**Warehouse Stocks Monitoring**

A PROJECT REPORT SUBMITTED BY

W.N.N.V.BANDARA

( S / 18 / 326 )

for the partial fulfillment for the requirement for the course of

CS 314 Image Processing Practical

To the **Department of Statistics and Computer Science** of the

FACULTY OF SCIENCE

UNIVERSITY OF PERADENIYA

SRI LANKA

2022

# DECLARATION

I do hereby declare that the work reported in this project report was exclusively carried out by me under the supervision of Prof. Amalka Pinidiyaarachchi.. It describes the results of my own independent work except where due reference has been made in the text. No part of this project report has been submitted earlier or concurrently for the same or any other degree.

Date: ……………………. ………………………………………

Signature of the Candidate

Certified by:

1. Instructor: **Mr. Rochana Obadage**

Date: ……………………. Signature:…………………………….

1. Supervisor: **Prof. Amalka Pinidiyaarachchi**

Date: ……………………. Signature:…………………………….

1. Head of the Department: **Dr. Sachith Abeysundara**

Date: ……………………. Signature:…………………………….

# ABSTRACT

*In today's dynamic business landscape, the effective management of warehouse stocks is pivotal for ensuring operational efficiency, cost savings, and customer satisfaction. the abstract of the system described outlines the development of a sophisticated warehouse management tool that leverages computer vision technology to enhance warehouse operations. This system utilizes strategically positioned cameras to detect and categorize various objects on warehouse shelves, with a key focus on accurately identifying storage boxes, regardless of their color. Real-time data processing ensures continuous updates to inventory levels and the detection of vacant storage spaces. The tool integrates seamlessly with existing warehouse management systems, providing a user-friendly interface for monitoring and interaction. By automating the tracking of inventory and storage availability, this system promises to significantly improve warehouse efficiency, reduce errors, and streamline inventory management processes.*

**ACKNOWLEDGEMENTS**

I would like to acknowledge all those without whom this project would not have been successful. Firstly, I would like to thank Prof. Amalka Pinidiyaarachchi who gave the opportunity to apply the Digital Image Processing concept on an individual project and giving us all the theoratical and practical knowledge on it. She is one of the main guider and motivator to manage the project work with all the other academic works. I would also like to extend my thanks to Mr.Rochana Obadage for providing the practical knowledge, support, guidance and hands-on experience on Digital Image Processing laboratory works. He made us understand how to complete this project successfully. I would like to extend my sincere gratitude to Dr. Sachith Abeysundara, the Head of the department of Statistics and Computer Science for providing the facilities for successfully completing this project.

**Table of Contents**

[**DECLARATION** 2](#_Toc17063162)

[**ABSTRACT** 4](#_Toc17063163)

**1.** [**INTRODUCTION** 1](#_Toc17063165)

[1.1. Introduction 1](#_Toc17063166)

[1.2. Problem Statement 1](#_Toc17063167)

[1.3. Solution 2](#_Toc17063168)

**2.** [**RELATED WORK** 3](#_Toc17063170)

**3.** [**METHODOLOGY** 5](#_Toc17063172)

[3.1. Image Processing Pipeline 5](#_Toc17063173)

[3.2. Used Functions and Their Applications 6](#_Toc17063174)

[3.3. Used Tools , Technologies and Libraries 8](#_Toc17063175)

[3.4. Procedure 9](#_Toc17063176)

[3.5. Procedure Explained (Screenshots) 9](#_Toc17063177)

**4.** [**RESULTS AND DISCUSSION** 34](#_Toc17063179)

[4.1. Major Resulting Steps 34](#_Toc17063180)

[4.2. Some tested image results 36](#_Toc17063181)

[4.3. Discussion 38](#_Toc17063182)

**5.** [**CONCLUSIONS** 39](#_Toc17063184)

[5.1. Advantages of the Implemented system 39](#_Toc17063185)

[5.2. Issues 41](#_Toc17063186)

[5.3. Conclusion 42](#_Toc17063187)

[**REFERENCES** 43](#_Toc17063189)

**LIST OF FIGURES**

Figure 3.1.1-Image Processing Pipeline

Figure 3.5.5.1-Rectangle shape shelves

Figure 3.5.5.2-Squar shape shelves

Figure 3.5.2.1-console

Figure 3.5.3.1-part of code

Figure 3.5.4.1-part of code

Figure 3.5.4.2-part of code

Figure 3.5.4.3-part of code

Figure 3.5.4.4-part of code

Figure 3.5.4.5-Select area of ROI

Figure 3.5.4.6-Cropped ROI

Figure 3.5.4.7-part of code

Figure 3.5.4.8-Converts ROI Grayscale

Figure 3.5.4.9-Kernel

Figure 3.5.4.10-Blur

Figure 3.5.4.11-Thresh

Figure 3.5.4.12-Dilation

Figure 3.5.4.13-Binary thresholding operation is applied to the dilated image

Figure 3.5.4.14-part of code

Figure 3.5.4.15-part of code

Figure 3.5.4.16-Blur

Figure 3.5.4.17-BGR

Figure 3.5.4.18-Available Rectangle Shape Shelves

Figure 3.5.4.19-part of code

Figure 3.5.4.20-part of code

Figure 3.5.4.21-part of code

Figure 3.5.5.1-Available Rectangle Shape small Shelves

Figure 3.5.5.2-ROI

Figure 3.5.5.3-Empty slots detected image

Figure 3.5.5.4-console

Figure 4.1.1.1-Available Rectangle Shape small Shelves

Figure 4.1.1.2-ROI

Figure 4.1.1.3-Empty slots detected image

Figure 4.1.1.4-console

Figure 4.1.2.1-Available Squar Shape large Shelves

Figure 4.1.2.2-ROI

Figure 4.1.2.3-Empty slots detected image

Figure 4.1.2.4-console

Figure 4.2.1-Rectangle Shape large Shelves

Figure 4.2.2-ROI

Figure 4.2.3-Empty slots detected image

Figure 4.2.4-Available slots detected image

Figure 4.2.5-console

Figure 4.3.1-wrong camera setup

Figure 4.3.2-correct camera setup

**LIST OF TABLES**

**LIST OF ABBREVIATIONS**

openCV- Open Source Computer Vision Library

NumPy - Numerical Python

cmd - Command Line

WSMS-Warehouse Stocks Monitoring System

RFID -Radio-Frequency Identification

ROI -Region of Interest

**CHAPTER 01**

# INTRODUCTION

## **INTRODUCTION**

The Warehouse Stocks Monitoring System (WSMS) is a comprehensive solution designed to address the challenges faced by modern warehouses in managing and monitoring their inventory efficiently. With the advent of e-commerce and the increasing complexity of supply chain operations, it has become imperative for warehouses to adopt advanced technologies to streamline their inventory management processes.

This system leverages cutting-edge image processing techniques to automate the tracking, monitoring, and management of warehouse stocks, ensuring accuracy, efficiency, and real-time visibility into inventory levels.

* 1. **PROBLEM STATEMENT**

Warehouses are tasked with managing vast quantities of goods, often across multiple locations, making inventory management a complex and error-prone process. Traditional methods, such as manual counting or barcode scanning, are time-consuming, labor-intensive, and prone to human errors. These challenges lead to several problems, including:

**Inventory Inaccuracy:** Manual inventory counts and data entry errors can result in inaccurate stock levels, leading to overstocking or stockouts.

**Operational Inefficiency:** Traditional inventory management methods are slow and can slow down warehouse operations, leading to increased costs and delays.

**Loss Prevention:** Detecting theft or misplacement of goods is difficult without real-time monitoring and auditing.

* 1. **SOLUTION**

The Warehouse Stocks Monitoring System employs advanced image processing techniques to tackle the problems mentioned above. The key components of the solution include:

**Camera Integration:** High-resolution cameras are strategically placed throughout the warehouse to capture images of the shelves and storage areas. These cameras can be fixed or mounted on automated systems.

**Image Processing Algorithms:** Image processing algorithms are used to analyze the images captured by the cameras. These algorithms can identify items on shelves, determine their quantity, and track their movement.

**Real-time Data Processing:** The system processes the image data in real-time, updating the inventory database instantaneously. This provides warehouse managers with real-time visibility into stock levels.

**Machine Learning:** Machine learning models can be trained to recognize different products and their packaging variations. This enhances the system's ability to accurately identify items.

**Alerts and Notifications:** The system can send alerts and notifications to warehouse managers in case of stock discrepancies, potential theft, or low-stock levels. This enables proactive decision-making.

**Integration with Inventory Software:** The WSMS can be integrated with existing inventory management software, providing a seamless solution that automates the entire inventory management process.

**CHAPTER 02**

# RELATED WORK

Related work in the field of Warehouse Stocks Monitoring Systems (WSMS) encompasses a diverse range of technologies and methodologies. One prominent area of research and development focuses on image-based inventory management systems. These systems leverage cameras, computer vision, and deep learning techniques to monitor and manage warehouse inventory efficiently. Notable examples include Amazon's automated warehouses, where robots equipped with cameras play a crucial role in inventory control, ensuring accurate and timely order fulfillment.

RFID (Radio-Frequency Identification) technology has also been extensively explored as a means to track and manage warehouse inventory. RFID tags affixed to items and RFID readers strategically placed within the warehouse provide real-time visibility into the location and status of each item. This technology has the potential to greatly enhance inventory accuracy and streamline supply chain operations.

In the context of IoT (Internet of Things), warehouses have witnessed the deployment of various sensor-based solutions. These include smart shelves, temperature sensors, humidity sensors, and motion detectors, all interconnected to form an intelligent warehouse ecosystem. IoT data is instrumental in monitoring inventory conditions and ensuring the quality and safety of stored goods.

Barcode and QR code scanning systems remain foundational in many warehouse operations. Mobile devices equipped with scanning apps simplify the tracking and management of inventory items. Researchers and practitioners continually seek ways to optimize these systems for improved efficiency and accuracy.

Data analytics and predictive modeling have emerged as indispensable tools for optimizing warehouse inventory management. By analyzing historical data, demand patterns, and seasonality, businesses can make informed decisions about inventory levels, reorder points, and supply chain strategies. Machine learning and AI algorithms are increasingly utilized to make predictions and automate decision-making processes.

Moreover, robotic automation has gained prominence in modern warehousing. Robots equipped with sensors, cameras, and advanced navigation capabilities are deployed for tasks such as order picking and inventory replenishment. Research in this domain focuses on enhancing the efficiency and safety of robotic warehouse systems.

The integration of cloud-based inventory management systems has revolutionized inventory control by providing real-time access to data from anywhere. These scalable systems are especially valuable for businesses with multiple warehouses, enabling multi-location inventory tracking and seamless integration with other supply chain software.

Inventory security and anti-theft solutions are also integral components of warehouse management. Surveillance systems, access control mechanisms, and biometric authentication methods are deployed to safeguard valuable inventory assets. These security measures play a vital role in preventing theft and unauthorized access in warehouse environments.

Additionally, case studies and industry-specific applications continually demonstrate the successful implementation of various inventory management technologies in real-world warehouse settings. These practical examples showcase the effectiveness of innovative solutions and inspire further advancements in the field of warehouse stocks monitoring.

