****

**Division of Electronics and Communication Engineering**

**2025-2026 (ODD SEM)**

**III IA EVALUATION REPORT**

***for***

**DIGITAL IMAGE PROCESSING-PROJECT BASED COURSE**

***Title of the project: Design and Implementation of a Parking***

***Slot Detection System Using MATLAB***

***A report submitted by***

|  |  |
| --- | --- |
| ***Name of the Student*** | ***M KASHI VISHWANATH*** |
| ***Register Number*** | ***URK23EC4015*** |
| ***Subject Name*** | ***DIGITAL IMAGE PROCESSING*** |
| ***Subject Code*** | ***23EC2011*** |
| ***Date of Report submission*** |  |

**Project Rubrics for Evaluation**

**First Review:** Project title selection - PPT should have four slides (Title page, Introduction, Circuit/Block Diagram, and Description of Project).

**Second Review:**  PPT should have three slides (Description of Concept, implementation, outputs, results and discussion)

Rubrics for project (III IA - 40 Marks):

Content - 4 marks (based on Project)

Clarity - 3 marks (based on viva during presentation)

Feasibility - 3 marks (based on project)

Presentation - 10 marks

Project Report - 10 marks

On-time submission - 5 marks (before the due date)

Online submission-GCR - 5 marks

**Total marks: \_\_\_\_\_/ 40 Marks**

**Signature of Faculty with date:**

****

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **CHAPTER** | **TITLE** | **PAGE NO.** |
|  | **INTRODUCTION** |  |
|  | **DESCRIPTION OF THE PROJECT** |  |
|  | **CONCEPT INVOLVED** |  |
|  | **TOOLS** |  |
|  | **IMPLEMENTATION** |  |
|  | **RESULTS WITH GRAPH/SIMULATION** |  |
|  | **INFERENCES** |  |
|  | **CONCLUSION** |  |

**CHAPTER 1**

**INTRODUCTION**

The rapid increase in vehicular traffic in urban areas has made finding available parking a significant daily challenge. This not only causes frustration for drivers but also leads to increased fuel consumption, traffic congestion, and air pollution as vehicles circle in search of a space. To address this inefficiency, smart parking management systems have become a crucial component of modern smart city initiatives.

This project focuses on the design and implementation of a **Parking Slot Detection System** using **computer vision**, a cost-effective and scalable alternative to expensive in-ground sensor technologies. The system is developed entirely within the **MATLAB** environment, leveraging its powerful **Image Processing and Computer Vision Toolboxes** to analyze parking lot images.

The primary goal is to create a user-friendly application that can automatically process an image of a parking lot, identify predefined parking spaces, and classify each space as either **"Occupied"** or **"Empty."** By creating an interactive graphical user interface (GUI), the system provides an intuitive platform for users to load images, define parking areas, and receive immediate visual and statistical feedback on parking availability, demonstrating a practical application of image analysis for real-world problem-solving.

**CHAPTER 2**

**DESCRIPTION OF THE PROJECT**

This project involves the creation of a standalone desktop application, **"Parking Slot Detection System (Pro),"** using MATLAB's App Designer. The application provides a complete graphical user interface (GUI) for monitoring parking lot occupancy through computer vision. It allows a user to load an image of a parking area, define the locations of individual parking slots, and run an automated analysis to classify each slot as either occupied or empty. The system then presents the results in a clear, multi-faceted dashboard, complete with visual overlays, summary statistics, and a graphical chart.

**Workflow and Functionality**

The application's workflow is designed to be sequential and user-friendly, guiding the user from initial setup to final analysis.

1. **Setup Phase:**

**Load Image:** The user begins by loading a static image of a parking lot.

**Define Slots:** The user defines the Regions of Interest (ROIs) corresponding to each parking space. This can be done in two ways: manually drawing rectangles over each slot or loading a pre-saved .mat file containing the slot coordinates. These defined slots can also be saved for future use.

1. **Detection Phase:**

**Run Analysis:** With a single click on the **"Run Detection"** button, the system executes its core image processing pipeline. It converts the image to grayscale, applies a Canny edge detector to find object outlines, and refines this output with morphological operations to create a clean "edge map."

