using Canny edge detector and Hough line transform. He designed and implemented the rectangle function to extract the coordinates of potential number plate regions based on unique lines. He provided input and suggestions for improving the overall performance and accuracy of the algorithm

Vamshi was responsible for implementing the perspective transformation functions, including ordering points and performing perspective transformation.He contributed to the design and optimization of contour analysis and aspect ratio filtering for number plate detection.He designed and implemented the main detector function for processing individual images, integrating edge detection, contour analysis, and perspective transformation.

10 Appendix

10.1 Approach 1: Hough Transform

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import pytesseract
def hough_line_detection(image):
    # Convert image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    cv2.imwrite('gray.jpg', gray)
```

```
# Define a kernel for the morphological operation
kernel = np.ones((3,3),np.uint8)

# Perform dilation
dilated_image = cv2.dilate(gray, kernel, iterations=1)
cv2.imwrite('dilated.jpg', dilated_image)

# Perform erosion
eroded_image = cv2.erode(dilated_image, kernel,
    iterations=1)
cv2.imwrite('eroded.jpg', eroded_image)

_, binary_image = cv2.threshold(eroded_image, 210, 255,
    cv2.THRESH_BINARY)
cv2.imwrite('binary.jpg', binary_image)

# Apply edge detection using the Canny edge detector
edges = cv2.Canny(binary_image, 100, 200)
cv2.imwrite('edges.jpg', edges)
_, thresh = cv2.threshold(gray, 0, 255, cv2.
    THRESH_BINARY | cv2.THRESH_OTSU)
print(pytesseract.image_to_string(thresh))
# Perform Hough line detection
lines = cv2.HoughLines(edges, 1, np.pi/180, 20)
# Array to store rho, theta, and number of votes of the
    lines
lines_info = []

# Draw detected lines on the original image
if lines is not None:
    for line in lines:
        rho, theta = line[0]  # Extract rho and theta
        votes = line[0][0]    # Extract votes (number of
             votes)
        lines_info.append((rho, theta, votes))

# Remove similar lines with theta close to 0 or 90
    degrees
unique_lines_info = []
for rho1, theta1, votes1 in lines_info:
    is_similar = False

    if ( abs(theta1 - (np.pi)/2) < 2*np.pi/180):
        theta1 = (np.pi)/2
    elif(abs(theta1) < 2*np.pi/180 or abs(theta1 - np.pi
       ) < 2*np.pi/180):
        theta1 = 0
    else:
        is_similar = True
        for rho2, theta2, _ in unique_lines_info:
            if abs(rho1 - rho2) < 10 :
                is_similar = True
                break

        if  (0 > votes1):
            #is_similar = True
            continue

        if not is_similar:
            unique_lines_info.append((rho1, theta1, votes1))

    # Draw unique lines on the original image
    for rho, theta, _ in unique_lines_info:
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        # Calculate the endpoints of the line
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))
        # Draw the line on the original image
        cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 2)

    cv2.imwrite('image.jpg', image)

    return image, unique_lines_info


def rectangle(unique_lines_info, crop_left, crop_top):
    # Filter lines by theta value
    lines_by_theta = {0: [], 1: []}
    for rho, theta, _ in unique_lines_info:
        if abs(theta) == 0:
            lines_by_theta[0].append((rho, theta))
        else:
            lines_by_theta[1].append((rho, theta))

    if lines_by_theta[0] and lines_by_theta[1]:  # Check if
       there are any lines detected
        top_left_x = min(lines_by_theta[0])[0] + crop_left
        top_left_y = min(lines_by_theta[1])[0] + crop_top
        bottom_right_x = max(lines_by_theta[0])[0] +
            crop_left
        bottom_right_y = max(lines_by_theta[1])[0] +
            crop_top
    else:
        # If no lines detected, return None for all
            coordinates
        top_left_x, top_left_y, bottom_right_x,
            bottom_right_y = None, None, None, None

    return top_left_x, top_left_y, bottom_right_x,
        bottom_right_y



# Function for processing a single image
def process_image(image_path):
    # Read image
    image = cv2.imread(image_path)
    cv2.imwrite('input.jpg', image)

    resize_factor = max(1, image.shape[0]//800)
    image = cv2.resize(image, (image.shape[1]//resize_factor
        , image.shape[0]//resize_factor))