**CHAPTER 03**

# METHODOLOGY

**3.1. Image Processing Pipeline**

An image processing pipeline refers to a series of interconnected steps or processes used to analyze and manipulate digital images. This pipeline is designed to achieve specific objectives, such as enhancing image quality, extracting valuable information, or preparing images for further analysis or visualization. Below is a typical image processing pipeline, outlining the key stages involved:

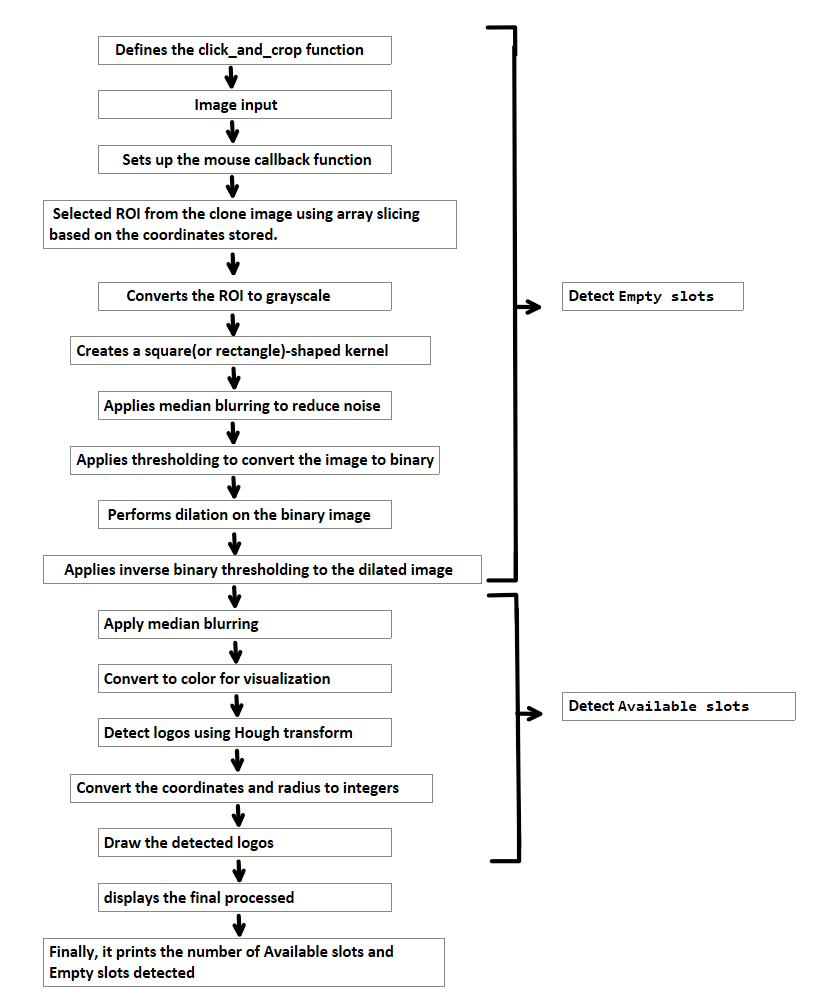
****

Figure 3.1.1

**3.2. Used Functions and Their Applications**

Several functions from the OpenCV library (cv2) and NumPy are used for image processing and analysis. Here are the used functions and their applications within the code:

**cv2.imread('B5.jpg'):**

Application: Loads an image from a file ('B5.jpg') for processing.

**cv2.namedWindow():**

Application: Creates named windows to display images with specified properties (e.g., cv2.WINDOW\_KEEPRATIO).

**cv2.setMouseCallback():**

Application: Sets a mouse callback function to respond to mouse events in the image window.

**cv2.rectangle():**

Application: Draws a rectangle on an image, used for marking the selected region of interest (ROI).

**cv2.imshow():**

Application: Displays an image in a window.

**cv2.waitKey():**

Application: Waits for a keyboard event, useful for controlling the execution flow.

**cv2.resizeWindow():**

Application: Resizes the image window for displaying the ROI and processed images.

**cv2.cvtColor():**

Application: Converts the color space of an image (e.g., BGR to grayscale).

**cv2.medianBlur():**

Application: Applies median blur to reduce noise in the ROI.

**cv2.threshold():**

Application: Converts a grayscale image to binary using a specified threshold value.

**cv2.dilate():**

Application: Dilates binary images to enhance features.

**cv2.findContours():**

Application: Detects and extracts contours in binary images.

**cv2.circle():**

Application: Draws circles on an image, used for marking detected logos or objects.

**cv2.HoughCircles():**

Application: Detects circles (logos) in the grayscale ROI using the Hough Circle Transform.

**np.uint16():**

Application: Converts data to the 16-bit unsigned integer data type.

**np.around():**

Application: Rounds floating-point numbers to the nearest integer.

**3.3. Used Tools , Technologies and Libraries**

**openCV:** which is an open-source library of programming functions mainly focused on real-time computer vision. openCV supports many programming languages like Python, C++, Java etc. Python programming language has been used in this project.

Application: OpenCV is a popular library for computer vision and image processing tasks. It provides functions for image loading, manipulation, filtering, and contour detection, among others.

**Numerical Python(NumPy):** Which is an open source Python library used for working with arrays.Also works in Fourier Transform, Linear Algebra and matrices as well.

Application: NumPy is used for numerical operations and manipulation of arrays, which is essential for various image processing tasks such as thresholding and creating data structures.

**Matplotlib:** Which is an open source graph plotting library that serves as a visualization utility.

**Pyplot:** which is a submodule of Matplotlib and most of the Matplotlib utilizes under it. Usually it is referred to as plt

**Argparse (Command-Line Argument Parsing):**

Application: argparse is a Python module used for parsing command-line arguments. In this code, it allows you to specify the input image file ('B5.jpg') from the command line.

**3.4. Procedure**

3.4.1. Get The Image

3.4.2. Setting up The Environment

3.4.3. Importing Libraries into The Python Shell

3.4.4. Used Functions and Their Applications

3.4.5. Display The Final Output

**3.5. Procedure Explained (Screenshots)**

**3.5.1. Get The Image**

all datasets are created by me using photoshop

**Rectangle shape shelves**

****

Figure 3.5.1.1

**Squar shape shelves**

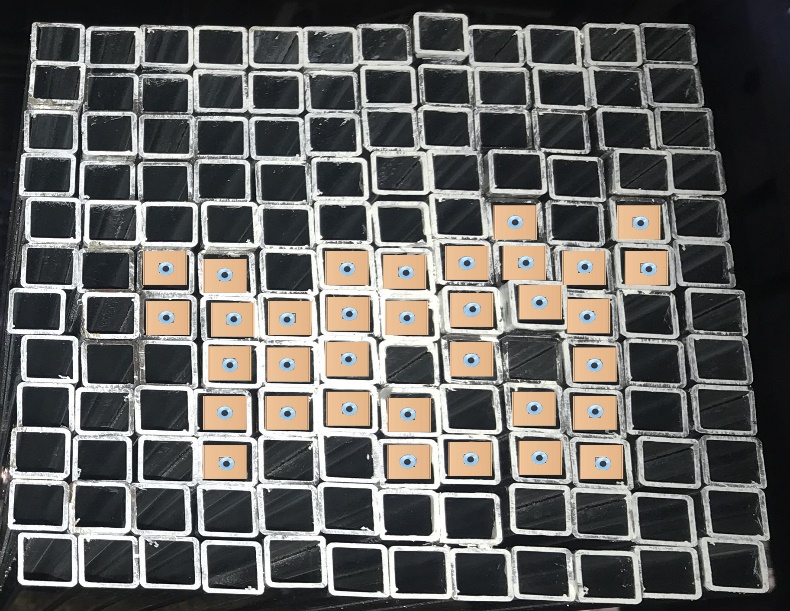
****

Figure 3.5.1.2

**3.5.2. Setting up The Environment**

First, check whether the environment had been already installed Python by typing

python –version in the command line(cmd). If Python is already installed, it will

generate a message with the Python version available. If not install Python.



Figure 3.5.2.1

To install the openCV, run the following command in cmd :

pip install opencv-python

To install the numpy, run the following command in cmd :

pip install numpy

To install the matplotlib, run the following command in cmd :

pip install matplotlib

To install the argparse, run the following command in cmd :

pip install argparse

**3.5.3. Importing Libraries Into The Python Shell**

Import the following packages and libraries which are really useful in getting the final desired output.

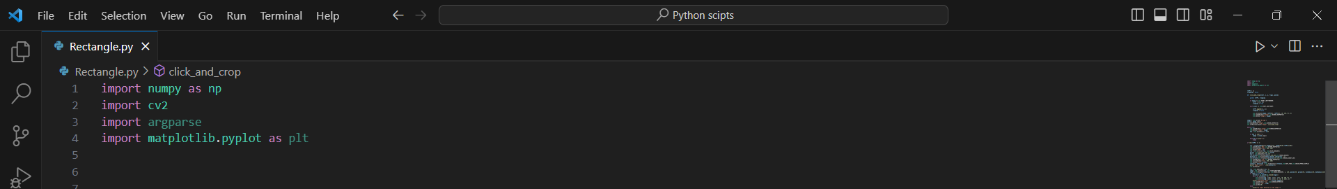


Figure 3.5.3.1

**3.5.4. Used Functions and Their Applications**

* **Mouse Event Handling (click\_and\_crop function):**

This function responds to mouse events and is used to select a region of interest (ROI) by clicking and dragging the mouse.

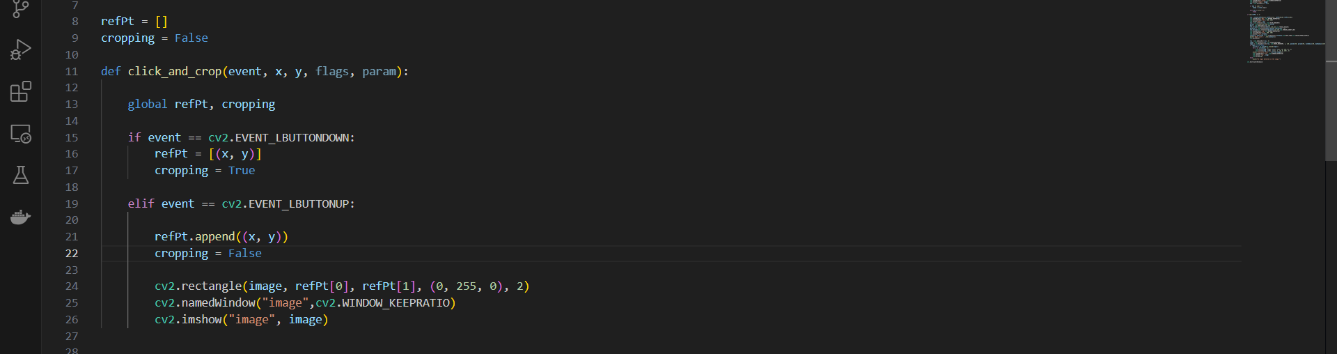


Figure 3.5.4.1

In this code segment, a function named **`click\_and\_crop`** is defined to handle mouse click and release events in OpenCV. This function is associated with a mouse callback and is responsible for enabling the user to select a region of interest (ROI) by clicking and dragging the mouse on an image. Here's an explanation of the key elements within this code segment:

- **`refPt`:** This is a global variable defined as an empty list. It will be used to store the coordinates of the two points that define the selected rectangular region (ROI).

**- `cropping`:** Another global variable, initially set to **`False`,** is used to indicate whether the user is currently in the process of selecting the ROI **(`True`)** or not **(`False`)**.

