Khulna University of Engineering & Technology

CSE 4128
Image Processing and Computer Vision Laboratory

Available Parking Spaces Counter From Image

Submitted By:
Irhamul Islam
Roll Number: 1907093

Course Teachers:

Dr. SK. Md. Masudul Ahsan
Professor
Department of CSE,KUET

Dipannita Biswas
Lecturer
Department of CSE,KUET

Date of Submission : September 2,2024

Abstract

In this project,an image processing system is developed to accurately count available parking slots from an image. Using Python's OpenCV library, the system processes images through a series of steps: smoothing, binary image conversion, dilation, noise reduction, and adaptive thresholding. Smoothing is applied to reduce image noise and enhance the accuracy of subsequent processing steps. The binary image conversion and adaptive thresholding techniques are utilized to distinguish parking slots from the background effectively. Dilation helps to highlight the boundaries of the parking slots, making them more discernible. This systematic approach ensures a reliable count of available parking slots, providing a practical solution for parking management systems. The project's user interface is developed using Tkinter, facilitating ease of use and interaction for end-users.

Contents

List of Tables

List of Figures

1 Introduction

Efficient parking management is a critical aspect of urban infrastructure, directly impacting traffic flow and convenience for drivers. As urban populations grow and vehicle numbers increase, the demand for effective parking solutions becomes more pressing. Traditional methods of managing parking slots, such as manual counting or sensors, often prove to be inefficient, costly, or unreliable in large-scale implementations.

This project addresses the need for an automated and accurate method to count available parking slots from images using advanced image processing techniques. By leveraging OpenCV, a powerful open-source computer vision library, and developing a user-friendly interface with Tkinter, we have created a system capable of processing images and determining the number of available parking slots with high accuracy.

The core operations involved in this project include smoothing, binary image conversion, dilation, noise reduction, and adaptive thresholding. These techniques work together to enhance the clarity of the image and accurately identify parking slots. Smoothing helps in reducing noise, making the image cleaner for further processing. Binary image conversion and adaptive thresholding distinguish the parking slots from the background, while dilation emphasizes the edges, ensuring clear identification of each slot.

This system aims to provide a reliable and cost-effective solution for parking management, which can be easily integrated into existing infrastructure. The use of Tkinter for the user interface ensures that the system is accessible and easy to use for individuals with varying levels of technical expertise. This project not only demonstrates the practical application of image processing techniques but also contributes to the broader goal of smart city development by enhancing the efficiency of parking management systems.

This project has wide-ranging applications in various sectors, including urban parking management, where it automates the counting of available parking slots, thereby improving traffic flow and reducing congestion. In shopping malls, commercial complexes, airports, and train stations, it enhances parking efficiency by providing real-time data on available spaces, improving user convenience.

2 Methodology

2.1 Gaussian Blur

Gaussian Blur works by applying a Gaussian function to an image, effectively smoothing it and reducing noise and detail. This process involves convolving the image with a Gaussian kernel, where each pixel's value is replaced by the weighted average of its neighbors. The extent of the blur is controlled by the sigma value, with larger values resulting in more significant blurring. The kernel size, typically derived from the sigma value, determines the neighborhood size for the averaging process.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

2.2 Adaptive Threshold

Adaptive Thresholding converts a grayscale image to a binary image by setting pixel values to either maximum or minimum based on local neighborhood properties. Unlike global thresholding, which uses a single threshold value for the entire image, adaptive thresholding calculates thresholds for smaller regions, allowing it to handle varying lighting conditions. The block size determines the size of these regions, and a constant C is subtracted from the mean or weighted mean of the pixel values in each block to set the threshold. This method is particularly useful for images with uneven illumination.

$$B(x, y) = \begin{cases} 1 & \text{if } I(x, y) > T(x, y) \\ 0 & \text{if } I(x, y) \le T(x, y) \end{cases} \tag{2}$$

2.3 Median Filter

Median Filtering is a non-linear filtering technique used to remove noise from an image. Instead of averaging pixel values like Gaussian blur, median filtering replaces each pixel's value with the median value of its neighboring pixels within a defined kernel size. This approach is highly effective in preserving edges while removing salt-and-pepper noise. The kernel size, which is typically an odd number, defines the neighborhood used for calculating the median value.

2.4 Dilation

Dilation is a morphological operation that expands the white regions in a binary image. This technique uses a structuring element (kernel) to determine the shape and size of the neighborhood around each pixel. During dilation, each pixel in the input image is replaced by the maximum pixel value of the neighborhood defined by the kernel, effectively growing the white regions. The shape and size of the structuring element can vary, commonly being rectangular, elliptical, or cross-shaped. Dilation is often used to close small holes and connect disjointed white regions in binary images.

3 Implementation

In this section, we will outline the steps to implement Gaussian blur, adaptive thresholding, median filtering, and dilation using OpenCV. Each method will be applied to a sample image to demonstrate its effects and practical usage.

3.1 Parking Slots Select

To implement the selection of parking slots, we utilize a combination of user interaction and the pickle module for data persistence. Initially, the height and width of the parking slots are predefined according to the dimensions of the input image. The user manually selects the parking slots to be evaluated by clicking on the image. Each click records the coordinates of the top-left corner of a slot. Using these coordinates, the corresponding parking slots are cropped based on the defined height and width, ensuring precise extraction of each slot.
The coordinates of the cropped slots are stored in a list and subsequently saved to a pickle file for later use. This approach allows for efficient and accurate selection and storage of parking slot positions, facilitating further image processing tasks. By leveraging the pickle module, the coordinates are easily serialized and deserialized, ensuring that the slot positions can be reliably retrieved and used in subsequent steps of the project.