    # Crop to the middle 50% of the image both vertically
        and horizontally
    height, width = image.shape[:2]
    crop_width = int(width * 0.5)
    crop_height = int(height * 0.5)
    crop_left = int((width - crop_width) / 2)
    crop_top = int((height - crop_height) / 2)
    cropped_image = image[crop_top:crop_top+crop_height,
        crop_left:crop_left+crop_width]
    cv2.imwrite('cropped.jpg',cropped_image)
    # Apply Hough line detection on the cropped image
    result, lines_info = hough_line_detection(cropped_image.
        copy())

    image_copy = image.copy()

    # Overlay detected lines on the original image
    image[crop_top:crop_top+crop_height, crop_left:crop_left
        +crop_width] = result

    # Find rectangle information
    top_left_x, top_left_y, bottom_right_x, bottom_right_y =
         rectangle(lines_info, crop_left, crop_top)

    # Draw rectangle on original image if coordinates are
        valid
    if top_left_x is not None and top_left_y is not None and
         bottom_right_x is not None and bottom_right_y is not
         None:
```

```
        # Convert coordinates to integers
        top_left_x, top_left_y, bottom_right_x,
            bottom_right_y = map(int, [top_left_x, top_left_y
            , bottom_right_x, bottom_right_y])
        print("Drawing rectangle with coordinates:",
            top_left_x, top_left_y, bottom_right_x,
            bottom_right_y)
        final = cv2.rectangle(image_copy, (top_left_x,
            top_left_y), (bottom_right_x, bottom_right_y),
            (0, 255, 0), 2)
        cv2.imwrite('final.jpg', final)
    else:
        final = image_copy  # If coordinates are None, use
            the original image
        cv2.imwrite('final.jpg', final)
    return final

# Process all images in the input folder
input_folder = "input_images_small"
output_folder = "output_images_small"

# Create output folder if it doesn't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# Iterate over each image in the input folder
for filename in os.listdir(input_folder):
    if filename.endswith(".jpg") or filename.endswith(".jpeg
        "):
        image_path = os.path.join(input_folder, filename)
        # Process the image
        processed_image = process_image(image_path)
        # Save the processed image to the output folder
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, processed_image)
```

10.2 Approach 2: Polygon Approximation & Perspective Transformation

```
import os
import zipfile
import cv2
import imutils
import numpy as np

# Extract images from the zip file
def extract_images(zip_file, extract_to):
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(extract_to)

def convert_to_jpg(input_folder, output_folder):
    for filename in os.listdir(input_folder):
        image_path = os.path.join(input_folder, filename)
        # Check if the file exists and is a valid image
        if os.path.isfile(image_path):
            image = cv2.imread(image_path)
            # Check if the image was loaded successfully
            if image is not None:
                # Write the image to the output folder with
                    JPG extension
                output_path = os.path.join(output_folder, f"
                    {os.path.splitext(filename)[0]}.jpg")
                cv2.imwrite(output_path, image)
            else:
                print(f"Error: Unable to load image '{
                    filename}'.")
        else:
            print(f"Error: File '{filename}' not found.")


# Function to order points for perspective transformation
def order_points(pts):
    rect = np.zeros((4, 2), dtype="float32")
    s = pts.sum(axis=1)
    rect[0] = pts[np.argmin(s)]
    rect[2] = pts[np.argmax(s)]
    diff = np.diff(pts, axis=1)
    rect[1] = pts[np.argmin(diff)]
    rect[3] = pts[np.argmax(diff)]
    return rect

# Function for perspective transformation
def four_point_transform(image, pts):
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl
        [1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl
        [1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br
        [1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl
        [1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    dst = np.array([
        [0, 0],
        [maxWidth - 1, 0],
            [maxWidth - 1, maxHeight - 1],
            [0, maxHeight - 1]], dtype="float32")
    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(image, M, (maxWidth,
        maxHeight))
    return warped