- **`click\_and\_crop**` Function: This function is designed to respond to mouse events, including both button down and button up events. It takes five arguments: `**event`, `x`, `y`, `flags`,** and `**param**`. Here's how it works:

- When the left mouse button is clicked (`**cv2.EVENT\_LBUTTONDOWN**`), the coordinates **`(x, y)**` of the mouse cursor are recorded in the `**refPt**` list, indicating the starting point of the ROI selection. Additionally, the `**cropping**` variable is set to `**True**`, indicating that the user has initiated the selection.

- When the left mouse button is released (`**cv2.EVENT\_LBUTTONUP**`), the coordinates **`(x, y)`** of the cursor are again recorded in the `**refPt**` list, this time indicating the ending point of the ROI selection. The `**cropping**` variable is then set to `**False**`, signifying that the selection process is complete.

- After the user has defined the ROI, a green rectangle is drawn on the original `**image**` to visually mark the selected region. The `**cv2.rectangle**` function is used for this purpose.

- Finally, the modified image with the rectangle is displayed in a window with the name "image" using `**cv2.imshow**`.

This function allows the user to interactively select an area of interest by clicking and dragging the mouse, and the selected region is visually highlighted by the green rectangle. The coordinates of the selected ROI are stored in the `refPt` list for further processing, such as cropping or analysis.

* **Image Loading and Cloning:**

The code loads an image (**'B5.jpg'**) and creates a clone for reference.



Figure 3.5.4.2

In this code segment, an image is loaded from a file named 'B5.jpg,' and a clone of the image is created for reference. Additionally, a named window is created, and a mouse callback function (`**click\_and\_crop**`) is set for that window. Here's an explanation of each part of this code segment:

- `**image = cv2.imread('B5.jpg')`:**

- This line of code reads an image from a file named 'B5.jpg' and stores it in the variable `**image`**. The image is typically loaded in the BGR (Blue, Green, Red) color format, which is the default format used by OpenCV.

- `**clone = image.copy()`:**

- Here, a clone of the loaded image (`**image**`) is created and stored in the variable `clone`. This clone is used as a reference image, allowing you to preserve the original image data while making changes to the displayed image.

- `**cv2.namedWindow("image", cv2.WINDOW\_KEEPRATIO)`:**

- This line creates a named window with the title "image." The named window is a graphical window where the loaded image will be displayed.

- `**cv2.WINDOW\_KEEPRATIO**` is a flag that specifies that the window should maintain the aspect ratio of the displayed image. This ensures that the image is displayed without distortion.

- `**cv2.setMouseCallback("image", click\_and\_crop)`:**

- This line sets a mouse callback function (`**click\_and\_crop**`) for the named window "image."

- The mouse callback function will be responsible for handling mouse events (such as clicks) that occur within this window. In this specific code, the `**click\_and\_crop**` function is called when mouse events occur within the "image" window.

Overall, this code segment prepares the environment for displaying the loaded image ('B5.jpg') in a named window, creating a clone for reference, and setting up the mouse callback function to enable the user to interact with the image by selecting a region of interest (ROI) using mouse events.

* **Display Loop:**

It continuously displays the image and waits for user input ('r' to reset or 'c' to continue).



Figure 3.5.4.3

In this code segment, a `while` loop is used to continuously display the image in an OpenCV window and handle user input. Here's an explanation of each part of this code segment:

- `**while True:`:**

- This initiates an infinite loop, which means the code inside the loop will run repeatedly until a certain condition is met.

- **`cv2.namedWindow('image', cv2.WINDOW\_KEEPRATIO)`:**

- Within each iteration of the loop, this line creates or updates a named window called "image" and specifies that it should maintain the aspect ratio of the displayed image (using the `**cv2.WINDOW\_KEEPRATIO`** flag). The named window is where the loaded image is displayed.

- `**cv2.imshow("image", image)`:**

- This line displays the current image (`**image**`) within the "image" window. This is where you see the image displayed.

- `**key = cv2.waitKey(1) & 0xFF`:**

- The `**cv2.waitKey(1)`** function waits for a keyboard event for up to 1 millisecond. The result is an integer representing the key pressed by the user.

- The `**& 0xFF**` operation is used to extract the least significant 8 bits of the integer (typically used for keyboard input) to ensure compatibility with different operating systems.

- `**if key == ord("r"):`:**

- This conditional statement checks if the key pressed by the user is the lowercase letter 'r'. If 'r' is pressed, the following code block is executed.

- `**image = clone.copy()`:**

- If the 'r' key is pressed, this line restores the current image (`**image**`) to its original state by copying the data from the reference image (`**clone`).** This effectively resets any changes made to the displayed image.

- **`elif key == ord("c"):`:**

- This conditional statement checks if the key pressed by the user is the lowercase letter 'c'. If 'c' is pressed, the following code block is executed.

- **`break`:**

- If the 'c' key is pressed, the **`break**` statement is encountered, which exits the `**while**` loop, terminating the program.

This code segment essentially creates a loop that continuously displays the image and waits for user input. The user can press 'r' to reset the image to its original state (based on the reference image) or 'c' to exit the program and close the image window. It provides an interactive way for the user to manipulate and analyze the image using the provided functionality.

* **ROI Selection:**

After selecting an ROI, the code extracts it and displays it in a separate window.



Figure 3.5.4.4

In this code segment, the program checks if the length of the `refPt` list is equal to 2. If this condition is met, it means that the user has successfully selected a region of interest (ROI) by clicking and dragging the mouse on the image. Here's an explanation of each part of this code segment:

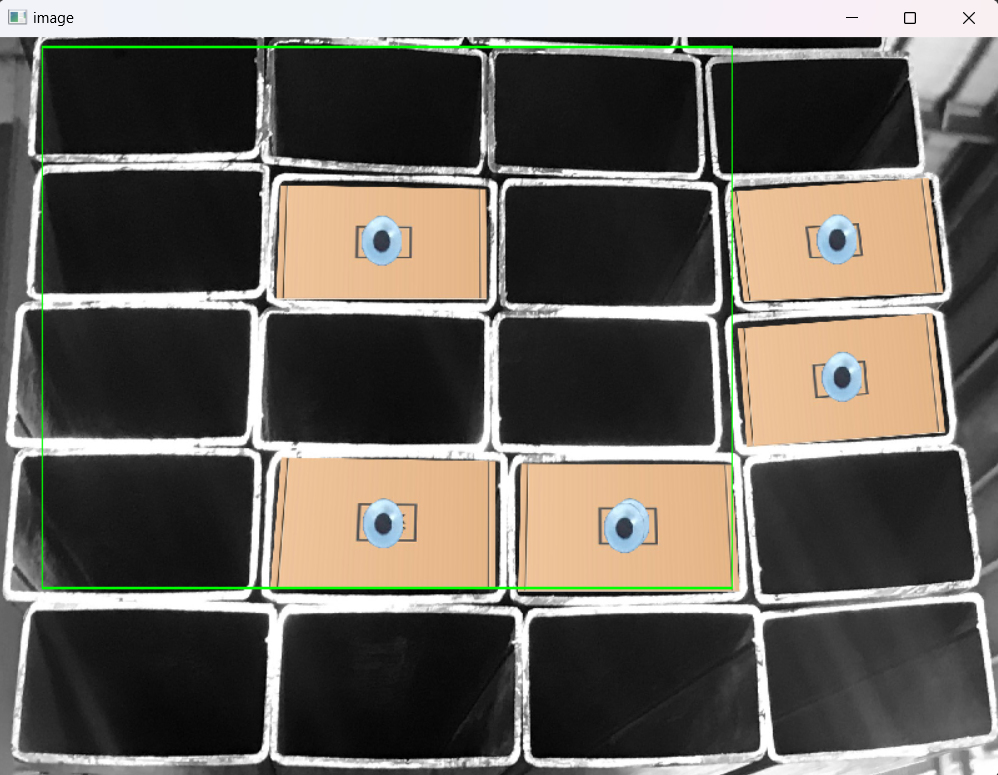


Figure 3.5.4.5

- `**if len(refPt) == 2:`:**

- This line checks if the length of the `**refPt**` list is equal to 2. The `**refPt**` list is used to store the coordinates of the two points that define the selected rectangular region (ROI).

- `**roi = clone[refPt[0][1]:refPt[1][1], refPt[0][0]:refPt[1][0]]`:**

- If the condition is met (i.e., two points have been selected), this line extracts the selected ROI from the `**clone**` image based on the coordinates of the two points stored in the `**refPt**` list.

- The `**refPt[0]`** represents the top-left corner of the ROI, and `**refPt[1]`** represents the bottom-right corner.

- This line effectively creates a new image (`**roi**`) that contains only the selected region from the original image.

- `**cv2.namedWindow('ROI', cv2.WINDOW\_KEEPRATIO)`:**

- After extracting the ROI, a new named window called "ROI" is created using `**cv2.namedWindow**`. This window is used to display the extracted region of interest.

- `**cv2.resizeWindow("ROI", 400, 300)`:**

- This line resizes the "ROI" window to have a width of 400 pixels and a height of 300 pixels. This is done to ensure that the window size is appropriate for displaying the ROI.

- `**cv2.imshow("ROI", roi)`:**

- Finally, the `cv2.imshow` function is used to display the extracted ROI (`**roi**`) in the "ROI" window. This allows the user to view the selected region separately from the original image.

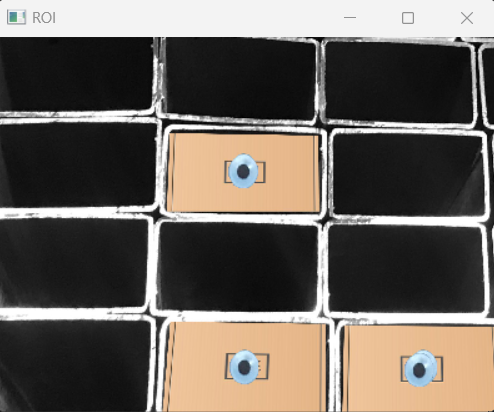


Figure 3.5.4.6

In summary, when the user selects a region of interest by clicking and dragging the mouse on the image, this code segment extracts that region, creates a new window ("ROI") to display it, and resizes the window for better visualization. This provides a closer look at the selected portion of the image, which can be useful for further analysis or processing.

* **Image Processing:**

The selected ROI is converted to grayscale and undergoes median blur, binary thresholding, dilation, and inverse binary thresholding for contour detection.



Figure 3.5.4.7

In this code segment, the selected region of interest (ROI) is further processed using various image processing techniques. Here's an explanation of each part of this code segment:

- **`roi = cv2.cvtColor(roi, cv2.COLOR\_BGR2GRAY)`:**

- This line converts the ROI (`**roi**`) from its original BGR color format to grayscale. Grayscale images have only one channel, representing the intensity or brightness of each pixel. Converting to grayscale simplifies subsequent image processing operations.