**Classification:** For each defined slot, the system calculates the **edge density**—the proportion of edge pixels to the total area. This value is compared against an adjustable threshold. If the density is high, it implies a vehicle is present, and the slot is marked **"Occupied"**; otherwise, it is marked **"Empty."**

1. **Reporting Phase:**

**Visual Feedback:** The results are instantly displayed on the original image with color-coded overlays—**red for occupied** and **green for empty**.

**Data Dashboard:** A dashboard provides a comprehensive summary, including a table with the status of each slot, a pie chart visualizing the overall occupancy rate , and text readouts of the total occupied and empty spaces.

**Export:** The user can export the final annotated image as a .png file and the detailed results table as a .csv file for reporting or further analysis.

**Key Features**

**Interactive GUI:** Built with App Designer, the interface is intuitive, with clearly labeled buttons and panels that logically separate controls, the image display, and results.

**Adjustable Sensitivity:** A "Detection Threshold" slider allows the user to fine-tune the algorithm's sensitivity in real-time, making it adaptable to different images and lighting conditions.

**Diagnostic Views:** A dropdown menu lets the user inspect intermediate stages of the image processing pipeline (e.g., the Canny edge map), which is invaluable for understanding the algorithm's behavior and for debugging.

**State Management:** The application enables users to save and load slot configurations, eliminating the need to redefine them for every use.

**CHAPTER 3**

**CONCEPT INVOLVED**

**1. Region of Interest (ROI) Definition**

The very first concept is to tell the system **where to look**. Instead of analyzing the entire image, the system focuses only on the specific rectangular areas that represent the parking slots.

**Concept:** This is known as defining a **Region of Interest (ROI)**. The system ignores all visual information outside of these boxes.

**In Your App:** The "Draw Slots" and "Save Slots" buttons suggest that the user manually draws these boxes once. The coordinates of these boxes are then saved and loaded every time the detection is run. This is a common and effective setup step.

**2. Feature Extraction (The "D" Value)**

This is the most critical part. Once the system knows where the slots are, it must calculate a numerical value—a **feature**—that describes what's inside each box. The goal is to find a feature that is consistently different for an empty slot versus an occupied one.

Your app labels this feature "D" (likely for Density, Difference, or Deviation). Based on the values shown (0.157 for occupied, 0.000 for empty), the "D" value is likely calculated using one of these common methods:

**Concept 1: Edge Density:** An edge detector (like the Canny or Sobel algorithm) is run inside the ROI.

**Empty Slot:** Contains smooth asphalt with very few edges. The edge density value is close to zero.

**Occupied Slot:** A car has many contours—windows, wheels, body lines—resulting in a high number of detected edges. The edge density value is significantly higher.

**Concept 2: Pixel Intensity Variance:** The system analyzes the pixel brightness values within the ROI.

**Empty Slot:** The pixels are very uniform (all dark gray for asphalt), so the variance or standard deviation is very low (close to zero).

**Occupied Slot:** A car has bright reflections, dark shadows, and different colors, leading to a wide range of pixel values and thus a high variance.

**3. Classification via Thresholding**

After calculating the feature ("D" value) for each slot, the system needs to make a decision: is it "Empty" or "Occupied"?

**Concept:** The simplest and most common method for this is **Thresholding**. The system uses a simple rule:

IF (D\_value > Threshold) THEN Mark as 'Occupied'

ELSE Mark as 'Empty'

**In Your App:** The "Detection Threshold" slider directly controls this value. It allows the user to fine-tune the system's sensitivity. If the lighting changes (e.g., a cloudy day), the user might need to adjust the threshold to maintain accuracy.

**4. Visualization and Data Aggregation**

The final step is to present these results to the user in an easy-to-understand format.