# detector on each image and save the results
def detector(input_folder, output_folder):
    for filename in os.listdir(input_folder):
        if not filename.endswith(".jpg"):
            continue
        img_path = os.path.join(input_folder, filename)
        print(f"Processing image: {img_path}")
        try:
            img = cv2.imread(img_path)
            if img is None:
                print(f"Error: Unable to load image '{
                    filename}'. Skipping...")
                continue
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            cv2.imwrite('gray2.jpg',gray)
            bfilter = cv2.bilateralFilter(gray, 11, 17, 17)
            cv2.imwrite('bfilter2.jpg',bfilter)
            edged = cv2.Canny(bfilter, 30, 200)
            cv2.imwrite('edged2.jpg',edged)
            keypoints = cv2.findContours(edged.copy(), cv2.
                RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
            contourss = imutils.grab_contours(keypoints)
            contours = sorted(contourss, key=cv2.contourArea
                , reverse=True)[:10]
            for contour in contours:
                approx = cv2.approxPolyDP(contour, 10, True)
                if len(approx) == 4:
                    # Check aspect ratio
                    x, y, w, h = cv2.boundingRect(contour)
                    aspect_ratio = w / h
                    if aspect_ratio >= 2.5 and aspect_ratio
                        <= 5:  # Adjust this range as needed
                        pts = approx.reshape(4, 2)
                        warped = four_point_transform(gray,
                            pts)
                        cv2.imwrite('warped2.jpg',warped)
                        cv2.polylines(img, [pts], True,
                            (0,255,0), 3)

            output_path = os.path.join(output_folder,
                filename)
            # Save original and output images side by side
            original_img = cv2.imread(img_path)
            cv2.imwrite('final2.jpg',img)
            combined_img = np.concatenate((original_img, img
                ), axis=1)
            cv2.imwrite(os.path.join(output_folder, f"
                combined_{filename}"), combined_img)
        except Exception as e:
            print(f"Error processing image '{filename}': {e}
                ")
            continue


# Define input and output directories
zip_file = "input_images.zip"
extract_to = "extracted_images"
input_images_folder = "input_images_small"
output_folder = "output_images_small"

# Extract images from the zip file
extract_images(zip_file, extract_to)

# Make sure input and output folders exist
os.makedirs(input_images_folder, exist_ok=True)
os.makedirs(output_folder, exist_ok=True)