3.2 Smoothing

After the parking slots are selected and their positions saved, the next step involves further image processing. The selected image is read in grayscale mode to simplify subsequent operations and focus on intensity variations rather than color information. Grayscale conversion reduces computational complexity while retaining essential image features for analysis.

Following this, Gaussian blur is applied to the grayscale image. Gaussian blur smooths out noise and sharpens transitions between pixel intensities by convolving the image with a Gaussian kernel. This process effectively reduces high-frequency components, resulting in a blurred version of the image where subtle details and noise are suppressed. By applying Gaussian blur after selecting the parking slots, the image becomes more suitable for subsequent analysis steps such as feature extraction or pattern recognition, enhancing the overall quality of image data used in further processing tasks.

3.3 Adaptive Thresholding

After applying Gaussian blur to the grayscale image, the next step involves applying adaptive thresholding . Here's a breakdown of the process:

3.3.1 Function Input Parameters:

The function takes several parameters including the grayscal image, maximum pixel value for thresholding, adaptive method which can be 'mean' or gaussian, threshold type which can be 'binary' or 'binary inv', block size which defines the size of the local neighborhood, and constant (C) subtracted from the calculated threshold.

3.3.2 Image Preparation

The function first ensures the block size is odd, essential for correct neighborhood sampling around each pixel. It calculates the height and width of the input image.

3.3.3 Padding for Neighborhood Sampling

The image is padded symmetrically based on the block size to handle edge cases during neighborhood sampling. Padding ensures that all pixels have a defined local context for threshold calculation.

3.3.4 Threshold Calculation:

For each pixel in the original image, the function iterates through its neighborhood defined by the block size. Depending on the specified adaptive method ('mean' or 'gaussian'), it calculates a local threshold. If 'mean' is chosen, the threshold is computed as the mean of the neighborhood minus C. If 'gaussian' is chosen, a Gaussian kernel is applied to the neighborhood to weight pixel contributions before calculating the threshold.

3.3.5 Threshold Application:

Based on the threshold type, which determines the type of binary thresholding ('binary' or 'binary inv'), each pixel in the output image is set to either max value or 0. Pixels exceeding the calculated threshold retain max value, while others are set to 0 in 'binary' mode. In 'binary inv' mode, the logic is inverted: pixels above the threshold are set to 0, and others to max value.

3.3.6 Output

Finally, the function returns the output imgage, which represents the binary image resulting from adaptive thresholding. This image highlights regions of interest based on local intensity variations, effectively segmenting the image for further analysis or visualization.

3.4 Remove Noise

Following the application of adaptive thresholding, the next step is to remove noise from the binary image using a median filter.

3.4.1 Function Input Parameters:

The median filter function takes the binary image and the kernel size as inputs. The kernel size is typically an odd number, ensuring a valid median calculation.

3.4.2 Image Preparation:

The function determines the image dimensions and calculates the padding needed on each side of the image. This padding ensures that the edges of the image have a complete neighborhood for median calculation.

3.4.3 Padding the Image:

The image is padded using cv2.copyMakeBorder with reflection border mode. This padding is crucial for handling the edges of the image where the neighborhood size may otherwise be incomplete.

3.4.4 Median Filtering Process:

An empty output image is initialized with the same dimensions as the input image. The median index is calculated as half the total number of elements in the kernel, which helps in identifying the median value efficiently.

The function then iterates over each pixel in the image, excluding the padded borders. For each pixel, it collects the pixel values within the kernel's neighborhood, sorts them, and selects the median value to replace the current pixel value in the output image.

3.4.5 Output:

The function returns the output image, where each pixel value is replaced by the median of its neighborhood, effectively reducing salt-and-pepper noise while preserving edges.

3.5 Dilation

After applying the median filter to remove noise from the binary image, the next step is to perform morphological dilation.

3.5.1 Function Input Parameters:

The my dilate function takes the binary image , a structuring element kernel (kernel), and an optional parameter iterations that specifies how many times dilation should be applied.

3.5.2 Image and Kernel Preparation:

The function retrieves the dimensions of the input image and the kernel.
It calculates the padding needed around the image based on the kernel size to ensure complete coverage of the neighborhood during dilation.

3.5.3 Padding the Image:

he image is padded using cv2.copyMakeBorder with a constant border mode and a value of 0. This padding is crucial for handling edges and ensuring the dilation operation covers all image pixels.

3.5.4 Dilation Operation:

An empty array of the same size and type as the input image is initialized to store the dilated image.
The function iterates through each pixel in the image and applies the dilation operation. For each pixel, it extracts the neighborhood defined by the kernel size, applies the kernel (which in dilation is typically the maximum value), and updates the corresponding pixel in the dilated imgage.

3.5.5 Multiple Iterations (if specified):

If iterations is greater than 1, the dilation operation is applied iteratively. After each iteration, the dilated imgage becomes the input for the next iteration, ensuring a cumulative effect of dilation.

3.5.6 Output:

The function returns the iamgeg, which represents the result of applying morphological dilation to the input binary image. Dilation expands the white regions, closing gaps between objects and enhancing features of interest.

3.6 Checking Blank Spaces:

After applying morphological dilation to enhance the image, the next step involves checking for blank spaces within selected parking slots using the checkBlankSpace function. Here's an overview of how this function operates:

3.6.1 Function Input Parameter:

This function takes the input image as an argument. This image is typically the result of the dilation operation.