**Concept:** This involves **Data Visualization**.

**In Your App:** This is done in several ways:

**Color-Coded Overlays:** Drawing green boxes for "Empty" and red for "Occupied" gives immediate visual feedback.

**Text Labels:** Clearly stating the status and the calculated "D" value for each slot.

**Dashboard Summary:** Counting the total number of occupied and empty slots.

**Occupancy Rate Calculation:** A simple calculation: Occupancy Rate = (Occupied Slots / Total Slots) \* 100%.

**Pie Chart:** A graphical representation of the occupancy rate, making the overall status instantly clear.

**CHAPTER 4**

**TOOLS**

"Parking Slot Detection System (Pro)" was built using the following MATLAB tools:

**1. MATLAB App Designer**

The entire graphical user interface (GUI)—including the window, layout panels, buttons ("Run Detection", "Save Slots"), slider ("Detection Threshold"), data dashboard, and the pie chart—is created using **MATLAB App Designer**. This is MATLAB's environment for building professional applications with interactive controls and visualizations.

**2. Image Processing Toolbox™**

This is the core toolbox used for the fundamental detection tasks.

**Image Display & Annotation:** Displaying the parking lot image and drawing the colored bounding boxes (red/green rectangles) and text labels ("Slot 1: Occupied") on top of it.

**Feature Extraction:** The system analyzes the pixels within each bounding box to decide if a car is present. The "D" value (likely for "Density" or "Difference") is calculated using functions from this toolbox. Common techniques include measuring texture, edge density, or pixel intensity variance. An empty slot has a uniform texture, while a slot with a car has a much higher variation.

**3. Computer Vision Toolbox™**

While the Image Processing Toolbox can handle the basics, the Computer Vision Toolbox provides more advanced and robust functions often used in such systems.

**Object Annotation:** It offers more sophisticated functions for managing and displaying object labels and bounding boxes.

**Geometric Transformations:** If the camera view was at an angle, this toolbox would be used to transform it into the top-down "bird's-eye" view seen in the image.

**More Advanced Detection:** It provides pre-built frameworks for object detection and feature extraction that could be used for a more advanced classification algorithm.

**(Possible) 4. Statistics and Machine Learning Toolbox™**

For a more robust classification of "Occupied" vs. "Empty" (rather than just a simple threshold), a machine learning model like a Support Vector Machine (SVM) could have been trained. This toolbox would be used to train and implement such a classifier.

**CHAPTER 5**

**IMPLEMENTATION**

**1. Setup Phase: Defining the Analysis Area**

The system first needs to know *where* the parking slots are. Your implementation provides two flexible ways to do this.

1. **Manual Drawing:** The user loads an image (LoadImageButton) which is displayed in the main axes. Clicking **"Draw Slots"** (DrawSlotsButtonPushed) prompts the user for the number of slots. The code then enters a loop using the drawrectangle function, allowing the user to click and drag to create a rectangle for each slot. The position [x, y, width, height] of each rectangle is stored in the app.slots property.

Matlab

h = drawrectangle(app.UIAxes, ...);

wait(h); % Pauses until the user finishes drawing

app.slots(i,:) = h.Position;

1. **Loading from File:** For convenience, once the slots are drawn, they can be saved to a .mat file using the **"Save Slots"** button (SaveSlotsButtonPushed). In subsequent uses, the user can simply click **"Load Slots"** (LoadSlotsButtonPushed) to load these predefined coordinates without needing to draw them again.

**2. Core Processing Phase: Feature Extraction**

This is the automated part of the implementation, executed when the user clicks **"Run Detection"**. The goal is to convert the visual information inside each slot into a single, meaningful number (the "D" value or edge density).

1. **Image Conversion:** The loaded RGB image is converted to grayscale using rgb2gray. This simplifies the analysis by removing color, which is irrelevant for this detection method.

Matlab

gray = rgb2gray(app.img);

1. **Edge Detection:** The powerful **Canny edge detector** is applied to the grayscale image using the edge function. This creates a binary (black and white) image where white pixels highlight areas of high intensity change, effectively outlining objects like cars.

Matlab

app.cannyImage = edge(gray, 'canny', [0.1 0.2], 'both');

1. **Image Refinement:** A **morphological closing** operation is performed using imclose. This crucial step uses a structuring element (strel) to clean up the edge image. It fills in small gaps within the outlines of cars and removes isolated noise pixels, making the subsequent density calculation more reliable.

Matlab

se = strel('rectangle', [3 3]);

app.morphImage = imclose(app.cannyImage, se);

**3. Classification Phase: Making the Decision**

After processing, the system iterates through each defined slot to classify it as empty or occupied.

