School of Computer Science and Information Technology

Department of Computer Science and Information Technology

Semester: V
Specialization: BCA - Internet of Things

COMPUTER VISION

(Development of a Parking Slots Identification and Tracking Model)

Date of Submission: 25-05-2025

Submitted by:
Name: Somnath Gorai
Reg No./USN No: 23BCAR0339
Signature

INDEX

Parking Slot Identification

&

Tracking Model

(Computer Vision)

1. ABSTRACT

In this project, we present a robust computer vision-based system for automatic parking slot detection and occupancy classification using classical image processing techniques. Designed to operate on both static images and real-time video input, the system addresses the core problem of efficient and scalable parking space monitoring, as part of the IHub Navavishkar challenge hosted by IIT Tirupati.

The primary goal was to detect individual parking slots and determine whether each slot is occupied or free, without relying on sensor data or heavy pre-trained models. To achieve this, two independent yet complementary approaches were implemented and evaluated.

(1) The first method utilizes pixel intensity-based detection by applying adaptive thresholding, Gaussian blurring, median filtering, and morphological operations to isolate potential vehicles. Parking slots are manually annotated and stored, and classification is performed by counting non-zero pixels within each region of interest.

(2) The second approach is based on Connected Component Analysis (CCA), where an empty reference image (created by manipulating the given input Image) is compared with the current input to extract foreground masks through absolute differencing. Connected regions in the cleaned binary mask are labelled as potential vehicles and are matched against slot boundaries to infer occupancy.

In both methods, the final output includes visual annotation of the parking image with color-coded bounding boxes and slot labels, along with a CSV summary file listing the total number of slots, the count of occupied and free spaces. Additionally, intermediate processing outputs are saved for transparency and validation. These solutions are designed to be lightweight, fast, interpretable, and deployment-ready; making them ideal for edge-based systems with limited computational resources.

While modern deep learning models like YOLO or SSD offer promising results, they require large labelled datasets, complex training pipelines, and may lack explainability. By contrast, the approaches proposed in this project offer an effective alternative that performs competitively without the overhead of data-driven learning.

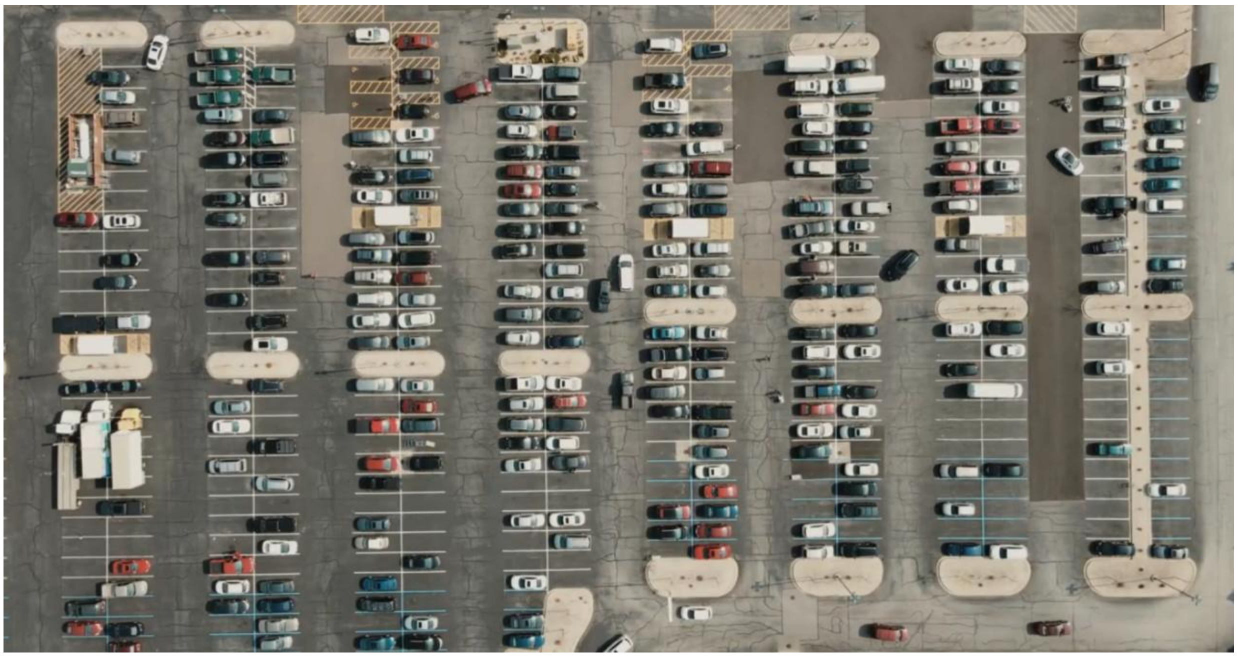
This report explores the methodology, implementation steps, visual results, mathematical foundation, and a critical comparison of both models, ultimately demonstrating how traditional techniques can still offer strong solutions in real-world parking analytics.

2. INTRODUCTION

2.1. Problem Context

Efficient parking management is a critical component of urban infrastructure, directly affecting traffic flow, fuel consumption, and user convenience. In conventional parking systems, monitoring and managing vehicle occupancy often requires physical sensors, manual inspections, or sophisticated AI models, all of which pose scalability and cost challenges.

The IHub Navavishkar challenge, organized by IIT Tirupati, presents a clearly defined problem: develop a vision-based solution capable of automatically identifying parking slots and classifying them as either occupied or available using a single image to implement the solution. The system should be lightweight, interpretable, and practical to deploy in real-world conditions without relying on proprietary models or extensive training data.



2.2. Real-World Importance

In real-world settings, many parking lots already have CCTV cameras installed for surveillance. Leveraging these existing video feeds for intelligent parking analytics provides a cost-effective alternative to sensor-based solutions. By automating slot detection and classification using image processing, such systems can help:

- Reduce traffic congestion by guiding drivers to available slots
- Minimize time spent searching for parking
- Provide live parking data to city planning and facility managers
- Enable integrations with broader smart-city ecosystems

2.3. Project Objectives

This project aims to develop a computer vision-based system that can accurately detect parking slots and determine their occupancy status using only visual input. The solution must:

- Accept a **static image** of a parking lot as input (with possible extensions to live video)
- Identify and localize **individual parking slots**
- Classify each slot as **occupied or free**
- Produce **annotated visual output** and **CSV reports** with summarized status
- Be **lightweight and interpretable**, without dependence on deep learning models or large labelled datasets

2.4. Proposed Solutions

To achieve these goals, the project explores two classical image processing-based approaches:

1. **Pixel Intensity Thresholding** – using grayscale conversion, adaptive thresholding, and region-based analysis
2. **Connected Component Analysis (CCA)** – leveraging image differencing between an empty reference frame and the current input to detect foreground blobs representing vehicles

This report outlines the design, implementation, and evaluation of both methods and provides a comparative assessment with modern deep learning-based alternatives. The solution adheres to the challenge constraints while aiming for real-world practicality and robustness.

3. RELATED WORKS

Automated parking space detection has been a subject of research and practical implementation for several years. Existing solutions generally fall into three main categories: **sensor-based methods**, **deep learning-based vision systems**, and **classical image processing approaches**.

3.1. Existing Works

3.1.1. Sensor-Based Systems

Traditional smart parking solutions often rely on embedded hardware such as infrared sensors, ultrasonic sensors, or RFID tags to detect vehicle presence. These systems can provide reliable, real-time information but are expensive to install and maintain, especially in large-scale or outdoor environments. Moreover, they are difficult to retrofit in existing infrastructures without extensive modifications.