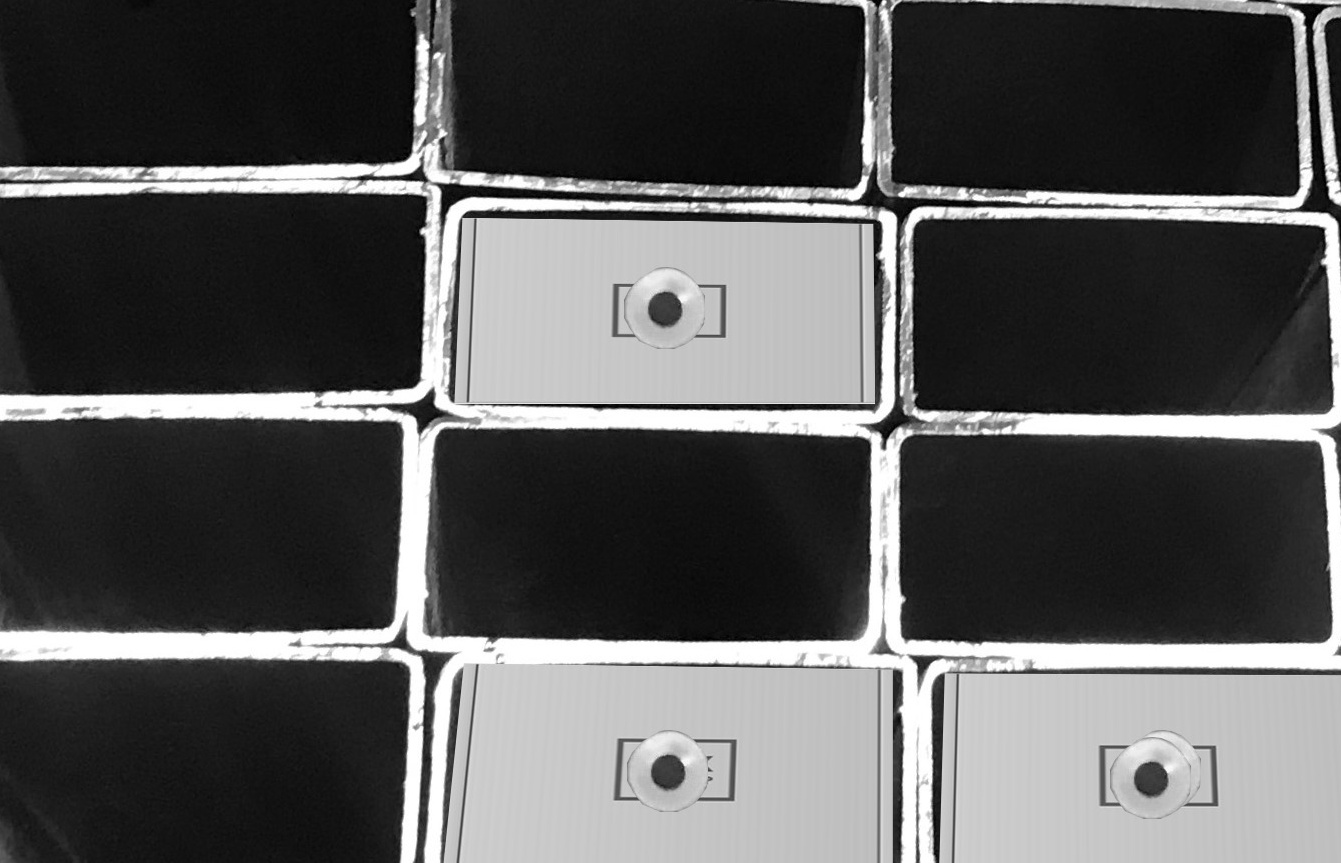


Figure 3.5.4.8

- `**kernel = np.ones((85, 110), np.uint8)`:**

- A kernel (structuring element) is created as a rectangular matrix of ones with dimensions 85x110 pixels. Kernels are used in operations like dilation and erosion to define the neighborhood of pixels for processing.

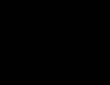


Figure 3.5.4.9

- **`blur = cv2.medianBlur(roi, 15)`:**

- Median blur is applied to the grayscale ROI (`**roi**`). Median blur is a non-linear filter that replaces each pixel's value with the median value of the neighboring pixels. In this case, it helps reduce noise and smooth the image. The parameter `**15**` represents the size of the kernel used for the median blur.

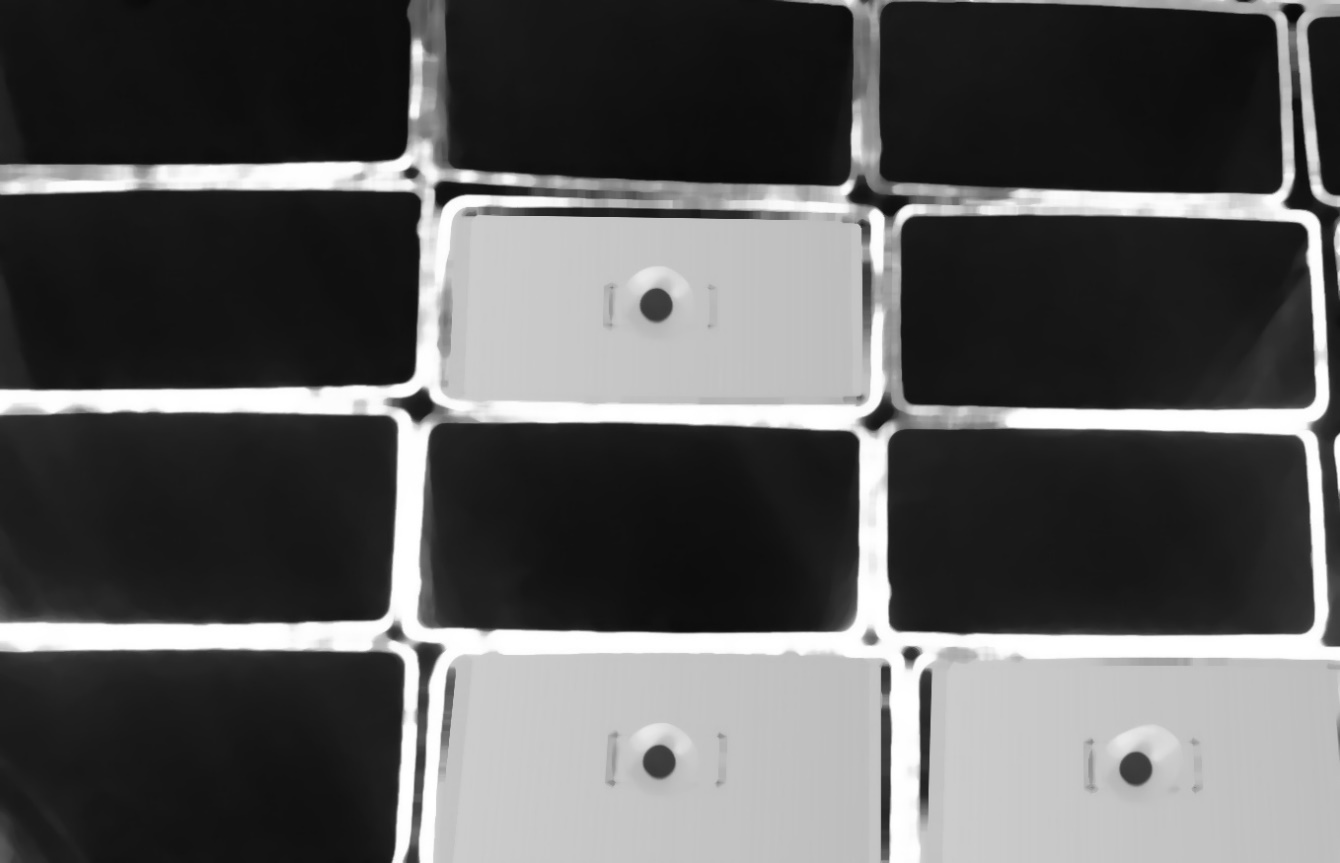


Figure 3.5.4.10

- `**ret, thresh = cv2.threshold(blur, 130, 255, cv2.THRESH\_BINARY)`:**

- This line applies binary thresholding to the blurred image (`**blur**`) to create a binary image (`**thresh**`). Binary thresholding converts the grayscale image into a black and white image where pixels are classified as either black (0) or white (255) based on their intensity values.

- Pixels with intensities greater than or equal to 130 are set to 255 (white), while those below 130 are set to 0 (black).

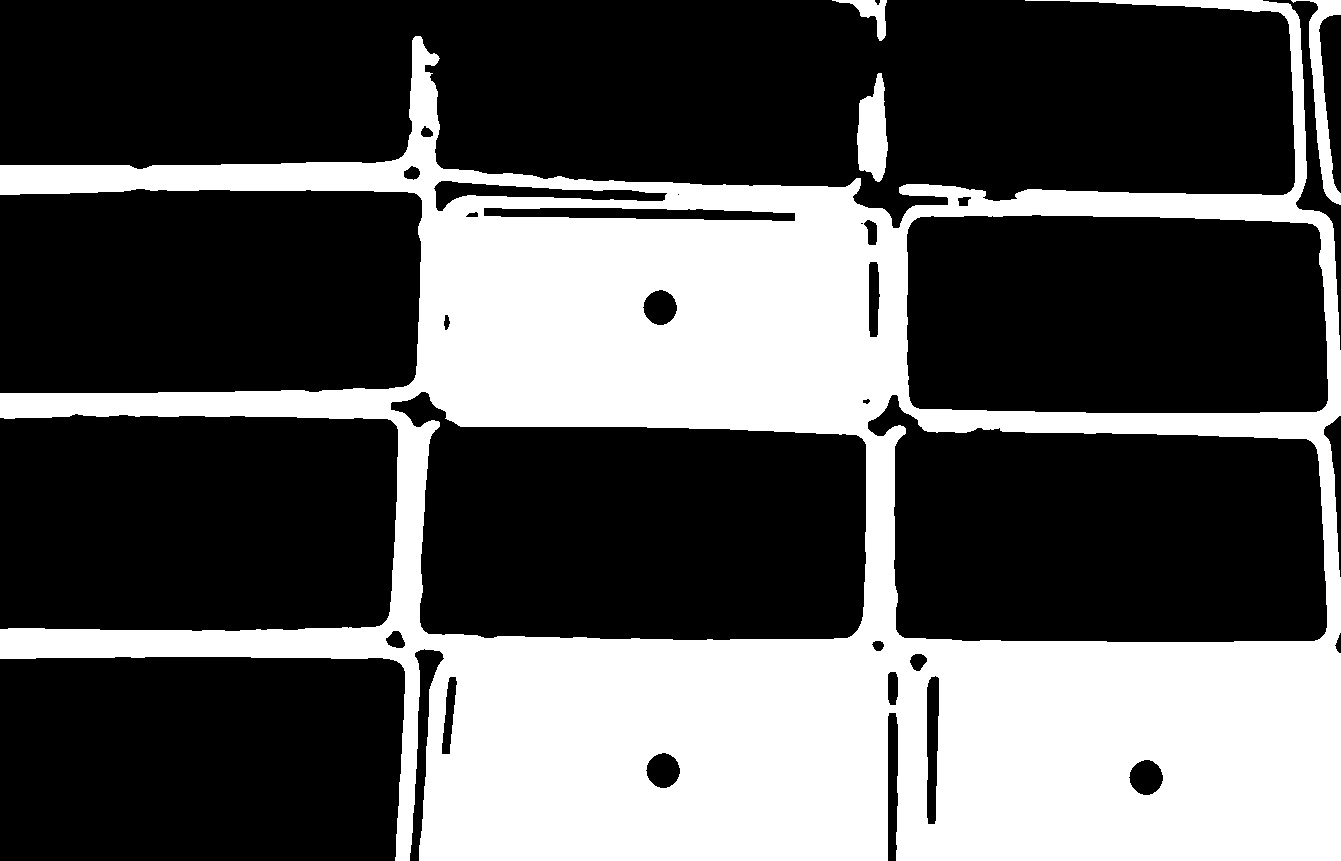


Figure 3.5.4.11

- `**dilation = cv2.dilate(thresh, kernel, iterations=1)`:**

- Dilation is performed on the binary thresholder image **(`thresh`)** using the defined kernel. Dilation is a morphological operation that expands the white regions (foreground) in a binary image by considering the neighborhood defined by the kernel.