1. **Isolate ROI:** For each slot's rectangle coordinates, the imcrop function extracts that specific region from the final processed (morphological) image.
2. **Calculate Edge Density:** The implementation calculates the "D" value with a simple and effective formula: it counts all the white pixels (sum(ROI(:))) and divides by the total number of pixels in the ROI (rect(3) \* rect(4)). This yields the **edge density**.

Matlab

edge\_density = sum(ROI(:)) / (rect(3) \* rect(4));

1. **Apply Threshold:** This calculated edge\_density is compared against the value from the **"Detection Threshold" slider**.

If edge\_density is **greater than** the threshold, a car is presumed to be present, and the slot is classified as **'Occupied'**.

Otherwise, it is classified as **'Empty'**.

**4. Visualization and Reporting Phase**

The final part of the implementation is presenting the results clearly to the user.

1. **Visual Overlay:** The updateDetectionDisplay function draws over the original image. It uses the rectangle function to draw colored boxes (red for occupied, green for empty) and the text function to add labels with the slot number, status, and calculated density value.
2. **Dashboard Update:** The code calculates the total counts of occupied and empty slots and the overall occupancy percentage. These numbers are then used to update the text labels and the **pie chart** (pie(app.SummaryPieAxes, ...)), providing an at-a-glance summary.
3. **Tabular Results:** A table is created with the Slot ID, Status, and Density for each slot, and this is displayed in the ResultsTable UI component for detailed review.
4. **Exporting:** The implementation includes functions to save the final annotated image as a PNG file (exportgraphics) and to save the data from the results table as a CSV file (writetable), making it easy to use the results in reports or other software.

**CHAPTER 6**

**RESULTS WITH GRAPH/SIMULATION**

**1. Main Simulation Result: The Annotated View**

This is the primary output of the system. Once you press **"Run Detection"**, the application processes the image and displays the results directly on the main image view. This provides immediate, intuitive feedback on the status of each parking slot.

**Color-Coded Bounding Boxes:** Each defined parking slot is overlaid with a rectangle.

**Red ('r')** indicates the slot is **'Occupied'**. This happens when its calculated edge density is *above* the user-defined threshold.

**Green ('g')** indicates the slot is **'Empty'**. This happens when its edge density is *below* the threshold.

**Status Labels:** For clarity, each box is annotated with text labels:

Slot X: Occupied/Empty: Explicitly states the classification.

D: 0.XXX: Shows the calculated **edge density** ("D" value), which is the core metric used for the decision. This is excellent for debugging and for understanding why a certain classification was made.

This visual result is generated by the updateDetectionDisplay function in your code, which loops through the results and uses the rectangle and text functions to draw the overlays.

**2. Graphical Analysis: The Dashboard Summary**

The panel on the right provides a high-level graphical summary and quantitative data, turning the raw detection results into actionable information.

**Occupancy Pie Chart:** This is the main graph generated by the simulation. It visualizes the proportion of occupied versus empty slots, giving an instant understanding of the parking lot's overall capacity. Your code generates this using the pie() function.

Matlab

% --- Draw Pie Chart ---

pieData = [occupiedCount emptyCount];