3.1.2. Deep Learning Approaches

With the advancement of computer vision, deep learning models such as **YOLO (You Only Look Once)**, **SSD (Single Shot Detector)**, and **Mask R-CNN** have been applied to parking detection tasks. These models can identify vehicles in images with high accuracy and work effectively on video feeds. However, their use typically requires:

- Large annotated datasets for training
- High computational resources (especially on edge devices)
- Longer development and deployment cycles
- Lower explainability due to their black-box nature

While these models perform well in dynamic environments, they may be unsuitable in contexts where transparency, speed, or hardware simplicity is prioritized.

3.2. Positioning of This Project

In this project, we deliberately chose two classical image processing approaches—**pixel intensity thresholding** and **Connected Component Analysis (CCA)**—over deep learning alternatives. This choice aligns with the challenge constraints and addresses the need for a **lightweight, interpretable, and data-independent solution**. These techniques leverage core principles of visual pattern recognition and demonstrate that effective parking slot detection can be achieved without complex models or training pipelines.

4. PROPOSED METHEDOLOGY

4.1. Method 1: Pixel Intensity Thresholding

This approach uses classical image processing techniques to determine the occupancy status of parking slots based on pixel intensity variations within each slot region. It does not require a reference image or any trained model, making it lightweight and easy to deploy in static-camera environments.

The method works by preprocessing the input image to emphasize the visual differences between vehicles and background (typically asphalt or concrete). Each predefined parking slot is then cropped and analysed individually. The number of white (non-zero) pixels within each slot is computed, and a threshold is applied to decide whether a vehicle is present.

4.1.1. Step-by-Step Methodology:

4.1.1.1. Grayscale Conversion

The input image is first converted to grayscale to simplify processing and focus on intensity-based features (mainly white and black, i.e., 0s and 1s respectively).

imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

4.1.1.2. Gaussian Blurring

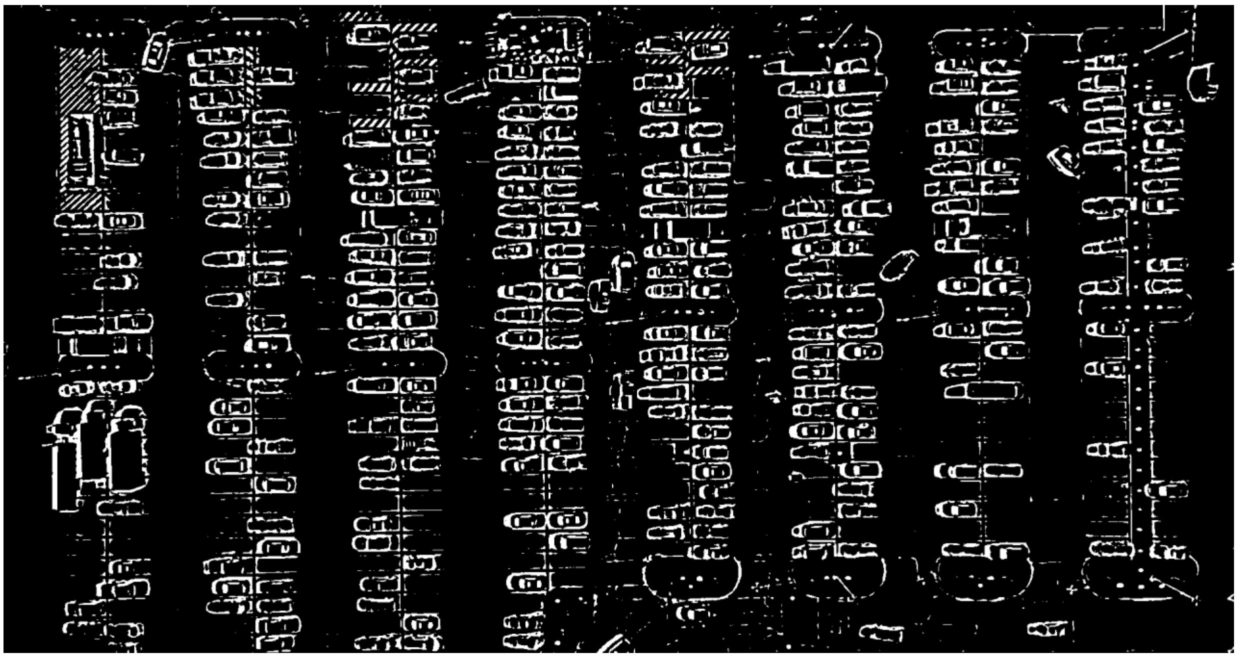
Gaussian Blur is applied to reduce noise and smooth out minor pixel-level inconsistencies which could affect thresholding.

imgBlur = cv2.GaussianBlur(imgGray, (3, 3), 1)

4.1.1.3. Adaptive Thresholding

Adaptive thresholding is used to convert the blurred image into a binary image where potential vehicles appear as white blobs. This method adjusts the threshold dynamically for different lighting regions.

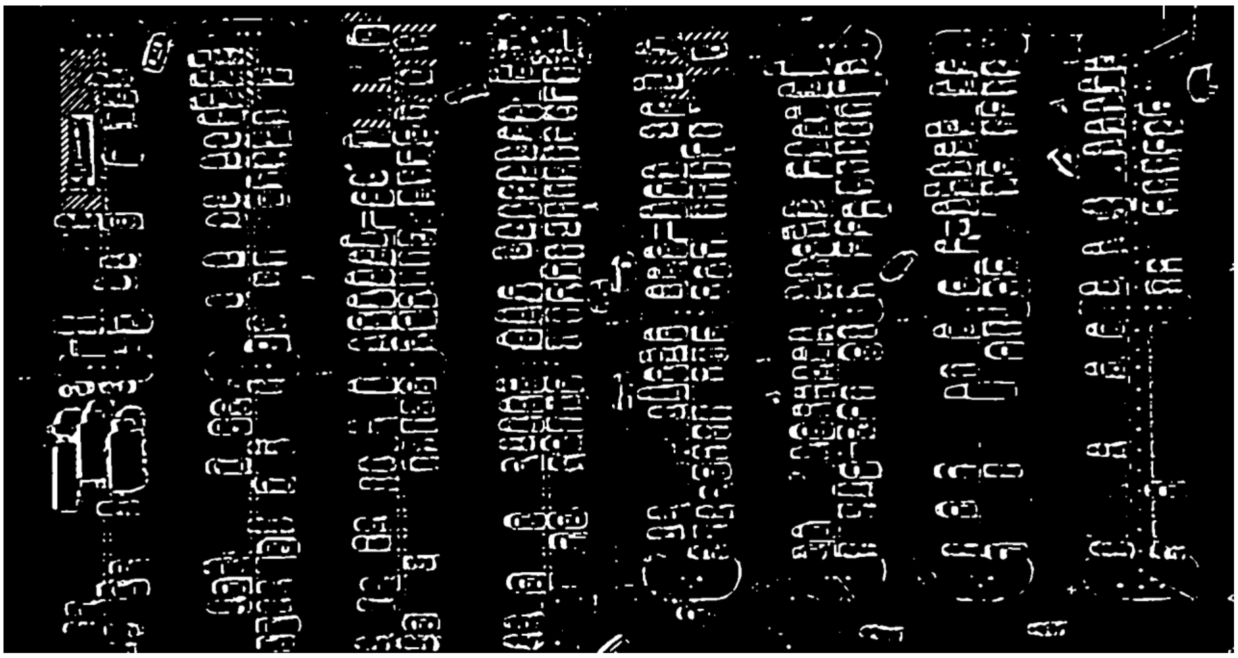
imgThres = cv2.adaptiveThreshold(imgBlur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, val1, val2)



4.1.1.4. Median Filtering

To remove salt-and-pepper noise from the thresholded image, a median filter is applied.