3.6.2 Initialization:

The function initializes a counter to keep track of the number of empty parking slots found.

3.6.3 Iterating through Selected Positions:

For each position (pos) in the posList (which contains the coordinates of selected parking slots):

- Extract the region of interest (ROI) from imgcheck using the coordinates (x, y) and the predefined height and width of each parking slot.
- Calculate the number of non-zero (non-white) pixels within the ROI using the non count zero function.
- If the count of non zero pixels (count) is below a certain threshold (900 in this case), it indicates that the parking slot is empty. In such cases, the slot is highlighted with a yellow rectangle (color = (255, 255, 0)) and thicker border (thickness = 5). The spaceCounter is incremented to keep track of empty slots.

- Otherwise, if the parking slot is occupied, it is highlighted with a red rectangle (color = (0, 0, 255)) and a thinner border (thickness = 2).

3.6.4 Drawing and Text Annotation:

- The function uses draw rectangle to outline each parking slot with the determined color and thickness based on occupancy status.
- It also annotates each parking slot with the count of non_zero pixels (count) using cvzone.putTextRect.
- Additionally, it displays the count of free spaces (spaceCounter) out of the total number of parking slots (len(posList)) at a fixed location on the image.

3.6.5 Final Output:

The function modifies imgage directly to annotate and highlight each parking slot based on its occupancy status and updates the space Counter dynamically as empty slots are identified.

Table 1: Parameter list of hand coded functions

Function	**Parameters**
my_count_nonzero	an image
draw_rectangle	image,point1,point2,color,thickness
get_gaussian_kernel	kernel size,sigma
gaussian_blur	image,kernel size,sigma
adaptive_threshold	image,maximumValue,adaptiveMethod,threshold type,block size,constant C
median_filter	image,kernel size
dilate	image,kernel,iterations

3.7 Psuedocode

3.7.1 Gaussian Blur

The following pseudocode illustrates the implementation of the Gaussian blur filter.

```
Function my_gaussian_blur(img, kernel_size, sigma):
    Get the dimensions of the image (image_h, image_w)
    Generate the Gaussian kernel using get_gaussian_kernel(kernel_size, sigma)

    Calculate the padding required for x and y directions:
        padding_x = (kernel_size - 1) // 2
        padding_y = (kernel_size - 1) // 2

    Pad the image with the calculated padding using cv2.copyMakeBorder
    Initialize the dimensions of the output image:
        output_image_h = image_h + kernel_size - 1
        output_image_w = image_w + kernel_size - 1

    Create an empty output image (gaussian_output) of size
    (output_image_h, output_image_w)

    For each pixel (x, y) in the padded image, excluding the padding:
        Initialize a temporary variable (temp) to 0
        For each element (i, j) in the kernel:
            Multiply the corresponding pixel in the padded image by the
            kernel value
            Add the result to temp
        Set the corresponding pixel in the output image (gaussian_output)
        to temp

    Crop the output image to the original image size:
        gaussian_output = gaussian_output[padding_y:padding_y + image_h,

        padding_x:padding_x + image_w]

    Normalize the output image to the range [0, 255] using cv2.normalize
    Convert the output image to unsigned 8-bit integer type

    Return the output image
```

3.7.2 Adaptive Threshold

The following pseudocode illustrates the implementation of Adaptive Threshold process.

```
Function my_adaptive_threshold(img, max_value, adaptive_method, threshold_type,
block_size, C):
    Ensure block_size is odd:
```

```
    If block_size is even:
        Raise ValueError "block_size must be an odd number."

Get the dimensions of the image (image_h, image_w)

Create an empty output image (output_img) of the same size as img,

initialized to zeros

Define the padding based on block size:
    padding = block_size // 2

Pad the image using cv2.copyMakeBorder with the calculated padding

For each pixel (y, x) in the original image:
    Get the neighborhood around the current pixel from the padded image

    Calculate the adaptive threshold for the current pixel:
        If adaptive_method is 'mean':
            threshold = Mean of the neighborhood - C
        Else if adaptive_method is 'gaussian':
            Create a Gaussian kernel with block_size
            threshold = Sum of the neighborhood weighted by the
            Gaussian kernel - C
        Else:
            Raise ValueError "adaptive_method must be either 'mean'
            or 'gaussian'."

    Apply the threshold to the current pixel:
        If threshold_type is 'binary':
            If the pixel value is greater than the threshold:
                Set the output pixel to max_value
            Else:
                Set the output pixel to 0
        Else if threshold_type is 'binary_inv':
            If the pixel value is greater than the threshold:
                Set the output pixel to 0
            Else:
                Set the output pixel to max_value
        Else:
            Raise ValueError "threshold_type must be either 'binary'
            or 'binary_inv'."

    Return the output image
```

3.7.3 Median Filter

The following pseudocode illustrates the implementation of the Median filter.

```
Function my_median_filter(img, kernel_size):
    Get the dimensions of the image (image_h, image_w)

    Calculate the padding required for x and y directions:
        padding_x = (kernel_size - 1) // 2
        padding_y = (kernel_size - 1) // 2

    Pad the image using cv2.copyMakeBorder with the calculated padding

    Create an empty output image (median_output) of the same size as
    img, initialized to zeros
    Calculate the median index:
        median_index = (kernel_size * kernel_size) // 2

    For each pixel (x, y) in the padded image, excluding the padding:
        Initialize an empty list (neighborhood)
        For each element (i, j) in the kernel:
            Append the corresponding pixel value from the padded image to
            the neighborhood list

        Sort the neighborhood list
        Set the corresponding pixel in the output image (median_output) to
        the median value (neighborhood[median_index])

    Return the output image
```