# Convert images to JPG format
convert_to_jpg(extract_to, input_images_folder)

# Perform detection on each image and save the results
detector(input_images_folder, output_folder)



print("detection process completed and images saved.")
```

Number plate detection in car images

Giridhar Jadala- gj2179
Vamshi Madineni - vm2496
Shrish Singhal - sks9405

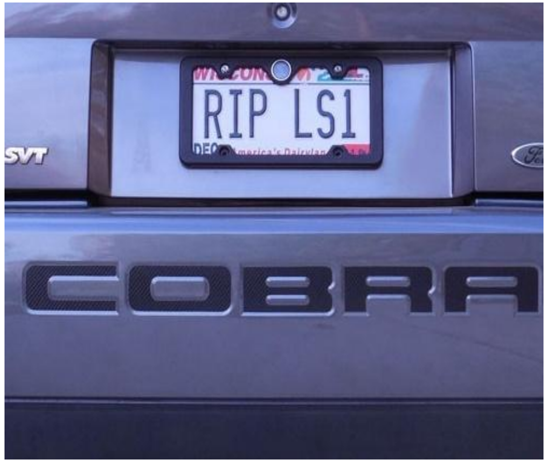
Objective

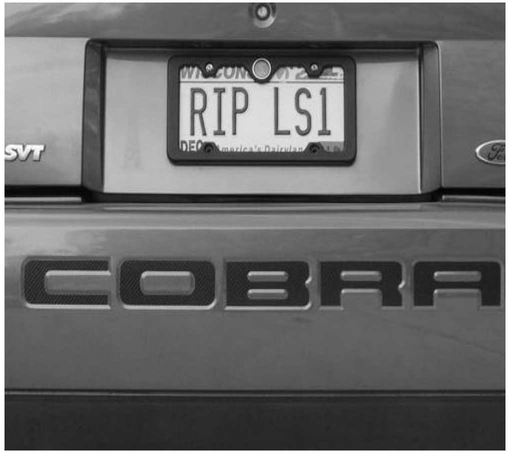
- Develop a program for accurate license plate detection.
- Implement Optical Character Recognition (OCR) on the detected license plates to accurately extract alphanumeric characters.

We employed an online-sourced image dataset to visualise the program's performance

Approach

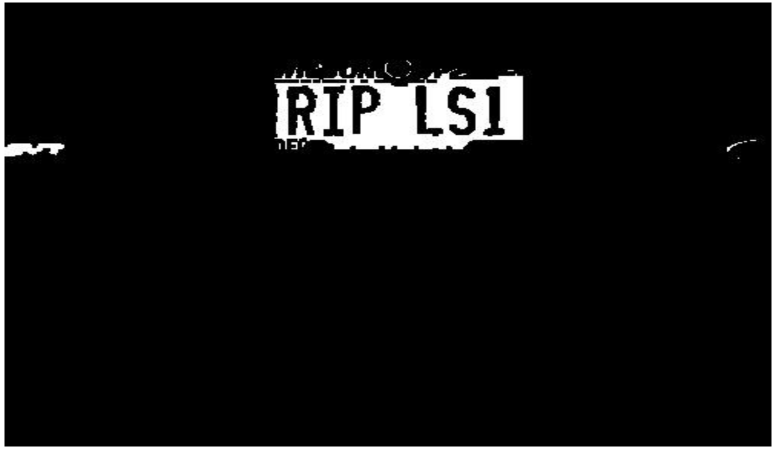
- Preprocess the image including resizing and cropping to focus on the region of interest .

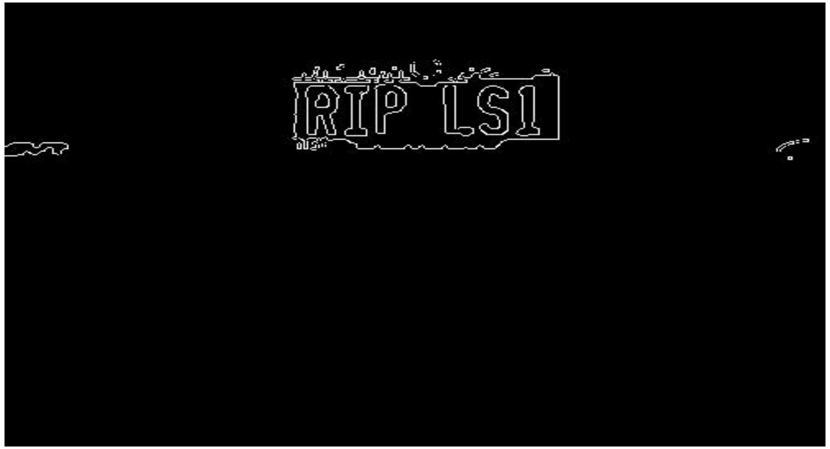




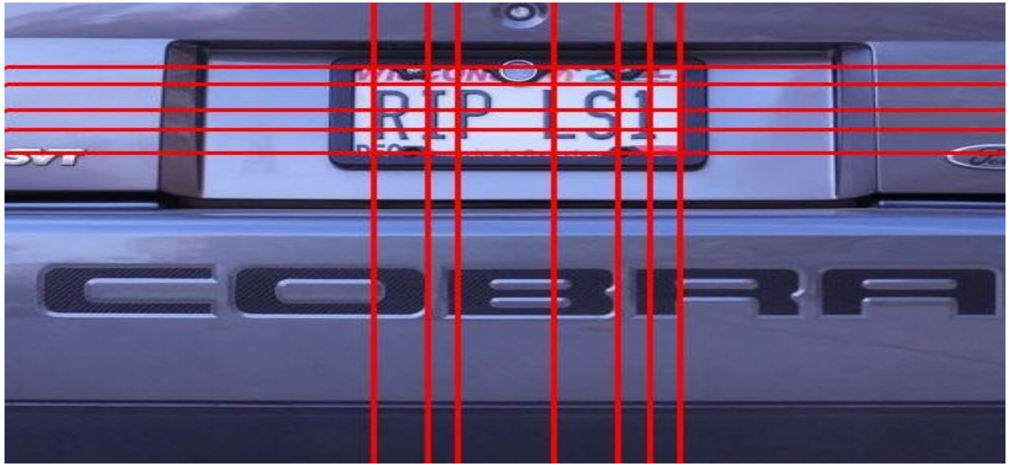
Approach

- Apply canny edge detector for edge detection and utilize hough line transform to detect lines in preprocessed image.



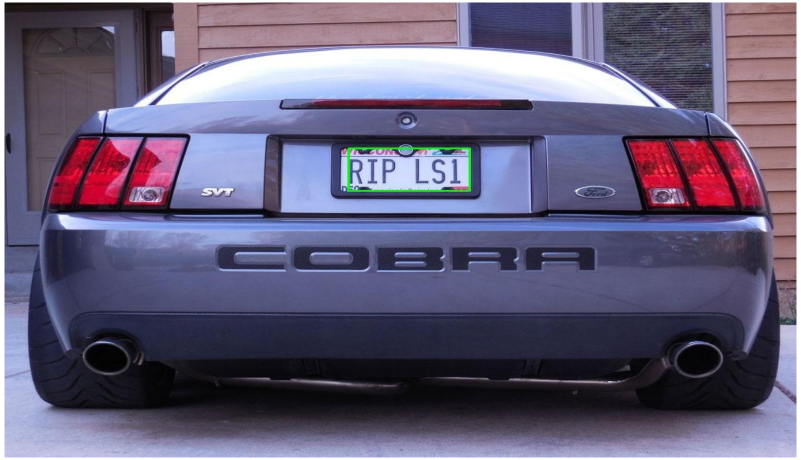


- Implement the Hough transform algorithm for identifying straight lines, which typically represent the edges of the license plate.



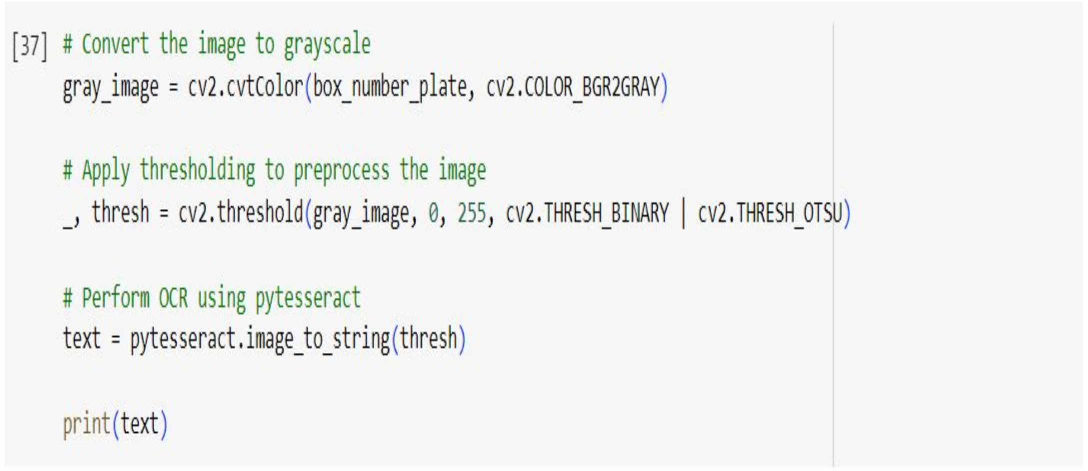
Results

- Filter the detected lines to remove similar lines and retain the unique ones.
- Calculate the bounding rectangle around the detected lines to localize the license plate region.



Results

- Implement OCR using Tesseract to extract text(license plate number) from the detected region.

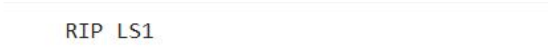


```
[37] # Convert the image to grayscale
     gray_image = cv2.cvtColor(box_number_plate, cv2.COLOR_BGR2GRAY)

     # Apply thresholding to preprocess the image
     _, thresh = cv2.threshold(gray_image, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

     # Perform OCR using pytesseract
     text = pytesseract.image_to_string(thresh)

     print(text)
```



```
RIP LS1
```

Summary

- Leveraged the Hough transform to extract lines from preprocessed images , subsequently filtering out redundant lines .
- Calculated precise bounding rectangles encompassing the detected lines to precisely localize license plate regions within images.
- Incorporated OCR using Tesseract to extract license plate numbers from identified images.

Future Steps

- Refine Thresholding: Experiment with different thresholding techniques like Otsu's method or adaptive thresholding to enhance image binarization for better edge detection.

- Train deep learning models on large scale datasets to leverage their capacity of feature learning and pattern recognition in diverse environmental conditions.