- The **`iterations=1**` parameter specifies that dilation should be applied only once.

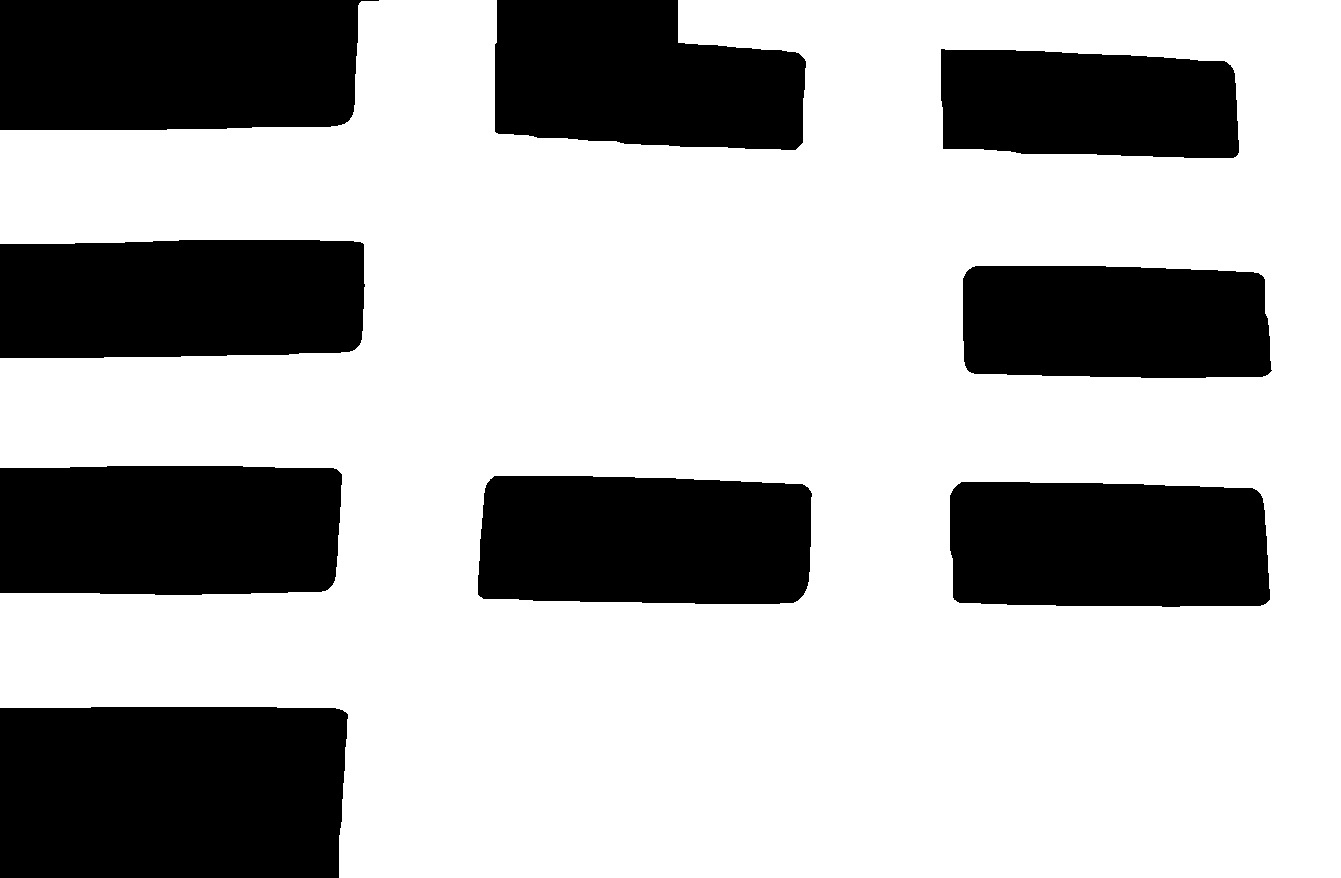


Figure 3.5.4.12

- `**ret, thresh1 = cv2.threshold(dilation, 127, 255, cv2.THRESH\_BINARY\_INV)`:**

- Another binary thresholding operation is applied to the dilated image (`dilation`) to create an inverse binary image (`**thresh1**`). In this case, pixels with intensities greater than or equal to 127 are set to 0 (black), and those below 127 are set to 255 (white).

- This operation effectively separates objects of interest (e.g., contours) from the background.

Overall, this code segment prepares the ROI for contour detection by converting it to grayscale, reducing noise, and creating a binary image with the objects of interest (foreground) highlighted. The final binary image (`**thresh1**`) is often used for contour analysis and object detection tasks.



Figure 3.5.4.13

* **Contour Detection:**

Contours in the processed binary image are detected, representing empty slots in the image.

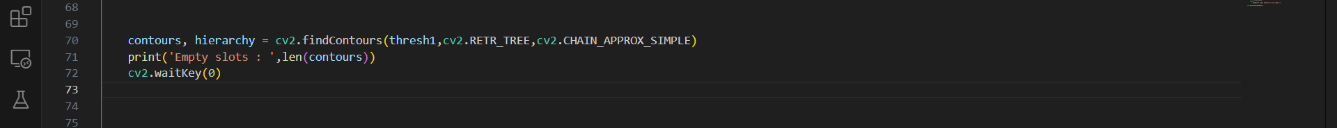


Figure 3.5.4.14

In this code segment, the binary image `thresh1` is processed to find and identify contours. Here's an explanation of each part of this code segment:

- **`contours, hierarchy = cv2.findContours(thresh1, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)`:**

- The `**cv2.findContours**` function is used to detect and extract contours from the binary image **`thresh1**`. Contours are the boundaries of objects or regions in an image.

- `**thresh1**` is the binary image from which contours will be extracted.

- `**cv2.RETR\_TREE**` is the retrieval mode for contours. It retrieves all of the contours and reconstructs a full hierarchy of nested contours. The hierarchy information can be useful when dealing with nested or hierarchical structures of objects.

- `**cv2.CHAIN\_APPROX\_SIMPLE**` is the contour approximation method. It compresses horizontal, diagonal, and vertical segments and leaves only their end points. This reduces memory consumption and simplifies the contour representation.

**- `print('Empty slots : ', len(contours))`:**

- After finding and extracting the contours, this line prints the number of detected contours to the console. In this specific context, the contours represent empty slots or regions in the image where objects are not present.

- The `**len(contours)**` function call returns the count of detected contours.

**- `cv2.waitKey(0)`:**

- This line waits for a keyboard event indefinitely (`**0**` as the argument). It effectively freezes the execution of the program until a key is pressed.

- This is often used to keep the OpenCV window displaying the processed image open until the user decides to close it manually.

Overall, this code segment performs contour detection on the binary image `**thresh1**`, counts the number of detected contours (which may correspond to empty slots in the image), and then waits for user interaction before proceeding. The contour information can be used for various purposes, such as identifying and analyzing objects or regions of interest in an image.

* **Logo Detection(Available Slots):**

The grayscale ROI is further processed to detect logos (circles) using the Hough Circle Transform.

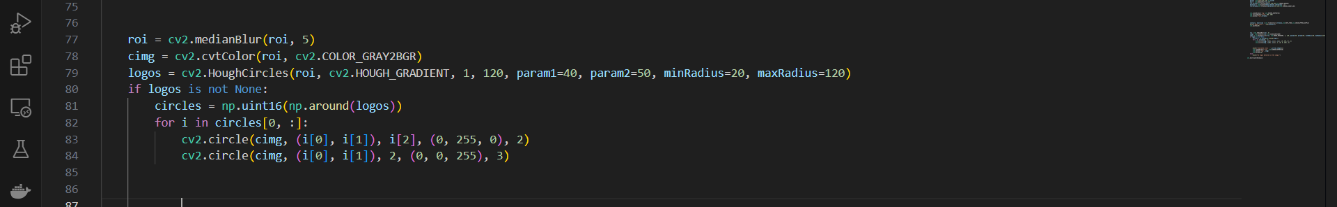


Figure 3.5.4.15

In this code segment, the selected region of interest (ROI), which has been processed and converted to grayscale, undergoes further processing to detect circles (logos) using the Hough Circle Transform. Here's an explanation of each part of this code segment:

- `**roi = cv2.medianBlur(roi, 5)**`:

- Median blur is applied once again to the grayscale ROI (`roi`) with a kernel size of 5. This additional blur operation is used to further reduce noise in the image before circle detection.