3.7.4 Dilation

The following pseudocode illustrates the implementation of the dilation process.

```
Function my_dilate(img, kernel, iterations=1):
    Get the dimensions of the image (image_h, image_w)
    Get the dimensions of the kernel (kernel_h, kernel_w)
```

```
Calculate the padding required for x and y directions:
    padding_x = (kernel_w - 1) // 2
    padding_y = (kernel_h - 1) // 2

Pad the image using cv2.copyMakeBorder with the calculated padding and
a constant value of 0

Create an empty output image (dilated_img) of the same size as
img, initialized to zeros

For each iteration from 0 to iterations - 1:
    For each pixel (x, y) in the padded image, excluding the padding:
        Extract the neighborhood around the current pixel from the
        padded image

        Apply the kernel (maximum value in the neighborhood):
            Set the corresponding pixel in the output image (dilated_img)

            to the maximum value in the neighborhood weighted by the kernel

    If more iterations are needed:
        Pad the dilated image using cv2.copyMakeBorder with the calculated

        padding and a constant value of 0

Return the output image
```

4 Result Analysis

In this section, we present the processed image along with the identified parking slots, highlighting empty and occupied spaces. This is achieved through an interactive user interface that visually displays the results and provides real-time feedback.

4.1 Home

The initial interface presents the homepage. The homepage serves as the starting point, guiding users through the process of selecting, processing, and analyzing parking slots.

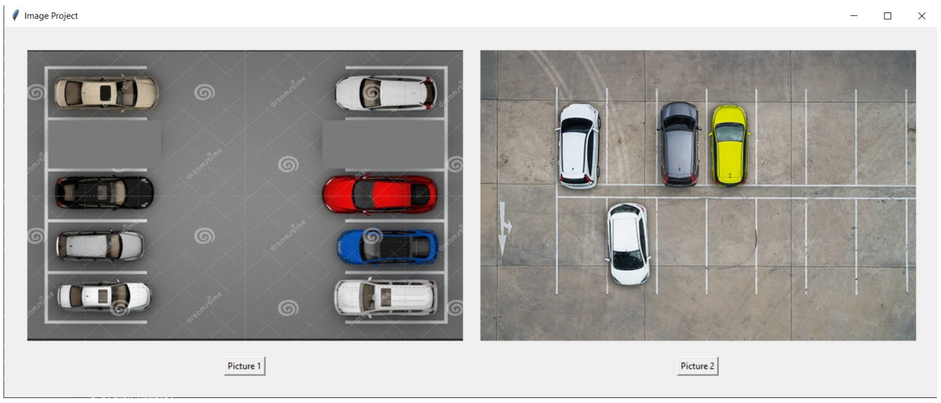

Figure 1: Home of parking slot counter.

4.2 Space Picker

In the Space Picker interface, users can manually select the parking slots to be evaluated. This interactive interface allows users to click on the image to mark the corners of each parking slot. Once the selection is complete, pressing the ’q’ key will exit the selection mode and save the coordinates of the selected slots. The user can then proceed to view the processed results

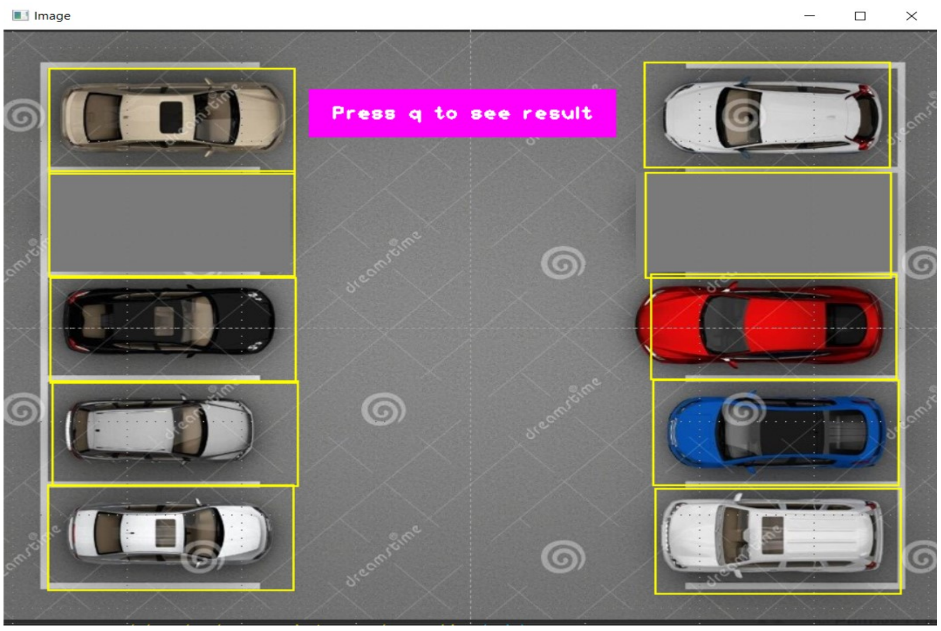

Figure 2: Picking Parking slots manually-Set 1.