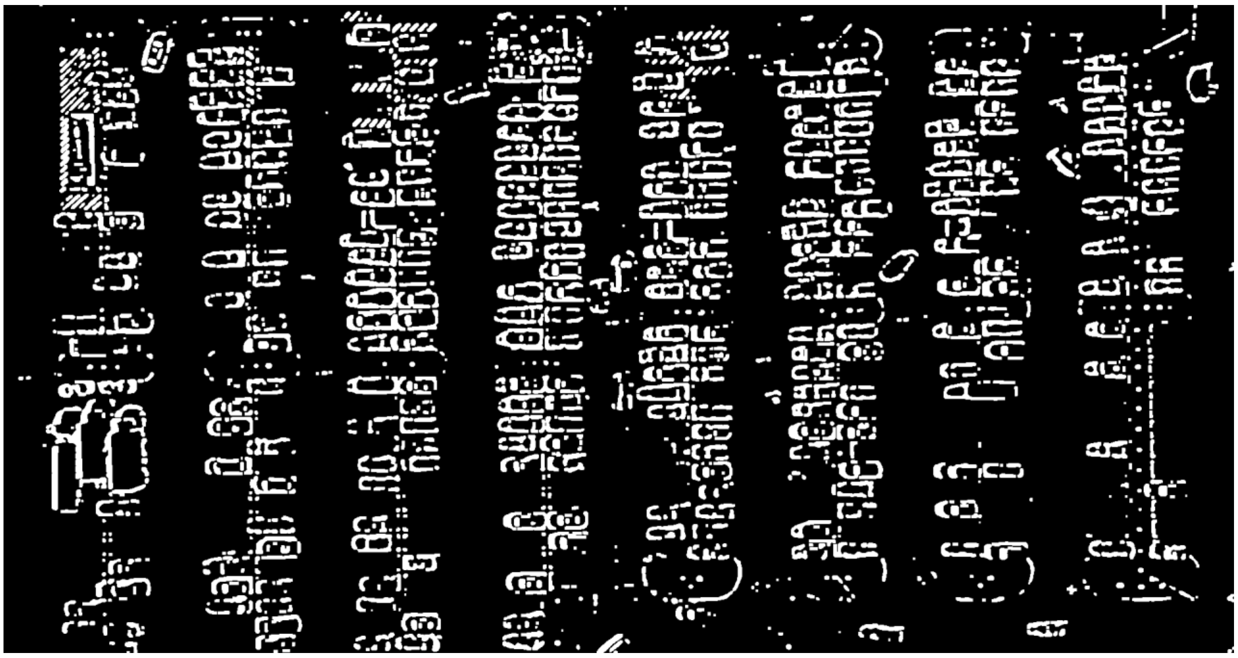
imgThres = cv2.medianBlur(imgThres, val3)



4.1.1.5. Morphological Dilation

Dilation is performed to strengthen object regions and fill small gaps within the vehicle shapes. This ensures more reliable pixel counting.

kernel = np.ones((3, 3), np.uint8)
imgThres = cv2.dilate(imgThres, kernel, iterations=1)

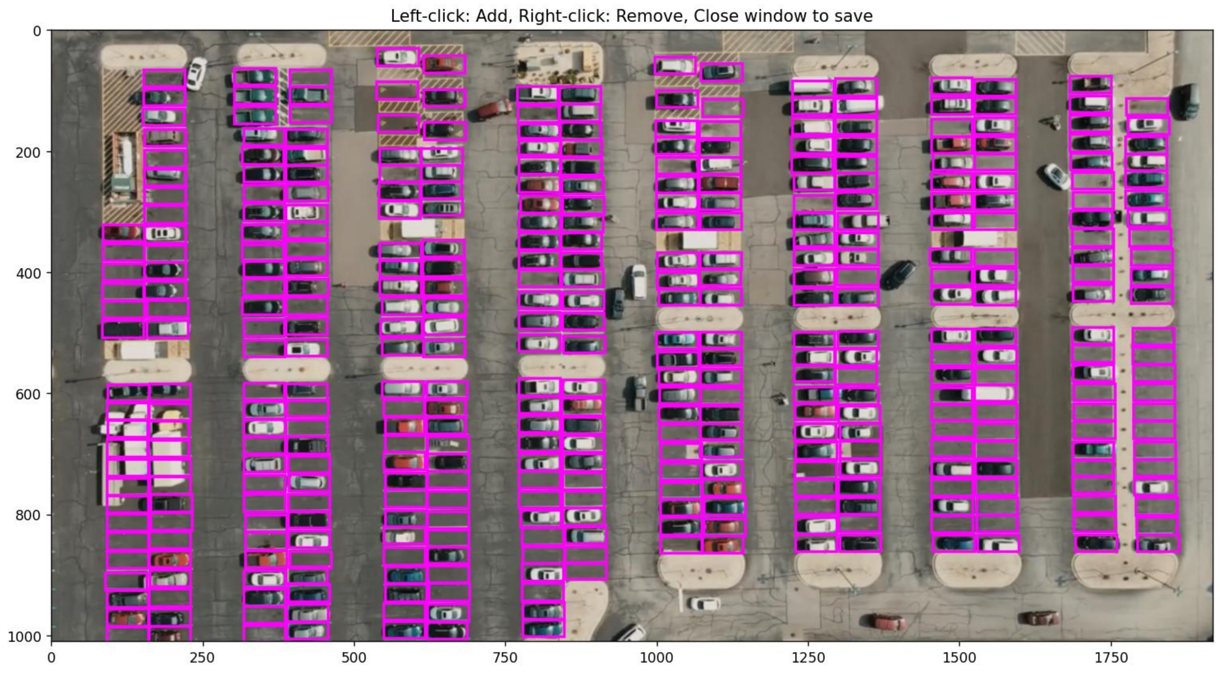


4.1.1.6. Slot-wise Occupancy Classification (with Manual Labeling Tool)

Before classifying the slots, we first need to define where each parking slot is located in the input image. To achieve this, a simple interactive labeling tool was created using **Matplotlib** and **OpenCV**. This tool allows the user to manually annotate the top-left corner of each rectangular parking slot by clicking directly on the image.