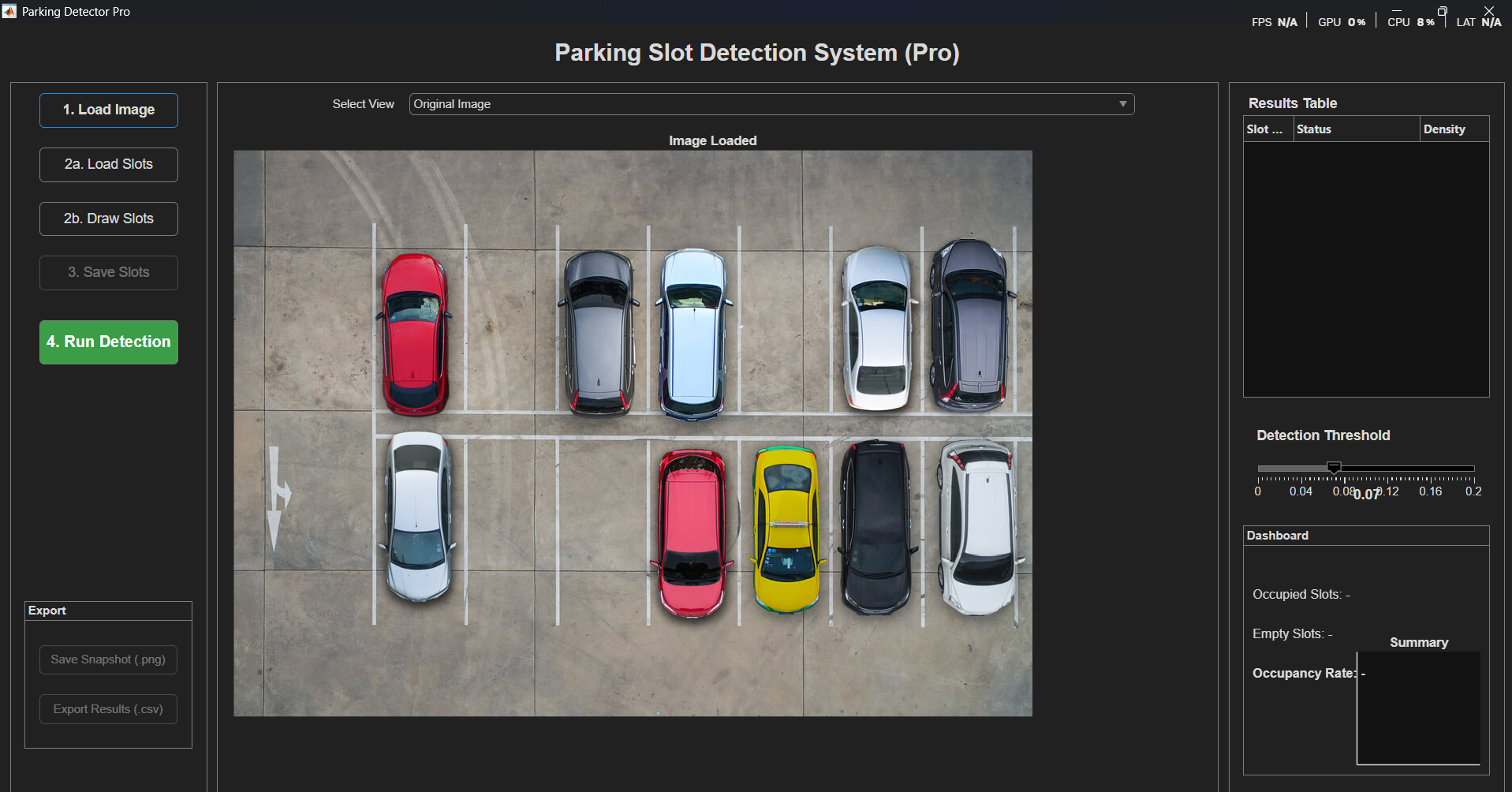
p = pie(app.SummaryPieAxes, pieData);