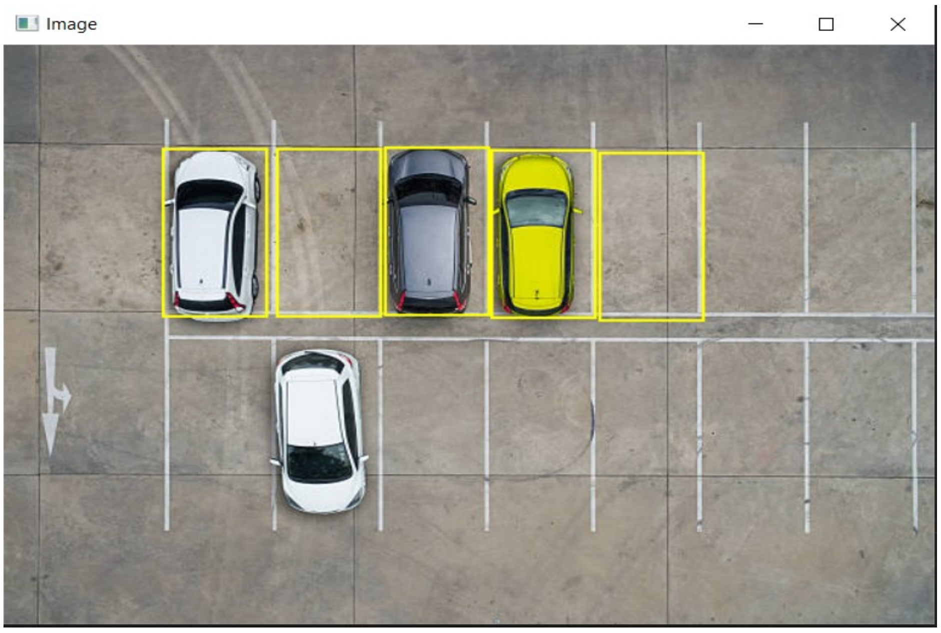

Figure 3: Picking Parking slots manually-Set 2.

4.3 Output of internal steps

The following section presents the intermediate outputs of the image processing pipeline:

4.3.1 Gaussian Blur

The grayscale image after applying Gaussian blur, which smooths the image and reduces noise.

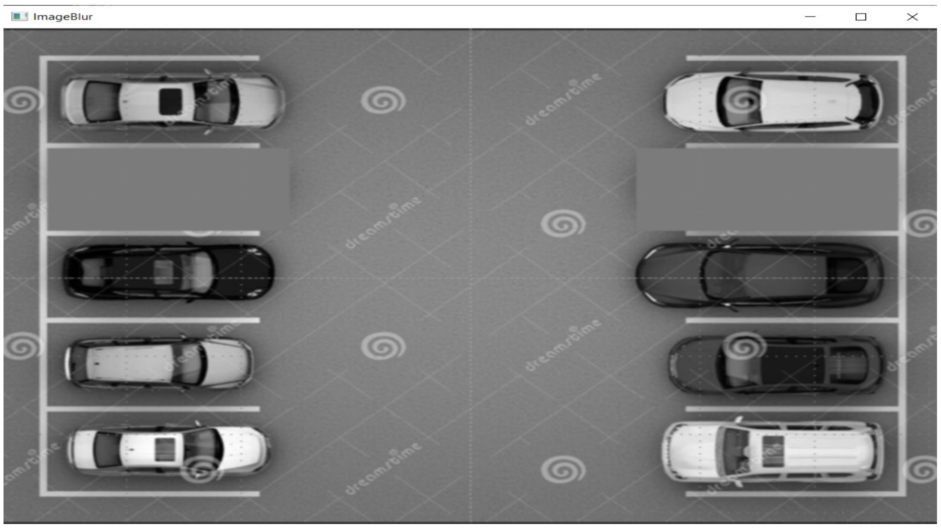

Figure 4: Gaussian blur output for set 1.

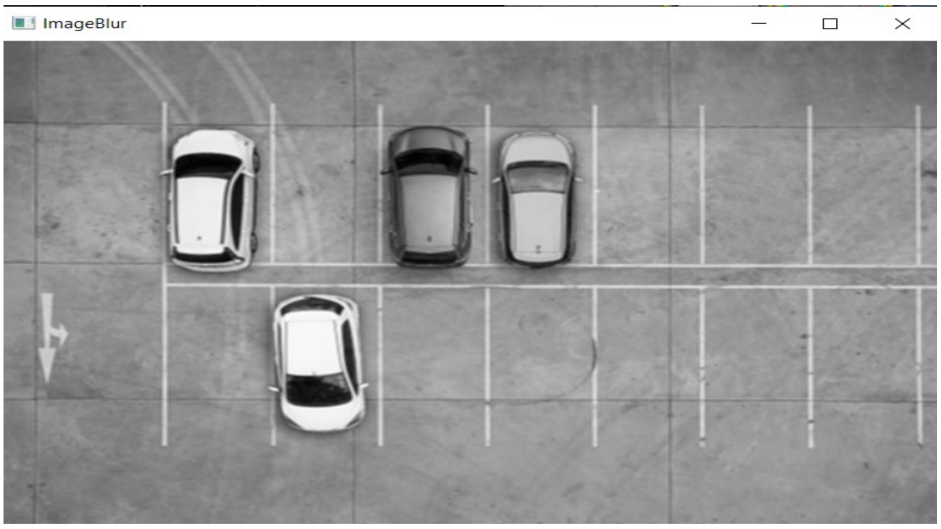

Figure 5: Gaussian blur output for set 2.

4.3.2 Adaptive Threshold

The binary image after applying adaptive thresholding, where each pixel is binarized based on local intensity values.