- **Left-click** adds a new slot.
- **Right-click** removes an existing slot if clicked within its bounds.
- Slots are drawn as magenta rectangles of fixed width and height.
- Once labeling is complete, the slot coordinates are stored in a file named parking_slots using Python's pickle module for later use.

```
with open('parking_slots', 'wb') as f:
    pickle.dump(posList, f)
```

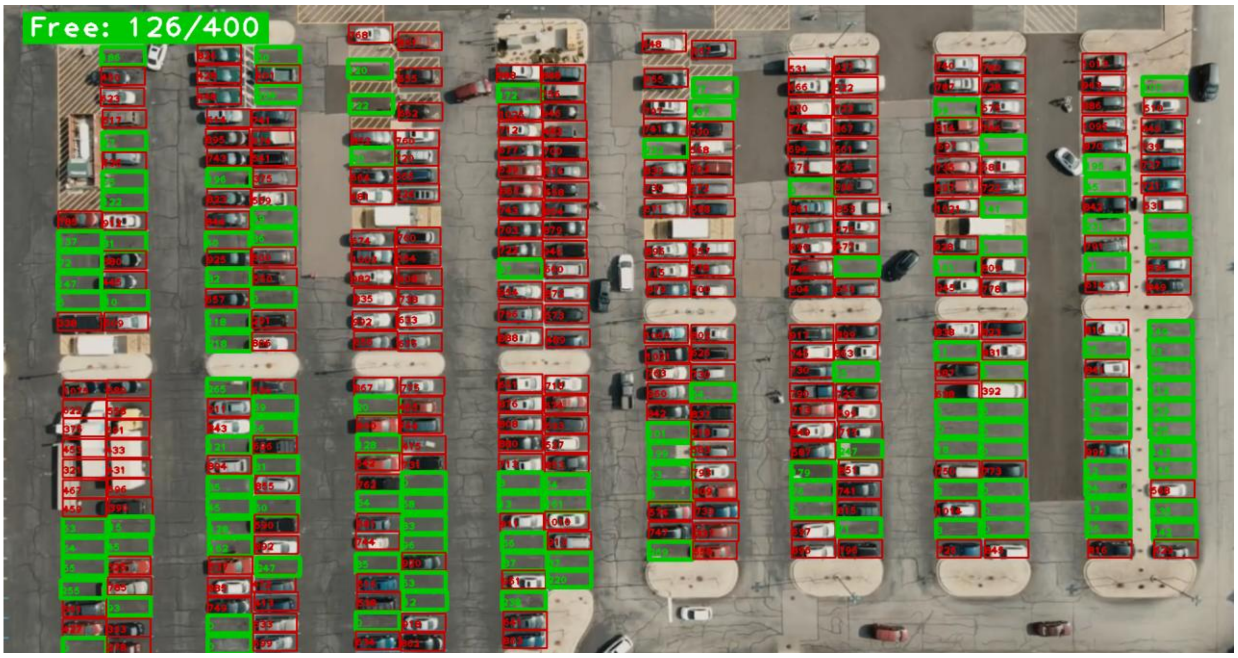


Once the slots are defined, each region is extracted from the thresholded binary image. The number of white (non-zero) pixels in each slot is computed using OpenCV's countNonZero() function. This value indicates the presence of a vehicle based on how much white (foreground) area is detected.

The logic used to classify a slot as **occupied** or **free** is based on a manually tuned threshold (threshold_count). If the pixel count exceeds this threshold, it is assumed that a vehicle is present in the slot.

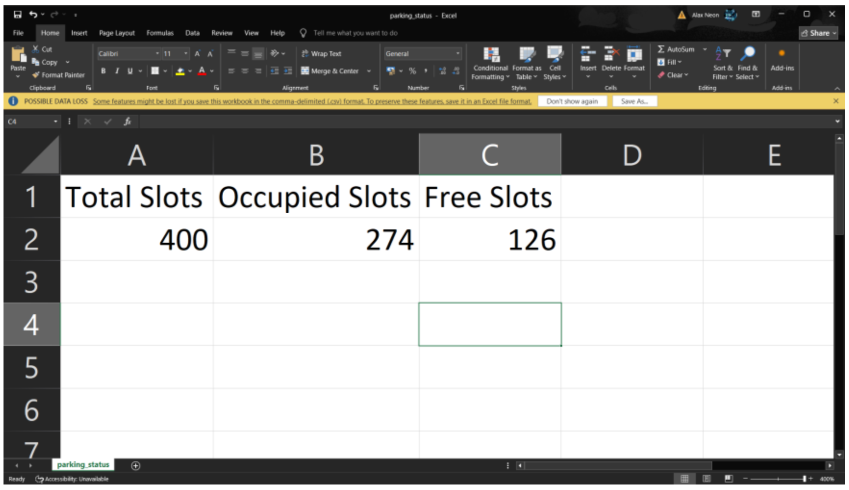
```
imgCrop = imgThres[y:y + h, x:x + w]
count = cv2.countNonZero(imgCrop)

if count < threshold_count:
    color = (0, 200, 0)  # Green = Free
    spaces += 1
else:
    color = (0, 0, 200)  # Red = Occupied
```



4.1.2. Output

In addition to the image output, the total number of slots, the count of occupied and free slots are saved in a CSV file.



POSSIBLE DATA LOSS Some features might be lost if you save this workbook in the comma-delimited (.csv) format. To preserve these features, save it in an Excel file format. Don't show again Save As...

	A	B	C	D	E
1	Total Slots	Occupied Slots	Free Slots		
2	400	274	126		
3					
4					
5					
6					
7					

4.1.3. Advantages

- **Lightweight**: Can run on basic hardware with minimal computational load.
- **No dataset required**: Works on any input of the same parking plot with one time labelling the parking slots, without needing training.
- **Fast**: Suitable for near real-time use cases.

4.1.4. Limitations

- **Lighting Sensitivity**: Shadows and highlights may influence the thresholding.
- **Fixed Slot Geometry**: Slots must be manually annotated.
- **Camera Angle Constraints**: Works best with fixed-angle, top-down or wide-angle views.
- **Hard Thresholding**: Misclassification may occur if vehicle edges are incomplete or if the threshold is improperly set.

While the pixel intensity thresholding method is effective in controlled environments, it has several inherent limitations. It is sensitive to shadows, lighting variations, and image noise, all of which can affect threshold accuracy and lead to misclassification. Moreover, this approach requires careful tuning of parameters and assumes relatively consistent scene conditions. To overcome these challenges and improve robustness, I explored an alternative technique — **Connected Component Analysis (CCA)** — which compares the current image with a clean background reference to detect foreground objects more reliably.

4.2. Method 2: Connected Component Analysis (CCA)

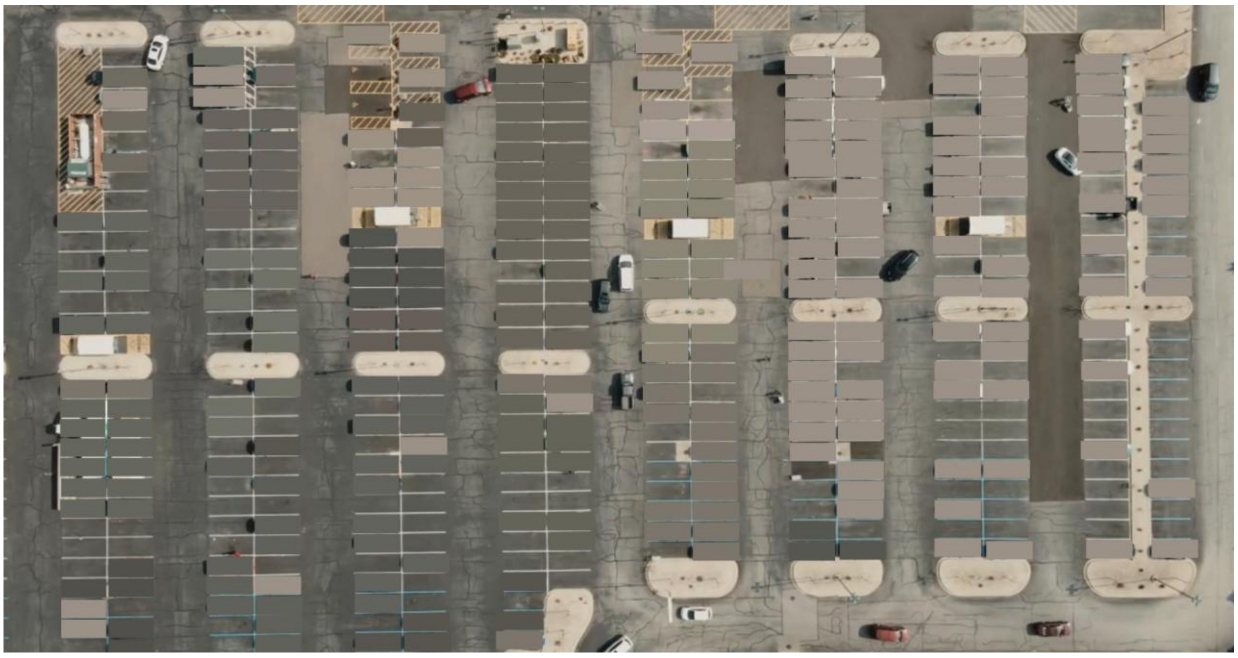
Connected Component Analysis (CCA) is a blob detection method used to identify connected regions (components) in binary images. In this project, it is used to detect foreground objects (vehicles) by comparing a current image of the parking lot with a manually generated empty reference image. This approach is more robust to lighting variation and vehicle shape diversity compared to fixed pixel intensity thresholding.