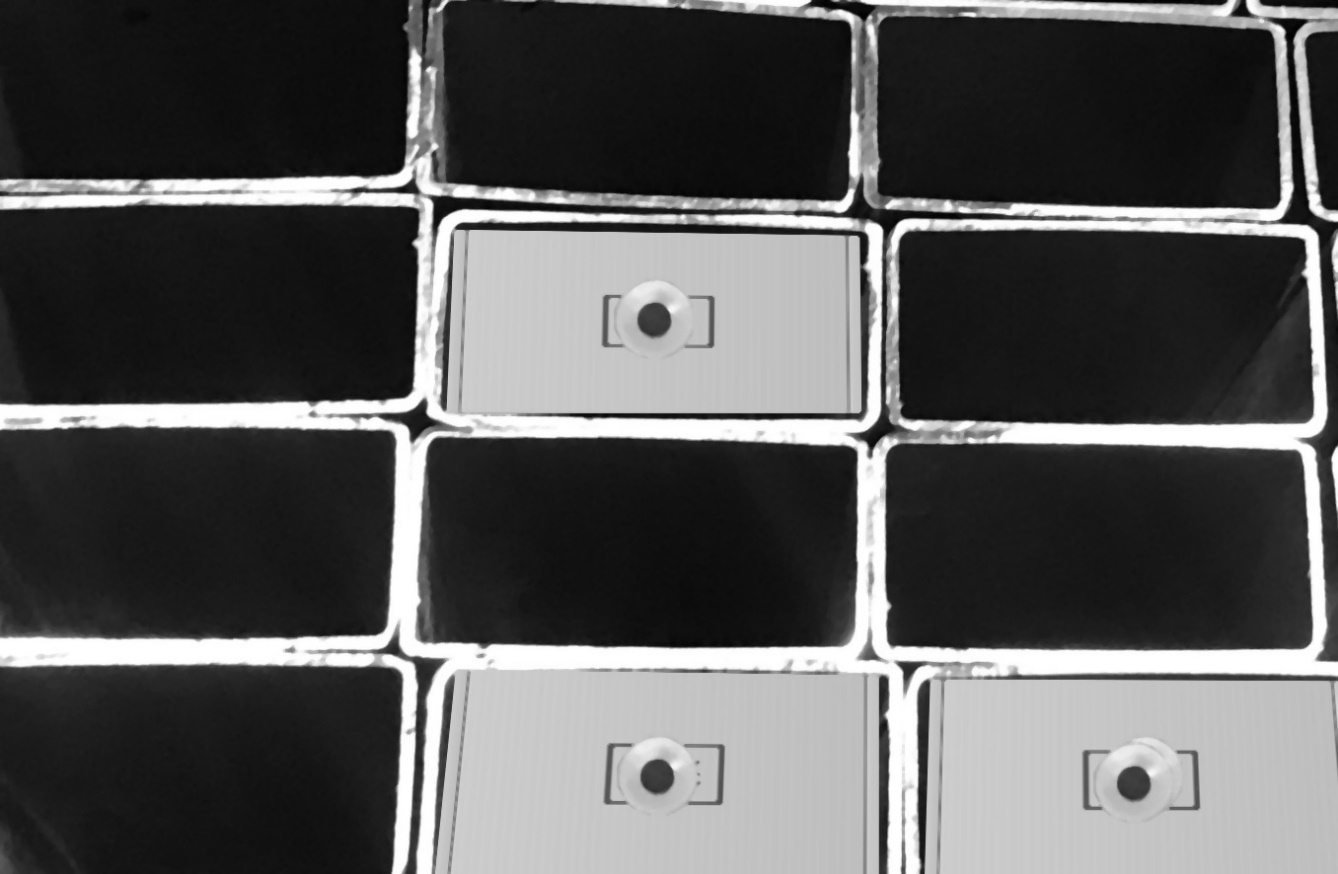


Figure 3.5.4.16

- `**cimg = cv2.cvtColor(roi, cv2.COLOR\_GRAY2BGR)`:**

- After blurring, the grayscale ROI (`**roi**`) is converted back to the BGR color format (`**cimg**`) because the Hough Circle Transform requires a color image as input.

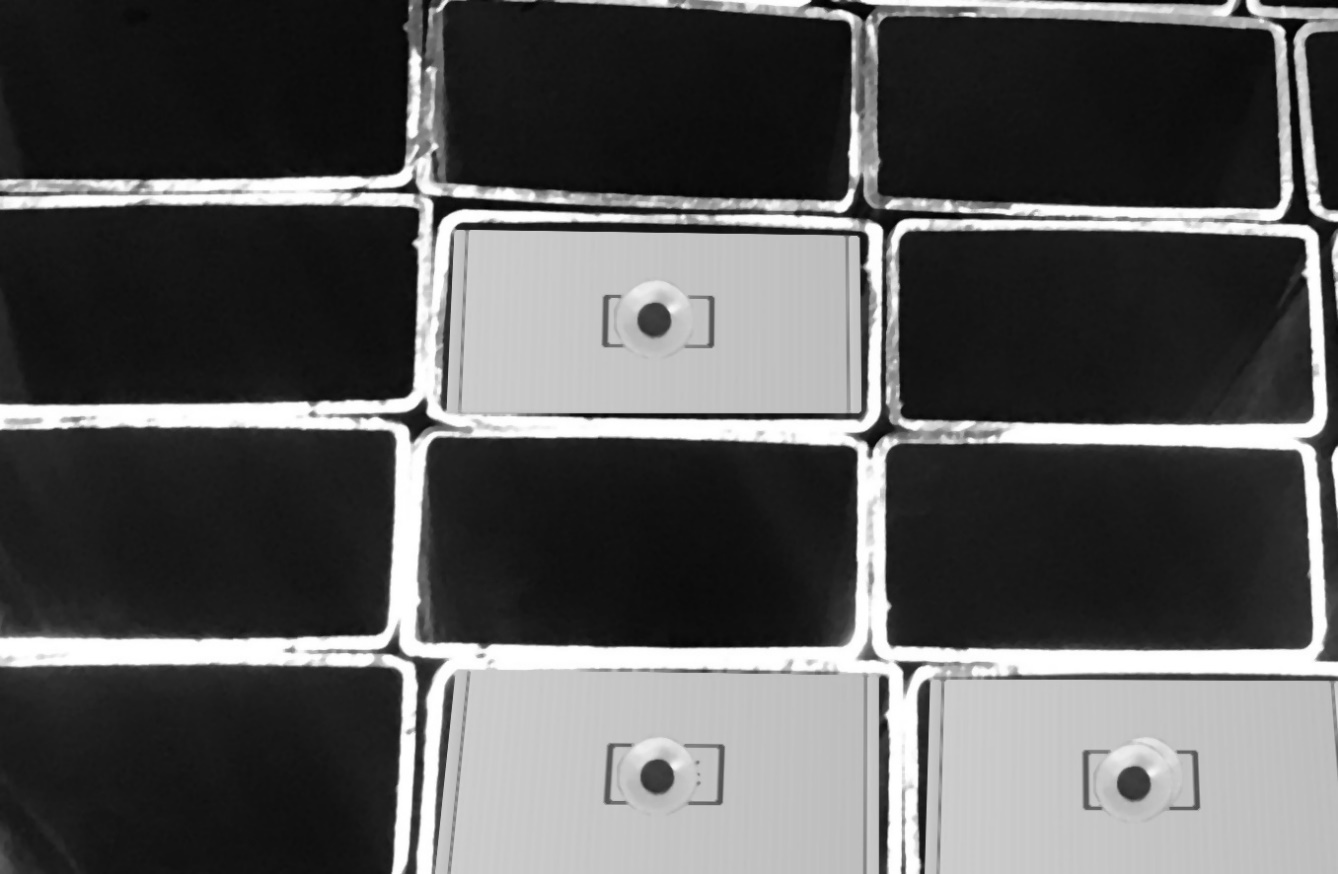


Figure 3.5.4.17

- **`logos = cv2.HoughCircles(roi, cv2.HOUGH\_GRADIENT, 1, 120, param1=40, param2=50, minRadius=20, maxRadius=120)`:**

- The Hough Circle Transform is applied to the grayscale ROI (`roi`) to detect circles (logos) within the image.

- `**cv2.HOUGH\_GRADIENT**` specifies the Hough detection method, which uses the gradient of the image.

- The parameters `**1**` and `**120**` specify the inverse ratio of the accumulator resolution and the minimum distance between the centers of detected circles, respectively.

- `**param1=40**` and `**param2=50**` are threshold parameters. The values control the sensitivity of circle detection; you may need to adjust these values based on your specific image.

- `**minRadius=20**` and **`maxRadius=120**` define the acceptable range of circle radii to be detected.

- `**if logos is not None:`:**

- This conditional statement checks if any circles (logos) were detected using the Hough Circle Transform.

- `**circles = np.uint16(np.around(logos))`:**

- If circles are detected, this line converts the coordinates of the detected circles to integers and rounds them to the nearest whole numbers for drawing purposes.

- Drawing Detected Circles:

- Inside the `**for**` loop, circles are iterated over, and the following drawing operations are performed:

- `**cv2.circle(cimg, (i[0], i[1]), i[2], (0, 255, 0), 2)`:** Draws a green circle around the detected logo. `i[0]` and `i[1]` are the coordinates of the circle's center, and `i[2]` is the radius.

- **`cv2.circle(cimg, (i[0], i[1]), 2, (0, 0, 255), 3)`:** Draws a red dot at the center of the detected logo for additional visualization.

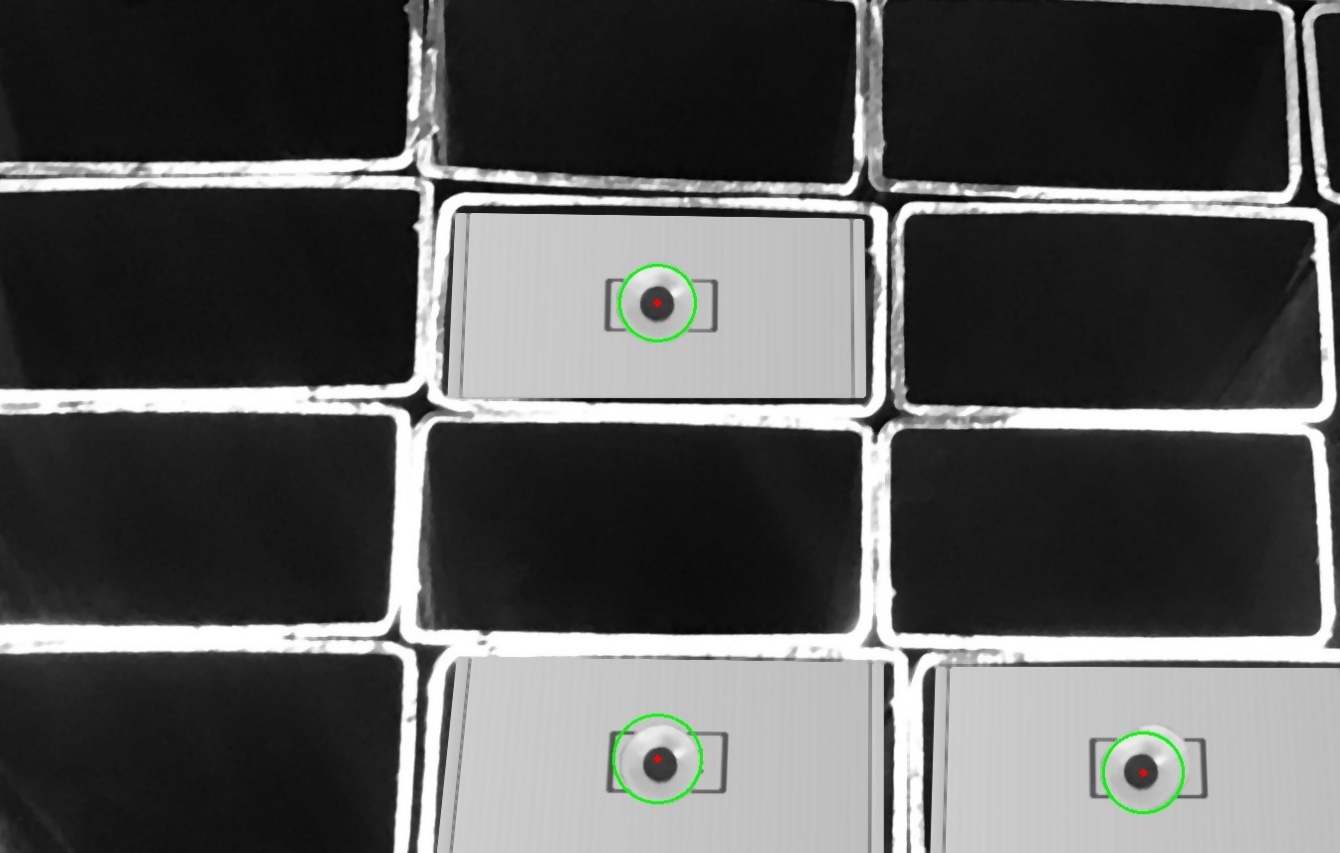


Figure 3.5.4.18

Overall, this code segment processes the ROI to detect circles (logos) using the Hough Circle Transform. Detected circles are then drawn on the image to highlight their locations. This can be useful for identifying and analyzing circular objects within the region of interest.

* **Display of Results:**

The number of empty slots and available slots (detected logos) are printed.

If logos are detected, circles are drawn around them, and the image is displayed.