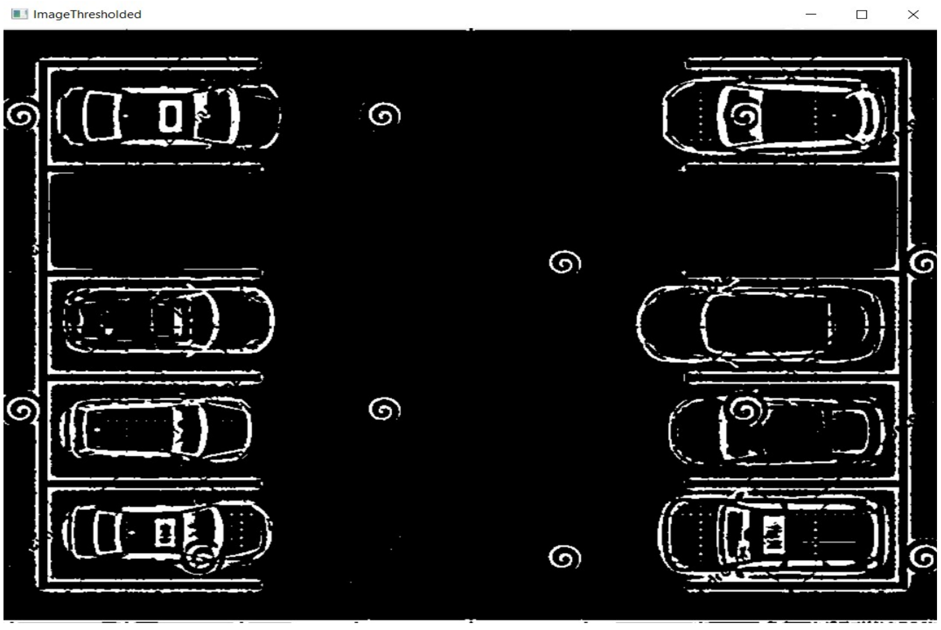

Figure 6: Threshold output for set 1.

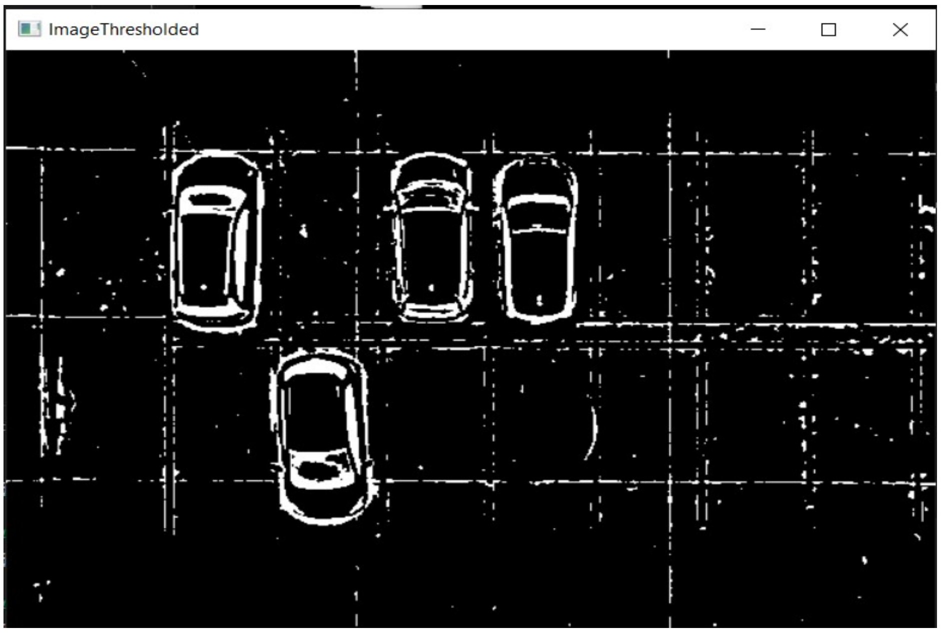

Figure 7: Threshold output for set 2.

4.3.3 Median Filter

The binary image after applying the median filter, which further reduces noise by replacing each pixel with the median value of its neighborhood.

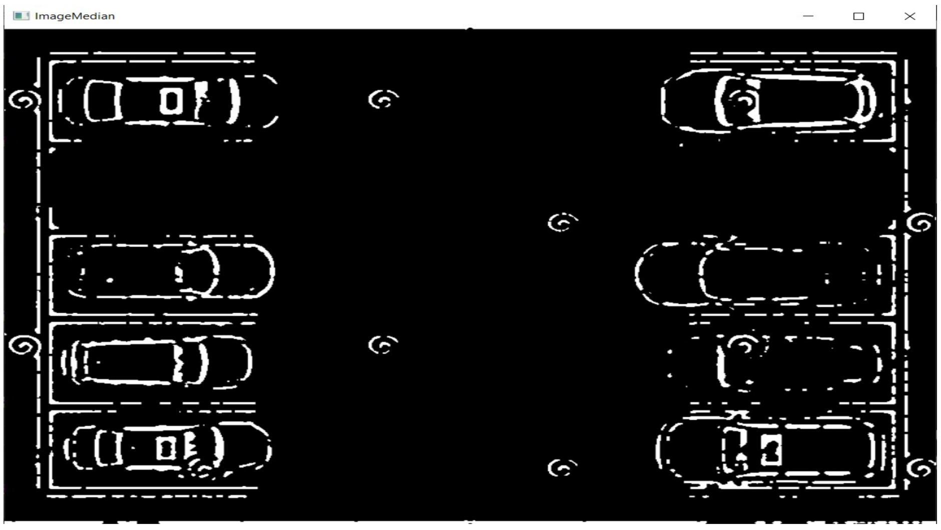

Figure 8: Noise reduction by median filter for set 1.