4.2.1. Step-by-Step Methodology:

4.2.1.1. Reference Image Setup

Unlike the pixel intensity method, the Connected Component Analysis (CCA) approach requires a clean reference image of the parking lot without any vehicles. However, for the competition, only a single image of the occupied parking lot was provided. To work around this, the reference image (parking_lot_empty.png) was manually created by **digitally masking out** the parked vehicles using an image editing tool (Canva). Patches of background-like color and texture were drawn over each car to simulate an empty lot.

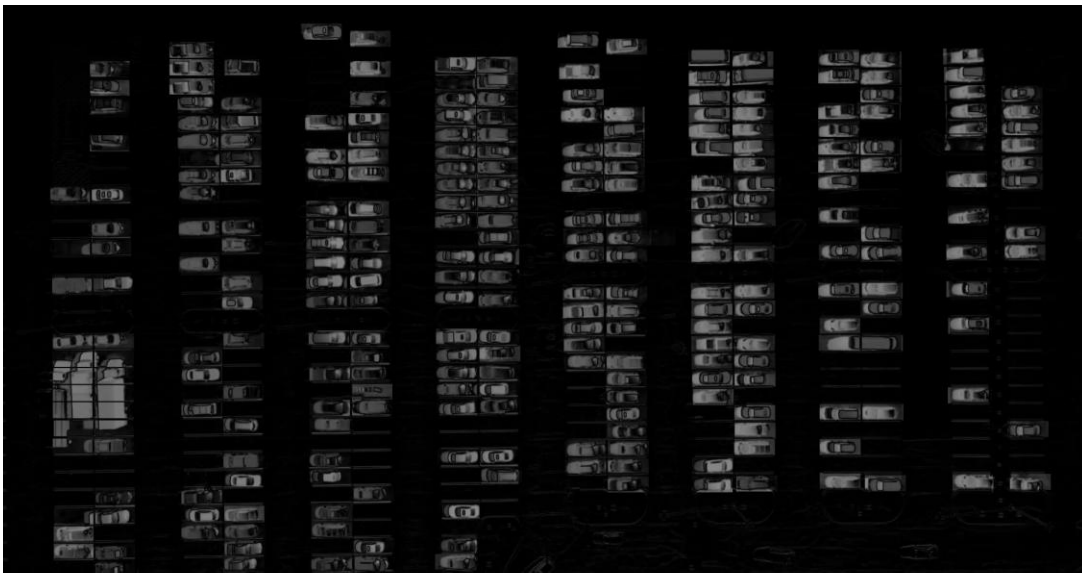
While this method was sufficient for prototyping, it introduces approximation errors. As a best practice, obtaining a **naturally captured image** of the empty parking plot would ensure more accurate foreground extraction and improve system robustness.



4.2.1.2. Image Preprocessing and Difference Extraction

Both the empty and current images are converted to grayscale. Their absolute difference highlights regions where vehicles are present (i.e., the foreground).

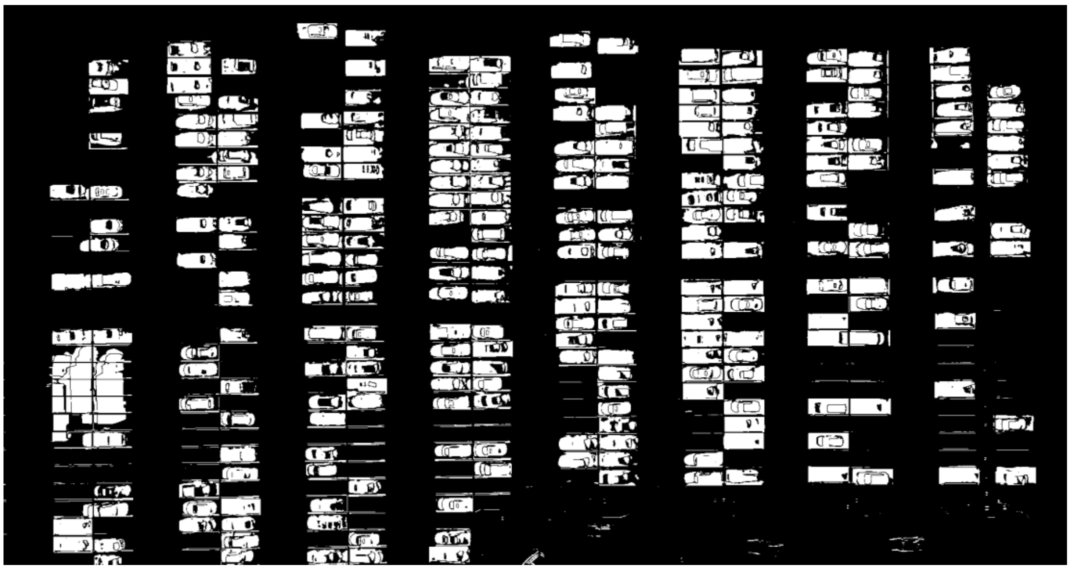
```
diff_img = cv2.absdiff(gray_empty, gray_current)
_, fg_mask = cv2.threshold(diff_img, THRESHOLD_BINARY_DIFF, 255, cv2.THRESH_BINARY)
```



4.2.1.3. Binary Thresholding

Once the absolute difference between the current and empty grayscale images is obtained, a **binary threshold** is applied to convert the difference image into a binary mask. This highlights foreground changes (likely vehicles) as white (255) and the background as black (0). The threshold value (THRESHOLD_BINARY_DIFF) determines the minimum intensity change required to consider a pixel as foreground.