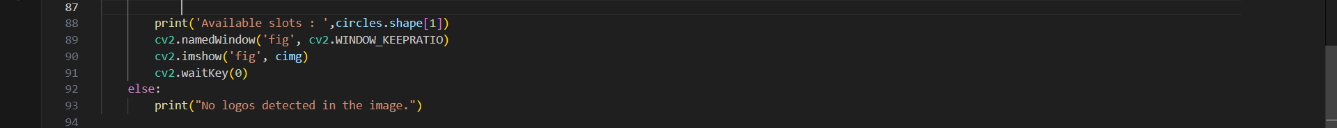


Figure 3.5.4.19

In this code segment, the program prints the number of available slots (detected circles) and displays the image with the detected circles drawn if any circles are found using the Hough Circle Transform. Here's an explanation of each part of this code segment:

- `**print('Available slots : ', circles.shape[1])`:**

- If the Hough Circle Transform successfully detected circles in the ROI, this line prints the number of available slots to the console.



Figure 3.5.4.20

- `**circles.shape[1]`** returns the number of columns in the `**circles**` array, which corresponds to the number of detected circles. Each column in the array represents a detected circle, and the `**shape[1]`** attribute counts these columns.

**- `cv2.namedWindow('fig', cv2.WINDOW\_KEEPRATIO)`:**

- A new named window named "fig" is created to display the image with the detected circles. The `**cv2.WINDOW\_KEEPRATIO**` flag ensures that the window maintains the aspect ratio of the displayed image.

- `**cv2.imshow('fig', cimg)`:**

- The image (`**cimg**`) with the detected circles drawn on it is displayed in the "fig" window using the `**cv2.imshow**` function. This allows the user to see the original image with the highlighted circles.

- **`cv2.waitKey(0)`:**

- This line waits indefinitely (`**0**` as the argument) for a keyboard event. It effectively keeps the "fig" window open until the user manually closes it by pressing a key.

- This is a common way to display images in OpenCV, as it allows the user to view the results and interact with the displayed image.

**- `else`:**

- If no circles (logos) are detected in the ROI using the Hough Circle Transform, the program prints the message "No logos detected in the image."

Overall, this code segment informs the user about the number of available slots (detected circles) and displays the image with the detected circles drawn if they were found. It provides visual feedback on the presence or absence of circular objects within the selected region of interest.

* **Window Cleanup:**

All OpenCV windows are closed at the end.



Figure 3.5.4.21

The `**cv2.destroyAllWindows()`** function is used to close all open windows that were created using the OpenCV library. Here's an explanation of its purpose:

- `**cv2.destroyAllWindows()`:**

- This function is called to close all the graphical windows that were opened by the OpenCV library during the course of the program's execution. These windows may have been used to display images, videos, or other graphical content.

- Purpose:

- When working with OpenCV, you often create named windows to display images, videos, or various processing results. These windows remain open until they are explicitly closed.

- The `**cv2.destroyAllWindows()`** function is typically called at the end of a program to ensure that all open windows are closed before the program exits. It helps prevent leaving unwanted windows open, which can clutter the user's desktop.

- Usage:

- You can call **`cv2.destroyAllWindows()`** without any arguments. It will automatically close all open windows, regardless of their names.

**3.5.5. Display The Final Output**

Available slots

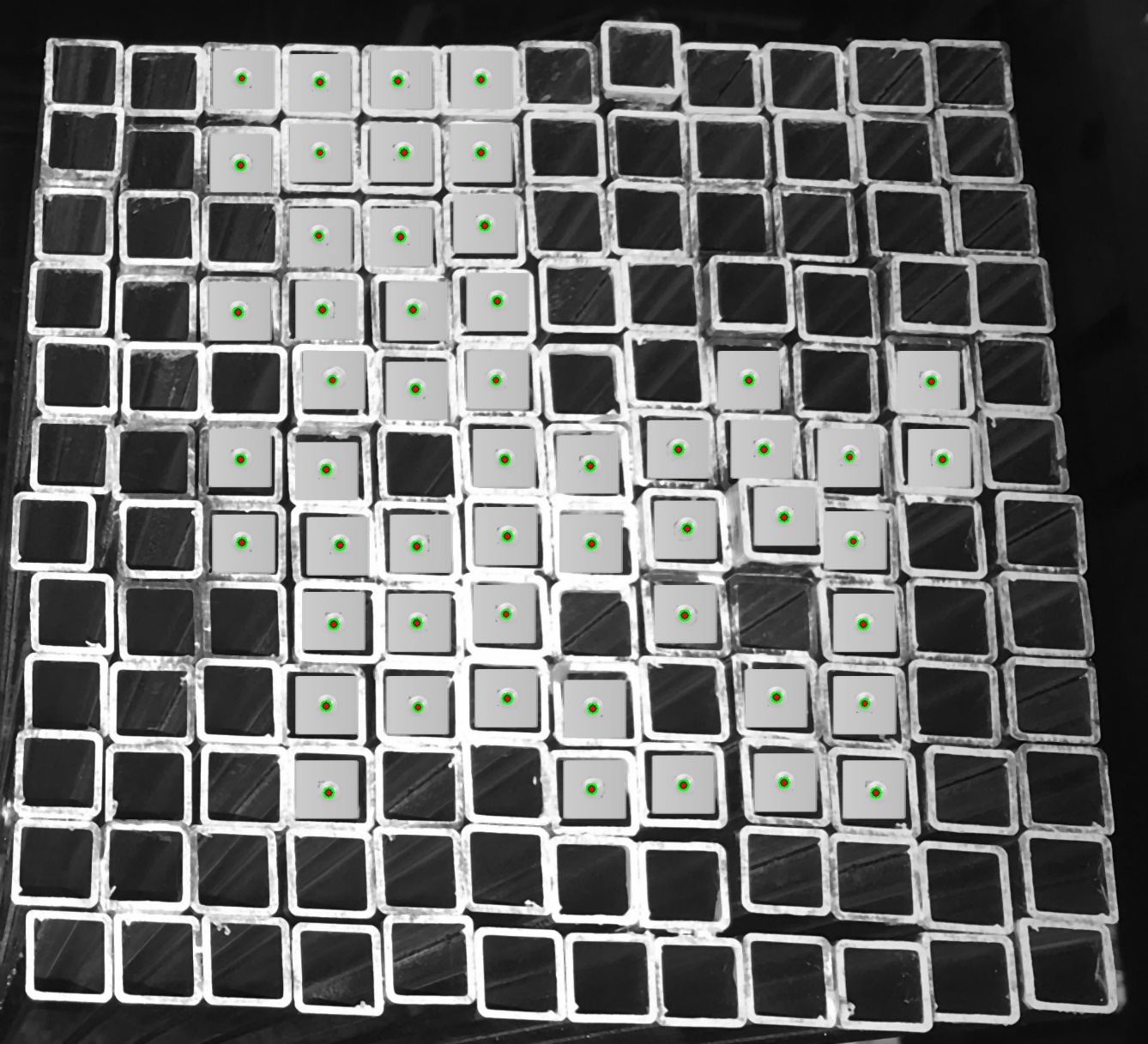


Figure 3.5.5.1

Empty slots

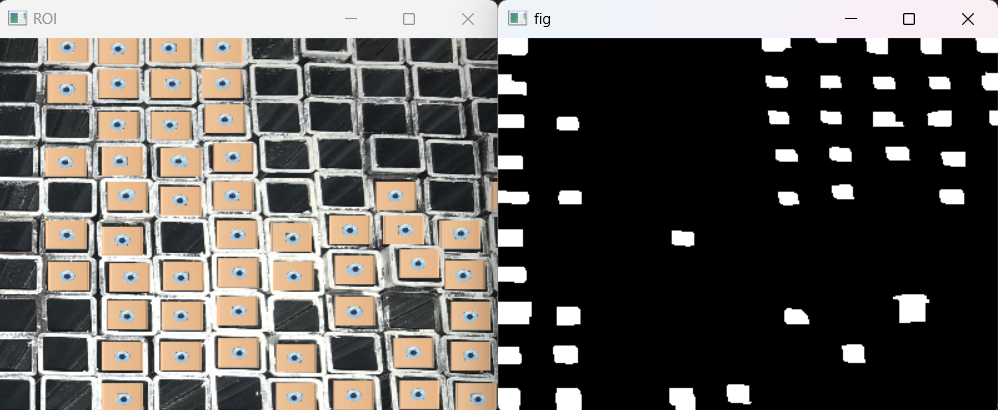


Figure 3.5.5.2 Figure 3.5.5.3

**Display The Final Result**



Figure 3.5.5.4

**CHAPTER 04**

# RESULTS AND DISCUSSION

# 

## **Major Resulting Steps**

**4.1.1 Rectangle Shape Shelves**

**Available Slots**

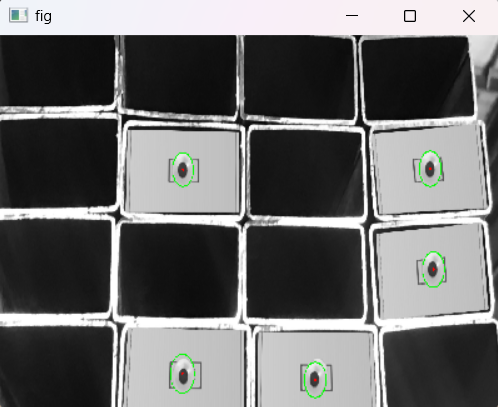
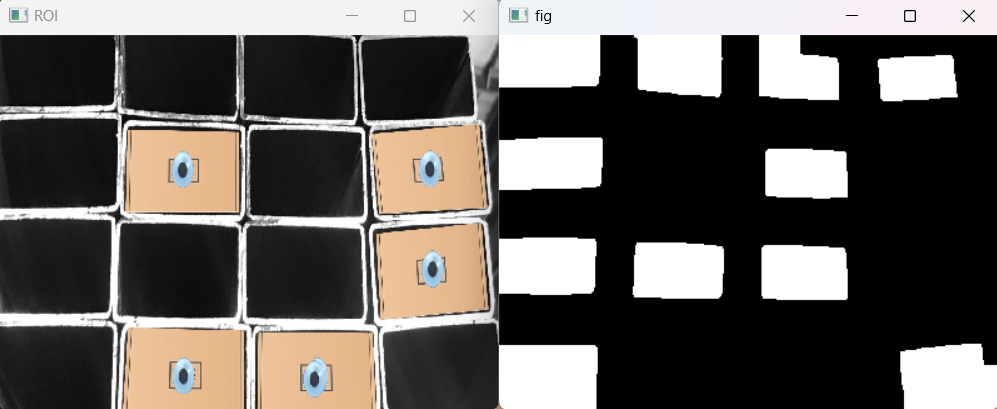
****