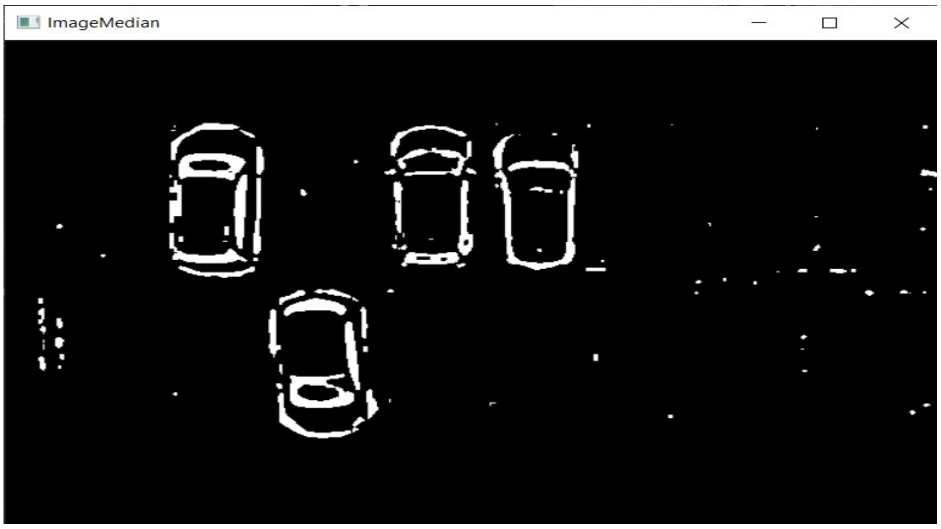

Figure 9: Noise reduction by median filter for set 2.

4.3.4 Dilation

The binary image after applying dilation, which expands the white regions and enhances the structure of the objects of interest.

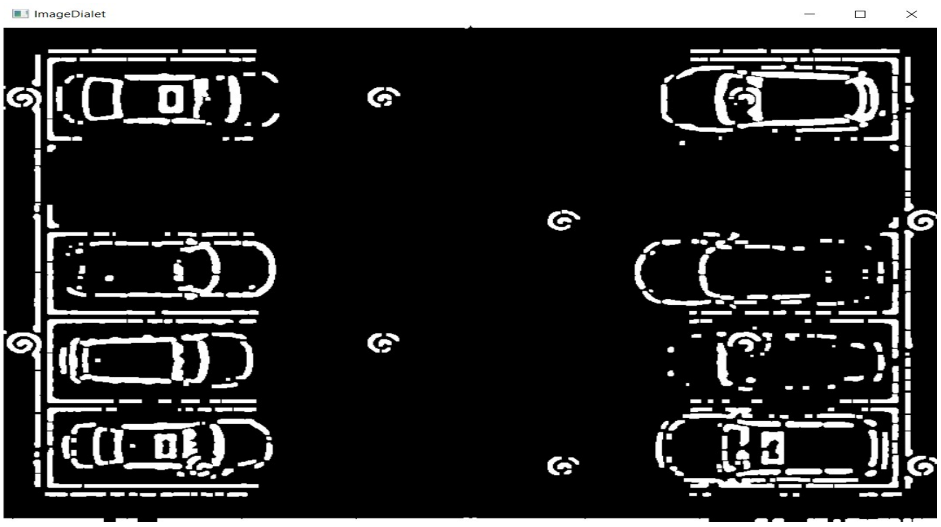

Figure 10: Dilation output for set 1.

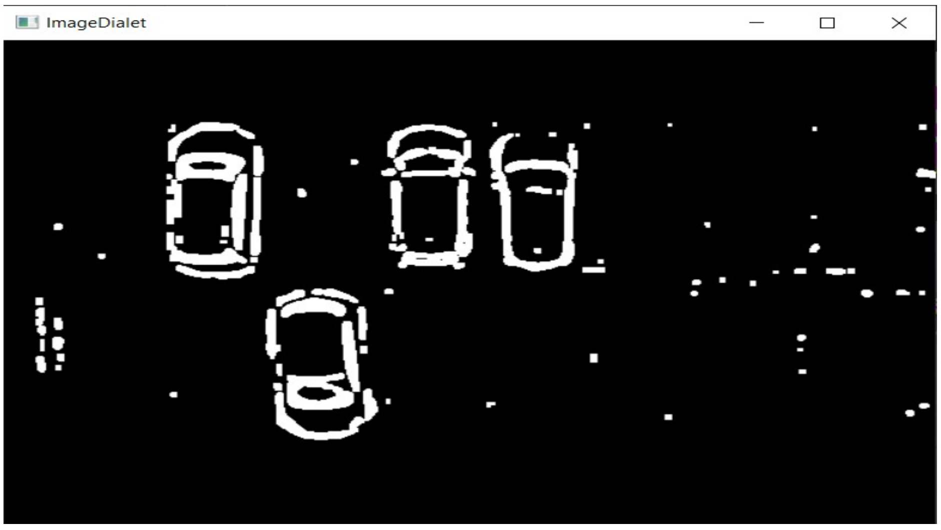

Figure 11: Dilation output for set 2.