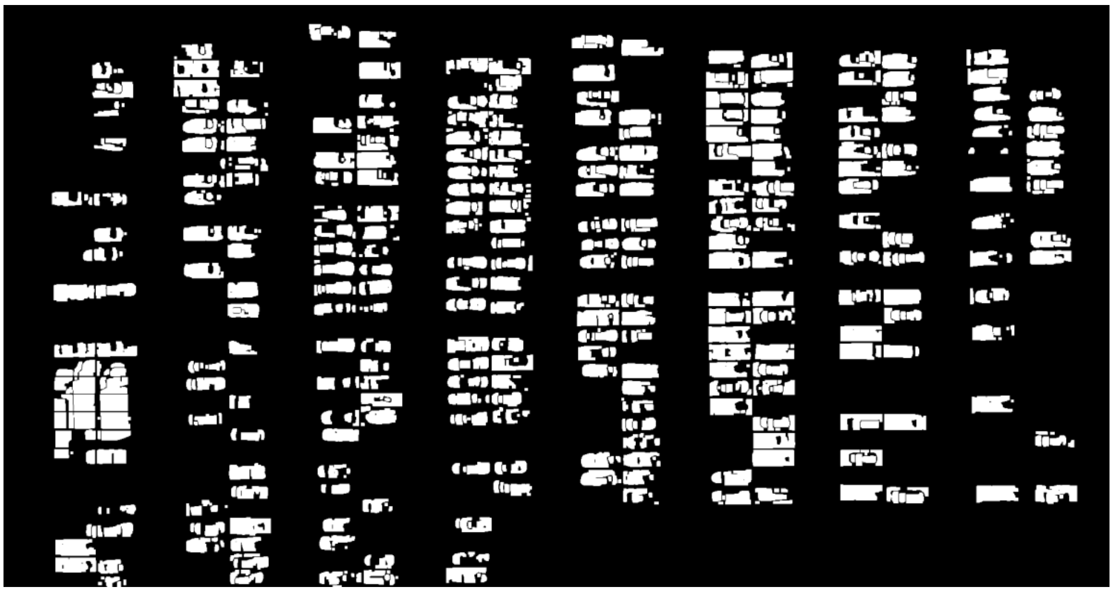
```
_, fg_mask = cv2.threshold(diff_img, THRESHOLD_BINARY_DIFF, 255, cv2.THRESH_BINARY)
```



4.2.1.4. Morphological Cleaning

Noise such as shadows or small objects is removed using morphological operations (e.g., morphological opening). This improves the clarity of the vehicle blobs.

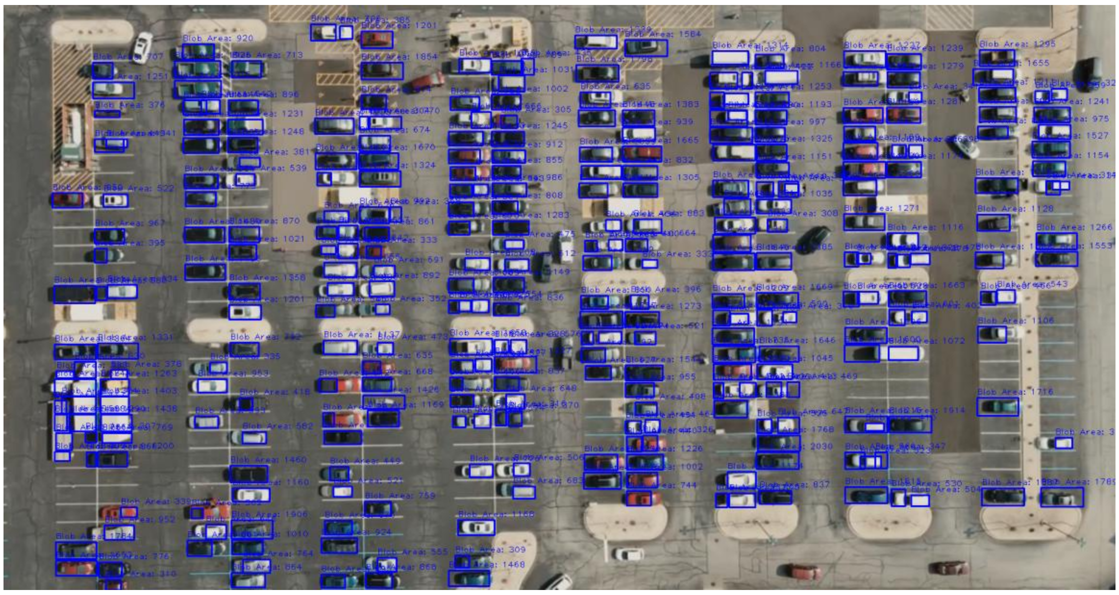
```
kernel = np.ones((3, 3), np.uint8)
fg_mask = cv2.morphologyEx(fg_mask, cv2.MORPH_OPEN, kernel, iterations=2)
```



4.2.1.5. Connected Component Analysis (Blob Detection)

The cleaned binary mask is passed to cv2.connectedComponentsWithStats() to label connected regions and extract their bounding boxes and areas. And gradually labels all the differences in the vehicles spotted on the parking plot.

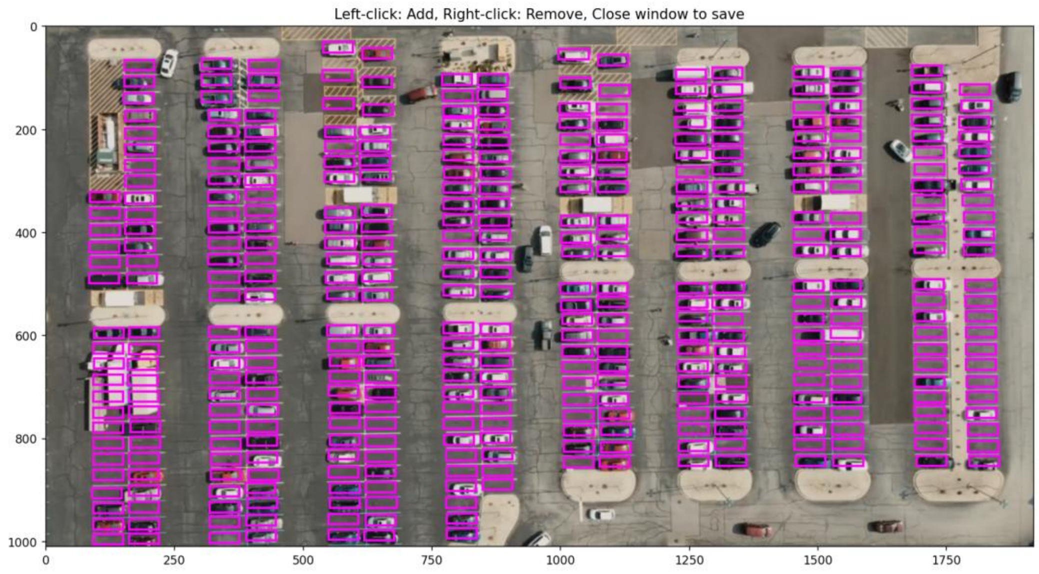
```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(fg_mask)
```



4.2.1.6. Slot Labelling (Slot Picker)

The **same slot picker tool** used in the pixel intensity method was reused for CCA, but with a minor change:

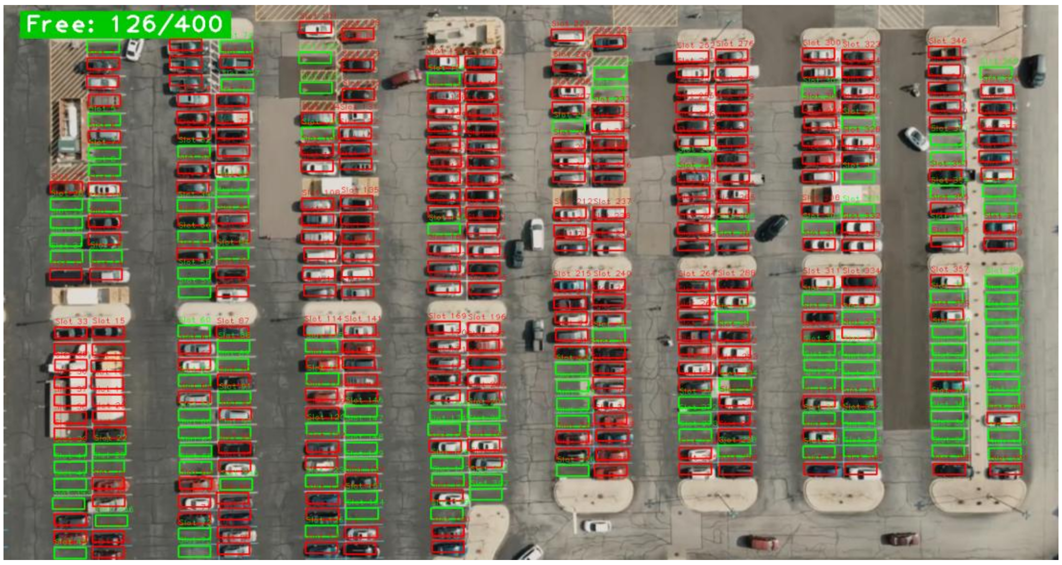
- In the previous method, the tool saved only the **top-left corner** of each rectangle.
- For CCA, the version used saved **slot ID and full dimensions** (i.e., x, y, width, height) into a CSV file (slots.csv), making it easier to match with detected blob bounding boxes.



4.2.1.7. Overlap Calculation and Occupancy Classification

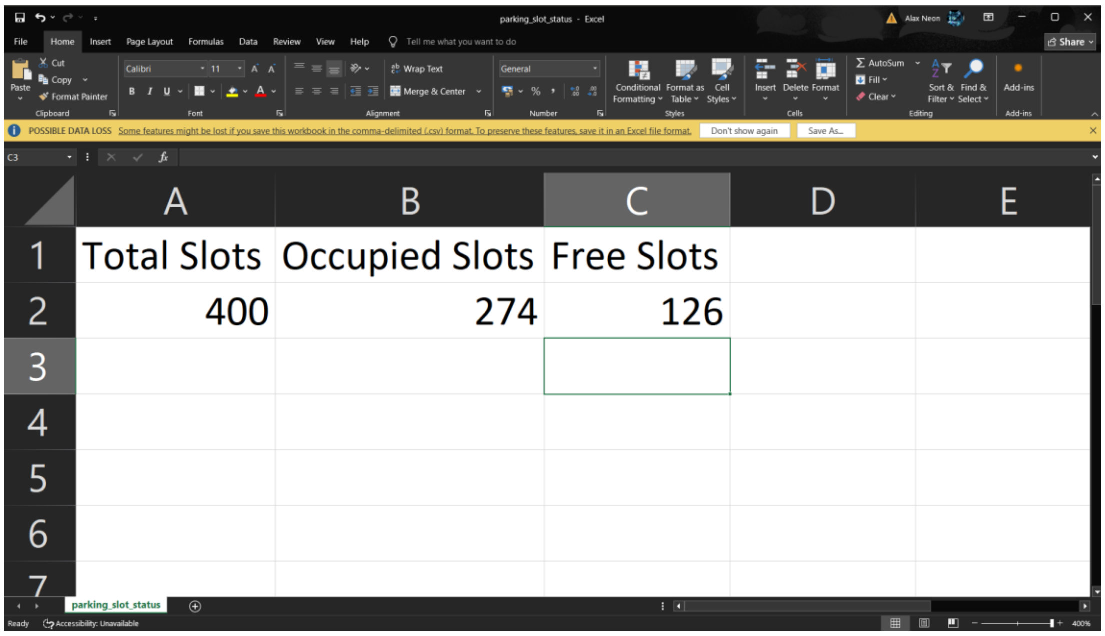
Each detected blob is checked for **intersection (overlap)** with parking slots using rectangle bounding boxes. A slot is marked as **occupied** if any significant overlap is found between it and a vehicle blob.

x_overlap = max(0, min(slot_box[2], vehicle_box[2]) - max(slot_box[0], vehicle_box[0]))
y_overlap = max(0, min(slot_box[3], vehicle_box[3]) - max(slot_box[1], vehicle_box[1]))
*overlap_area = x_overlap * y_overlap*



4.2.2. Output

In addition to the image output, the total number of slots, the count of occupied and free slots are saved in a CSV file.



Total Slots	Occupied Slots	Free Slots
400	274	126

4.2.3. Advantages

- **Robust to Lighting**: Doesn’t rely on absolute pixel values.
- **No Need for Parameter Tuning**: Eliminates threshold fine-tuning.
- **Handles Diverse Vehicle Sizes**: Automatically detects irregular blobs.

4.2.4. Limitations

- **Requires a Clean Reference Image**: Not always easy to obtain.
- **Less Suitable for Dynamic Lighting Scenes**: If shadows or lighting changes drastically between the empty and current image, false positives can occur.
- **Assumes Static Camera**: The reference and current image must be perfectly aligned.

5. MODEL ACCURACY

Evaluating the performance of a parking detection system, even when not based on machine learning, is crucial to understanding its real-world reliability. In this project, the occupancy classification is performed through deterministic rule-based logic using two methods: Pixel Intensity Thresholding and Connected Component Analysis (CCA). Both techniques output a binary classification for each parking slot: either **Occupied** or **Free**.

Although metrics like **accuracy**, **precision**, **recall**, and **F1-score** are traditionally used for machine learning models, they are equally applicable here because the system produces classification outputs that can be compared with a manually annotated ground truth.

5.1. Why Accuracy Evaluation Still Applies

This system does not use any trained model or learning-based approach. Instead, it applies handcrafted logic based on pixel-level changes and blob detection. However, the system still performs classification — deciding whether each parking slot is occupied or not based on specific thresholds or geometric overlaps. Therefore, it is possible to evaluate the **effectiveness of the logic itself** by comparing its output against human-labelled ground truth data.

This ensures transparency in system behaviour and helps assess generalizability, even without training or inference.

5.2. Real-World Consideration in Evaluation

Our system labels a slot as **occupied** even if a **non-vehicle object (e.g., trolley, delivery cart, crate)** is present in the slot. This is intentional — because from a practical standpoint, **a slot is unavailable for parking if any obstruction is present**. Thus, the system prioritizes **functional usability** over object type classification.

This design choice, while increasing some false positives (predicts occupied but actually free) compared to models that detect vehicles only, aligns more closely with **real-world user experience**: if something blocks a slot, a driver won’t be able to park there.

The only mismatch occurred is due to a trolley being left in one of the parking slots, which the system correctly interpreted as an obstruction, thus classifying it as 'occupied'. From a real-world usability standpoint, this is acceptable since the slot is functionally unavailable.

5.3. Evaluation Metrics

We used the following classification metrics to evaluate both methods:

- **Accuracy** = (TP + TN) / (TP + TN + FP + FN)
  Measures overall correctness of slot classification.
- **Precision** = TP / (TP + FP)
  Measures the proportion of slots marked as occupied that were actually occupied.
- **Recall** = TP / (TP + FN)
  Measures how many actually occupied slots were correctly identified.
- **F1-Score** = 2 × (Precision × Recall) / (Precision + Recall)
  Harmonic mean of precision and recall — useful for imbalanced classes.