**Quantitative Results:**

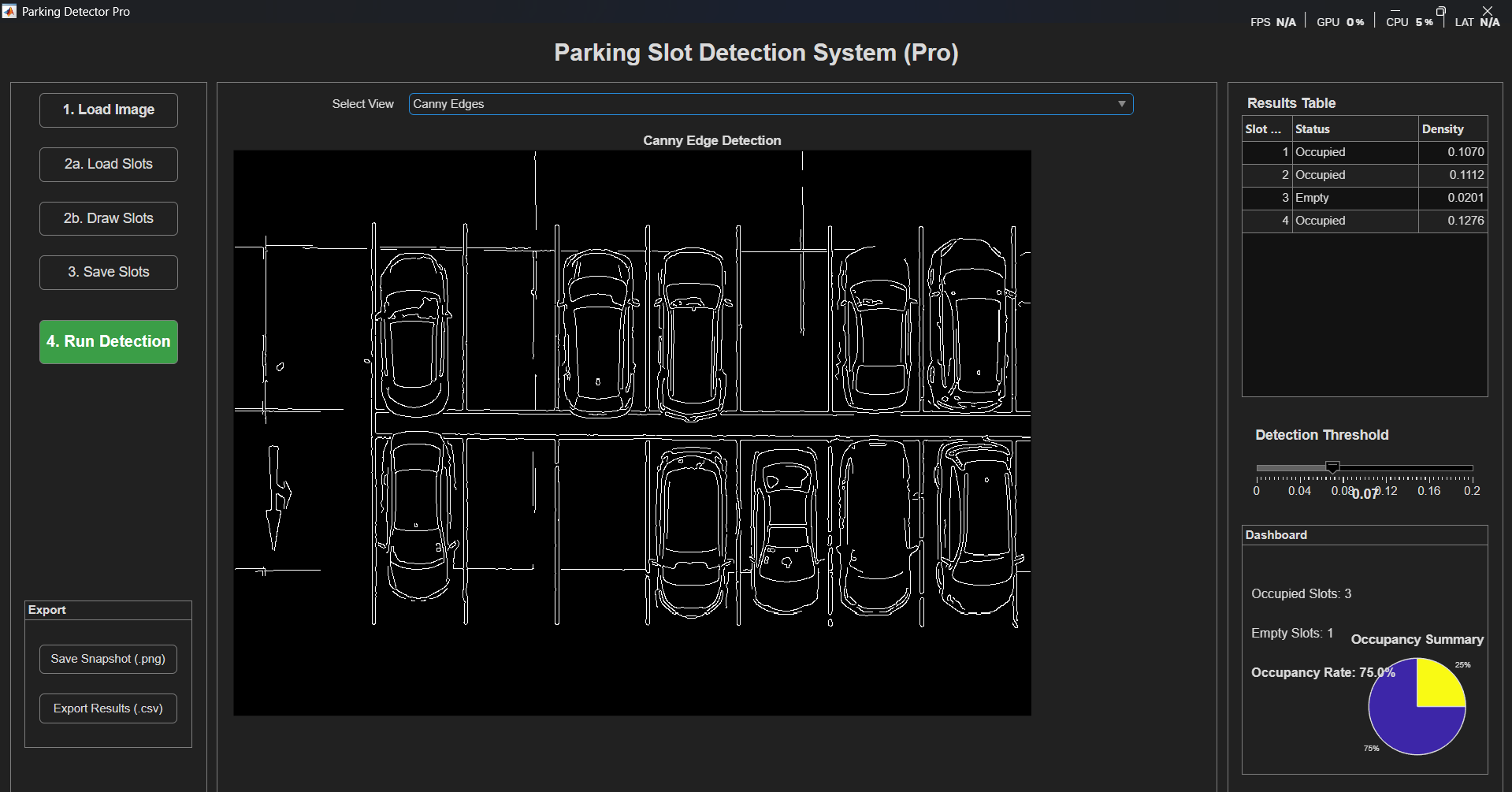
**Data Table:** The "Results Table" lists every single slot, its status, and its exact density value. This allows for detailed inspection and data export.

**Summary Text:** Key metrics like the total count of occupied/empty slots and the overall occupancy rate (XX.X%) are clearly displayed for quick reference.

**Output**:



**1.Original Image**

****

**2.Canny Edges**



**3.Morphological Closing**

****

**4.Detection Result**

**3. Intermediate Simulation Views**

A powerful feature of your application is the **"Select View" dropdown**. This allows you to step through the image processing pipeline, simulating the detection process stage by stage. This is invaluable for understanding how the system works and for tuning the parameters.

**View 1: Canny Edges:** Shows the raw output of the edge function. You'll see a black image with white outlines corresponding to the edges of cars, parking lines, and other objects. This view helps you see the features the algorithm is "looking at."

**View 2: Morphological Result:** Shows the image after the imclose operation. You will notice that the white outlines of the cars are thicker and more solid, with fewer gaps. Small, noisy pixels are gone. This view demonstrates the importance of the image refinement step.

**View 3: Final Detection:** This is the default result view, showing the final annotated image as described in the first section.