4.4 Final Result

Here are the final processed images displaying the identified parking slots for both sets of images.Empty parking slots are highlighted with a yellow rectangle and labeled with the count of non-zero pixels.

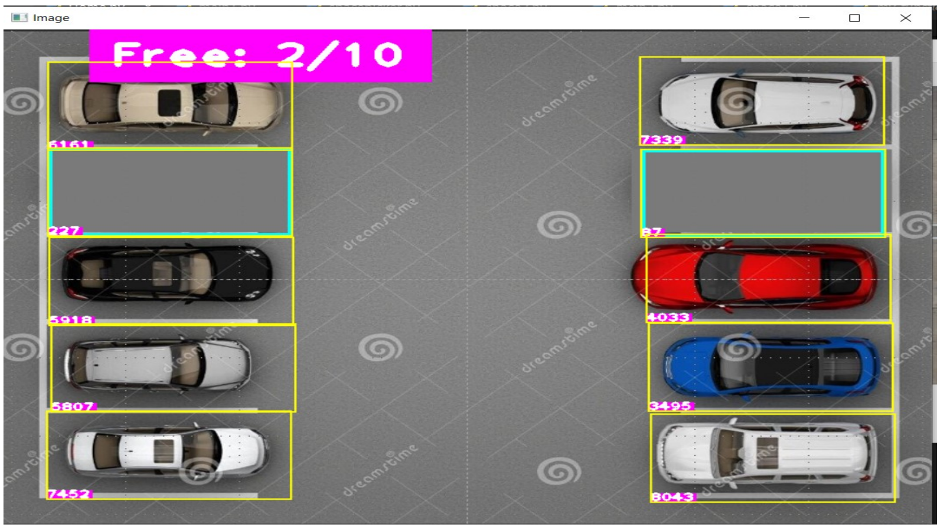

Figure 12: Final Outcome for set 1.

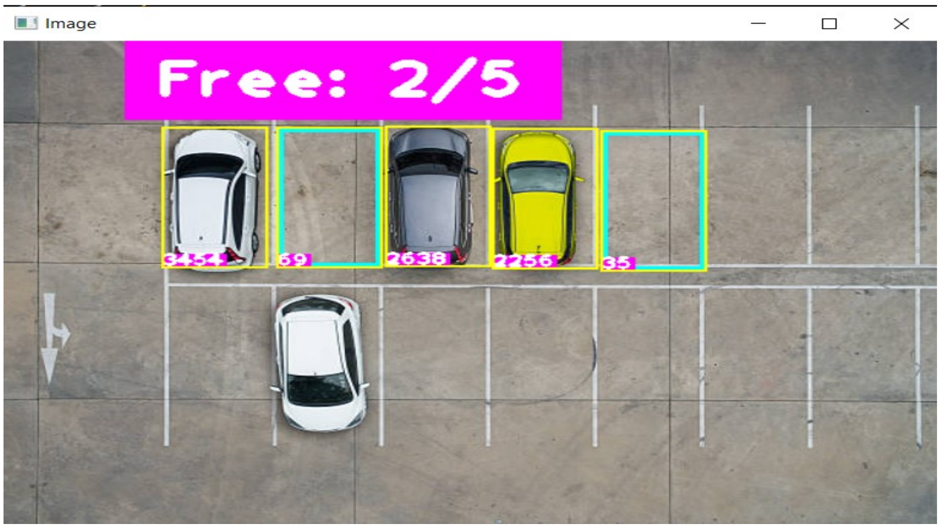

Figure 13: Final Outcome for set 2.

5 Future Improvements

Moving forward, several enhancements can be implemented to improve the parking slot detection system.
Implement algorithms for automated detection and selection of parking slots based on image processing techniques such as edge detection or clustering algorithms. This would eliminate the need for manual selection, making the system more efficient and scalable.Develop algorithms to dynamically adjust threshold values based on image characteristics or statistical analysis. Techniques like adaptive thresholding based on local image properties can be explored to automatically detect the presence of vehicles in parking slots.
Enhance the user interface (UI) to improve usability and provide intuitive controls for image selection, parameter adjustment, and result visualization. Implementing interactive features and real-time feedback can enhance user experience and facilitate easier interaction with the system.Modify the code architecture to handle multiple sets of images dynamically. Implement techniques such as batch processing or parameter tuning across different images to ensure robust performance and adaptability to varying scenarios.
Integrate machine learning models for more advanced tasks such as vehicle detection, parking slot occupancy prediction, or anomaly detection. Utilizing deep learning models can enhance accuracy and enable real-time decision-making capabilities in parking management systems.Gather user feedback and iterate on the system based on real-world usage scenarios. Continuous improvement based on user input and technological advancements will ensure that the parking slot detection system remains effective and relevant.

6 Conclusion

The parking slot detection project has demonstrated significant advancements in image processing and computer vision techniques for automated parking management systems. Through the implementation of Gaussian blur, adaptive thresholding, median filtering, and morphological operations like dilation, the system effectively identifies and analyzes parking slot occupancy.
While the current implementation requires manual selection of parking slots and manual threshold adjustment, future improvements can automate these processes using advanced algorithms and machine learning techniques. Enhancements in user interface design and system scalability are also crucial for broader application across diverse environments and larger datasets.
This project not only showcases the capabilities of image processing in real-

world applications but also highlights opportunities for further research and development in smart city technologies. By continuously refining and optimizing the system, we aim to contribute to more efficient and sustainable urban infrastructure management, ultimately enhancing the overall quality of urban living.