Here:

- **TP** (True Positive): Correctly predicted occupied slots
- **TN** (True Negative): Correctly predicted free slots
- **FP** (False Positive): Predicted occupied but actually free
- **FN** (False Negative): Predicted free but actually occupied

5.4. Results Overview

The performance of both methods was evaluated by comparing the predicted output with manually labelled ground truth annotations. Each parking slot was classified as either **occupied** or **free**, and the predictions were assessed using standard classification metrics.

For a dataset of **400 parking slots**, the following confusion matrix values were observed for both methods:

Metric	**Pixel Thresholding**	**CCA-Based Method**
TP	274	274
TN	125	125
FP	1	1
FN	0	0
Accuracy	99.75%	99.75%
Precision	99.64%	99.64%
Recall	100%	100%
F1-Score	99.82%	99.82%

6. USER'S MANUAL TO USE THE EXECUTABLES

This section describes how to run both methods provided in the project — **Pixel Intensity Thresholding** and **Connected Component Analysis (CCA)** — using the Python scripts. Labelling and Detection methods are provided as an independent executable *.py* script that generates both visual and tabular output for parking occupancy status.

6.1. Pixel Intensity Thresholding Manual

Executables:

- *parking_slot_status_detector.py*
- *slot_label_maker.py*

Step-by-step Instructions:

1. **Input Requirements**:

   Image of the parking lot in the images folder *(images/parking_lot.png)*
2. **To Label Slots**:

   Run *slot_label_maker.py* first. Use left-click to add a label, right-click on the label to remove. Close the window to save it as *parking_slots*. The number of plots labels will be printed on the terminal.
3. **To Run the Detection**:

   Run *parking_slot_status_detector.py*
4. **Outputs Generated**:

   All the outputs will be stored in the *output* folder: It will include the *.csv* file, visual representation of the parking plot status and all the processed images.

Notes:

- Thresholds are manually defined in the code (threshold_count = 270) and may be tuned if needed.
- Works well on clean, evenly lit images.

6.2. Connected Component Analysis (CCA) Manual

Executables:

- *parking_slot_status_detector.py*
- *slot_label_maker.py*

Step-by-step Instructions:

1. **Input Requirements**:
    - Occupied image: *images/parking_lot.png*
    - Empty reference image: *images/parking_lot_empty.png*
2. **To Label Slots for CCA**:

   Run *slot_label_maker.py* in the *slots_data* folder. Use left-click to add a label, right-click on the label to remove. Close the window to save it as *parking_slots and* slots.csv. The number of plots labels will be printed on the terminal.
3. **To Run the Detection**:

   Run *parking_slot_status_detector.py*

5. **Outputs Generated**:

   All the outputs will be stored in the *output* folder: It will include the *.csv* file, visual representation of the parking plot status and all the processed images.

Notes:

- Ensure camera viewpoint is fixed and matches the reference image.
- Threshold and blob area values are tuneable via code.

7. COMPARISON WITH CUSTOM TRAINED MODELS

In modern computer vision tasks such as object detection and localization, **deep learning-based models** like **YOLO (You Only Look Once)**, **SSD (Single Shot Multibox Detector)**, and **Faster R-CNN** are widely recognized for their high accuracy and robustness. These models have demonstrated exceptional performance across real-time applications, including **vehicle detection in parking lots**, due to their ability to learn complex features from large datasets.

However, despite their advantages, deploying such deep learning models for this project was not considered for the following reasons:

7.1. Why Deep Learning Models Are Commonly Used

- **High Accuracy**: Pre-trained and custom-trained models can localize vehicles even under occlusion or varying illumination.
- **Generalization**: Capable of working in diverse environments once trained.
- **State-of-the-art Performance**: Widely used in real-world smart city solutions and parking automation systems.

7.2 Why This Project Avoided Deep Learning

Challenge	**Limitation Description**
Data Scarcity	Only one image was provided; training a deep model typically requires thousands of labeled samples.
Labeling Overhead	Custom training requires labeled bounding boxes for vehicles and parking slots, which were unavailable.
High Computational Cost	Training and even inference of deep models (especially YOLOv5+) need GPUs and long run times.
Deployment Complexity	Exporting and integrating models to lightweight environments (e.g., Raspberry Pi, Jetson Nano) involves additional dependencies.
Lack of Flexibility	Pre-trained models may not perform well on specific custom angles or environments unless fine-tuned.

7.3. Justification for Using Lightweight Classical Approaches

Instead of using deep learning, this project focused on **two traditional vision-based methods**:

- **Pixel Thresholding**
- **Connected Component Analysis (CCA)**

These techniques were:

- **Lightweight**: Run efficiently on any device without GPU acceleration.
- **Interpretable**: Each step of the pipeline is understandable and tweakable, making debugging and customization easier.
- **Deterministic**: Unlike black-box models, classical methods do not require probabilistic inference or risk unpredictable behavior.
- **Sufficiently Accurate**: For a static and semi-controlled environment like the provided image, classical methods yielded near-perfect detection.

Conclusion

While deep learning approaches have their place in production-grade systems and dynamic environments, for this challenge scenario, **traditional image processing methods offered a simpler, faster, and equally reliable solution** without the overhead of data labeling, training, or GPU deployment.

8. LIMITATIONS

While the project demonstrates strong results under controlled settings, there are certain limitations inherent to both methods used:

8.1 Shadow-based False Detections

- **Issue**: Strong shadows cast by nearby vehicles or objects may be interpreted as part of the vehicle in both methods, particularly in threshold-based processing.
- **Effect**: This can lead to overestimation of vehicle regions, causing false classification of free slots as occupied.

8.2 Occlusion and Partial Visibility

- **Issue**: If a vehicle only partially overlaps with a parking slot (e.g., bumper or tail light), the system may not detect it properly.
- **Effect**: CCA may miss small blobs or wrongly assign a slot as free.

7.3 Manual Annotation Dependency

- **Issue**: The system relies on manually labelled slot positions using a slot label maker tool.
- **Effect**: Any errors or imprecision in labelling (position, size) directly affect the output accuracy.

8.4 Limited Adaptability

- **Issue**: The current implementation assumes a fixed camera angle and fixed slot layout.
- **Effect**: Not well-suited for dynamic environments or changing viewpoints without re-labelling or re-calibration.

9. CONCLUSION

In this project, a complete parking slot detection and occupancy classification system was developed using two lightweight and interpretable image processing techniques:

- **Pixel Thresholding-based Detection**
- **Connected Component Analysis (CCA)**

Both methods were implemented using OpenCV and other Python libraries without the need for training data. The system takes a static image (or frame from a live feed) of a parking lot and determines the occupancy status of each parking slot. The results are visualized with clear bounding boxes and recorded in structured CSV reports.

This work was completed as part of the **IHub Navavishkar challenge**, where the goal was to build a robust parking analytics system using limited input data. The solution adheres to that goal by being:

- **Fast and scalable**
- **Deployable without GPU or cloud**
- **Capable of delivering competitive performance without machine learning**

With minor modifications, this pipeline can be extended to real-time camera streams, enabling real-world deployment in schools, campuses, malls, and public parking facilities.

10. REFERENCES

Libraries and Tools

- OpenCV: https://opencv.org/
- cvzone: https://github.com/cvzone/cvzone
- NumPy: https://numpy.org/
- Pandas: https://pandas.pydata.org/
- GeeksforGeeks – Computer Vision & Image Processing Section: https://www.geeksforgeeks.org/python-image-processing-tutorials/

Datasets and Guidelines

- **IHub Navavishkar Challenge Brief**: (as provided)