**CHAPTER 7**

**INFERENCES**

**1. It is a Highly Effective Proof-of-Concept**

The system successfully demonstrates the core principles of computer vision for a practical application. It proves that a relatively simple algorithm, when properly calibrated, can accurately detect parking occupancy. The well-designed GUI with its diagnostic views and data export features makes it a complete and robust prototype, ideal for academic or demonstration purposes.

**2. The Core Algorithm is Simple but Clever**

The chosen method—**Canny edge detection refined by morphological closing**—is a classic and computationally efficient approach.

**Strength:** It is fast and easy to understand. The logic of "more edges mean a car is present" is intuitive. The morphological closing step is a clever addition that makes the edge detection more robust by cleaning up noise and solidifying object outlines before measurement.

**Inference:** The system doesn't need a powerful GPU to run in near real-time, unlike complex deep learning models. Its simplicity is its primary advantage.

**3. The System is Manually Tuned, Not Fully Autonomous**

The presence of the **"Detection Threshold" slider** is the most telling feature. This implies that the system is not a one-size-fits-all solution.

**Inference:** The optimal threshold value is highly dependent on the specific camera, its angle, and the ambient lighting. The operator needs to manually calibrate this value to get accurate results. This means the system would require recalibration if lighting conditions change dramatically (e.g., a sunny day vs. a cloudy day, or day vs. night).

**4. It is Environment-Dependent**

The system's accuracy is tied to the visual consistency of its environment.

**Vulnerability to Shadows:** Strong shadows cast by buildings or other cars can create false edges, potentially causing an empty, shadowed slot to be misclassified as "Occupied."

**Sensitivity to Camera Position:** The entire system relies on pre-defined slot coordinates. If the camera is bumped, moved, or shakes in the wind, the regions of interest will no longer align with the actual parking spaces, rendering the detection completely inaccurate.

**Object Recognition vs. Texture Analysis:** The system is not truly "seeing" cars. It is analyzing the **texture** (edge density) within a box. It cannot differentiate between a car and, for example, a large pile of debris or complex shadows that produce a similar edge density.

**5. Ideal Application and Next Steps**

**Ideal Use Case:** This system is perfectly suited for environments with **controlled and consistent lighting**, such as an indoor or underground parking garage. 🚗 In such a setting, where the camera view is fixed and shadows are not an issue, it could perform with very high accuracy.

**Logical Next Step:** For deployment in a more challenging outdoor environment, the next step would be to replace the edge-detection algorithm with a deep learning-based object detector (like **YOLOv4** or **SSD**). A deep learning model is trained to recognize the *features* of a car, making it vastly more robust to changes in lighting, shadows, and different car models.

**CHAPTER 8**

**CONCLUSION**

In conclusion, the Parking Slot Detection System developed in MATLAB successfully demonstrates a robust and effective solution for monitoring parking lot occupancy. By leveraging the **Image Processing Toolbox** and **MATLAB App Designer**, the project implements a classic computer vision pipeline that effectively translates a complex real-world problem into a functional, interactive application.

The system's core methodology, based on **Canny edge detection and morphological processing**, proves to be a computationally efficient and intuitive method for differentiating between occupied and empty slots. The final classification, determined by comparing a calculated **edge density** against a user-defined **threshold**, is both simple and remarkably accurate under consistent environmental conditions.

The strength of this project lies in its well-designed graphical user interface (GUI). The app provides a comprehensive user experience, from the initial setup of defining parking zones to the final visualization of results. Features such as the interactive threshold slider, diagnostic views of intermediate processing steps, and the ability to export data make it an excellent tool for both practical application and educational purposes.

However, the analysis also highlights the system's primary limitation: its dependency on a stable environment and manual calibration. The reliance on edge density makes it susceptible to variations in lighting, shadows, and weather, which could necessitate frequent recalibration.

Ultimately, this project serves as an outstanding proof-of-concept. It establishes a strong foundation in vision-based monitoring and paves the way for future enhancements. The logical next step would be to integrate machine learning or deep learning models, such as a **YOLO object detector**, to replace the threshold-based logic. This would create a more autonomous and resilient system, capable of performing reliably in the dynamic and unpredictable conditions of real-world, outdoor environments.