Figure 4.1.1.1

**Empty Slots**

**** Figure 4.1.1.2 Figure 4.1.1.3

**Result**

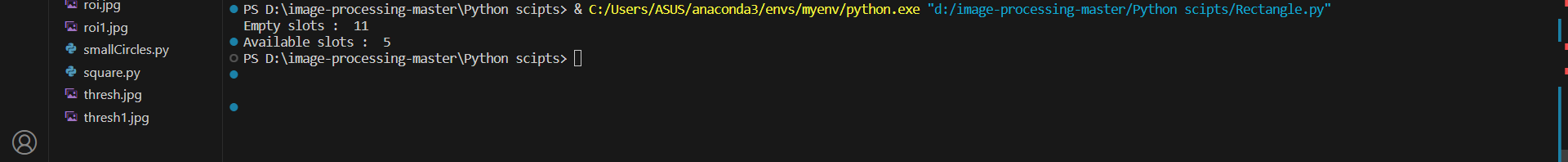
****

Figure 4.1.1.4

**4.1.2 Squar Shape Shelves**

**Available Slots**

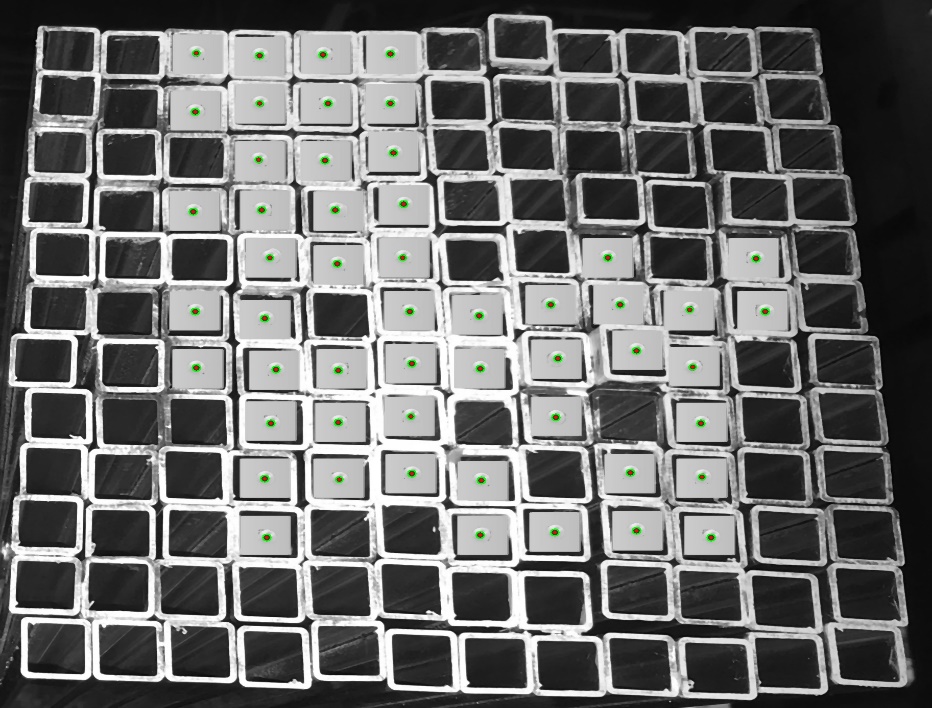
****

Figure 4.1.2.1

**Empty slots**

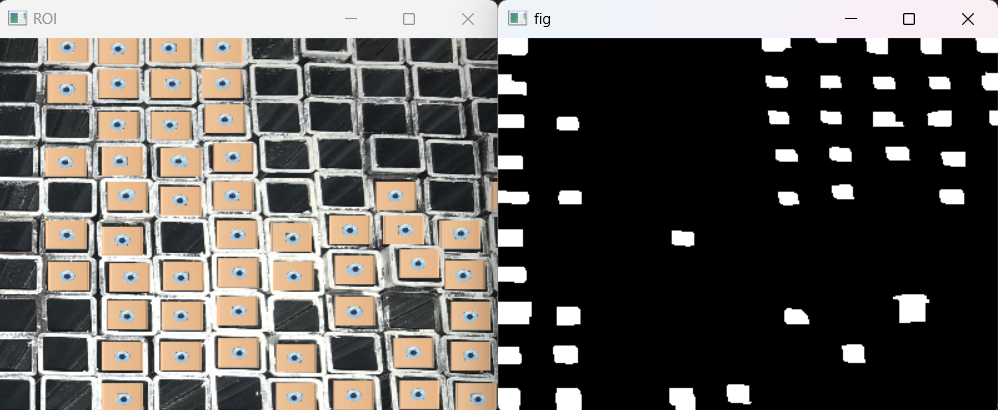
****

Figure 4.1.2.2 Figure 4.1.2.3

**Result**

****

Figure 4.1.2.4

* 1. **Some Tested Image Results**

****

Figure 4.2.1

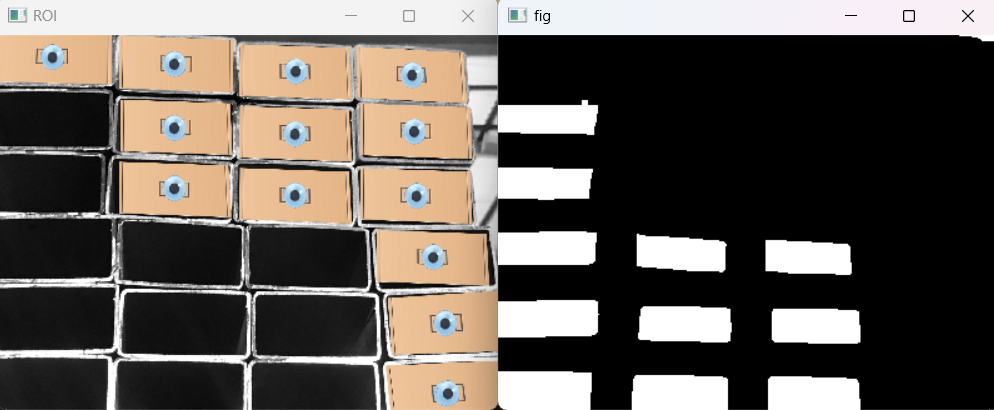
****

Figure 4.2.2 Figure 4.2.3

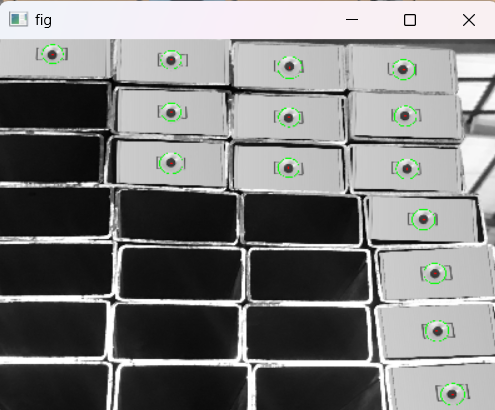
****

Figure 4.2.4

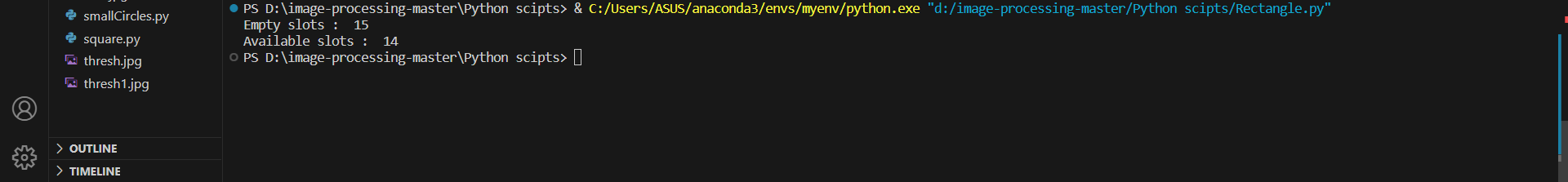
****

Figure 4.2.5

* 1. **Discussion**

Figure 4.3.1 Figure 4.3.2

All the images of dataset are created by myself using a photoshop. More than 50 images were created using different reference objects, different background and different shelves. So far from all those attempts, I understood that the position of reference object and the camera angle is important to get a better output. One of the example for camera angle failure is shown in the Figure 4.3.1. The measurement is deviated from the original

image because of the wrong camera angle. Therefore, as a first step camera should be positioned correctly. In companies the fixed camera setup can be done as shown in the figure 4.3.2.

**CHAPTER 05**

# CONCLUSIONS

## **Advantages of the Implemented system**

The implemented Warehouse Stocks Monitoring System based on image processing offers several advantages, which can benefit businesses and organizations involved in warehouse management and inventory tracking. Here are some of the key advantages of the system:

1. Efficient Inventory Management: The system provides real-time monitoring and tracking of warehouse stocks, allowing businesses to efficiently manage their inventory. This can lead to reduced stockouts, overstock situations, and associated costs.

2. Accuracy and Precision: Image processing techniques ensure accurate and precise identification of objects or products within the warehouse. This reduces the risk of errors in inventory counts.

3. Automation: The system automates the process of stock monitoring and tracking, reducing the need for manual labor and human intervention. This can lead to cost savings and increased operational efficiency.

4. Quick Detection of Empty Slots: The system can quickly detect and report empty storage slots or areas within the warehouse. This information can be used to optimize storage space and improve warehouse layout.

5. Real-Time Updates: By continuously monitoring the warehouse, the system provides real-time updates on stock levels and the availability of products. This enables businesses to make timely decisions regarding restocking and order fulfillment.

6. Reduced Theft and Loss: The system can help identify discrepancies between recorded stock levels and actual stock levels, which can be indicative of theft or loss. This can act as a deterrent and aid in the prevention of such incidents.

7. Data Analytics: The collected data can be analyzed to gain insights into warehouse operations and trends. Businesses can use this information for demand forecasting, optimizing stock levels, and making informed decisions.

8. Improved Customer Service: With better inventory management and tracking, businesses can ensure that products are readily available to meet customer demands. This can lead to improved customer satisfaction and retention.

9. Cost Savings: By optimizing inventory levels and reducing manual labor requirements, the system can lead to cost savings in warehouse operations.

10. Scalability: The system can be scaled to accommodate larger warehouses or multiple warehouse locations, making it suitable for businesses of varying sizes.

11. Integration: It can be integrated with other warehouse management systems (WMS), enterprise resource planning (ERP) software, or inventory management software for seamless data sharing and management.

12. Enhanced Security: The system can enhance security by monitoring access to the warehouse and alerting personnel to unauthorized entry.

13. Compliance and Reporting: It can generate reports and documentation required for compliance with regulations and auditing purposes.

* 1. **Issues**

While the implemented Warehouse Stocks Monitoring System using image processing offers several advantages, it may also face certain issues and challenges that need to be considered. Here are some of the potential issues:

1. Accuracy and Reliability: The accuracy of object detection and counting in the warehouse relies on the effectiveness of image processing algorithms. Factors like lighting conditions, object placement, and image quality can impact accuracy.

2. Complexity of Implementation: Developing and deploying an image processing system can be complex and may require specialized skills in computer vision and machine learning. It may also involve significant initial setup and configuration.

3. Maintenance and Calibration: Regular maintenance and calibration of cameras and sensors are essential to ensure consistent performance. Dust, changes in lighting, and camera misalignment can affect system accuracy.

4. Scalability: Scaling the system to larger warehouses or multiple locations can be challenging. It may require additional hardware and computational resources.

5. Costs: Initial setup costs for cameras, sensors, and computing equipment can be substantial. Additionally, ongoing maintenance and software updates can incur ongoing expenses.

6. Privacy and Security: Warehouse surveillance systems may raise privacy concerns among employees and visitors. It's important to implement privacy policies and security measures to protect sensitive data and ensure compliance with privacy regulations.

7. False Positives and Negatives: Image processing algorithms may produce false positives (identifying an object that isn't there) or false negatives (failing to identify an existing object). These errors can impact inventory accuracy.

8. Integration Challenges: Integrating the system with existing warehouse management or enterprise software can be complex. Compatibility issues and data synchronization may arise.

9. Limited Object Recognition: The system's ability to recognize and track specific types of objects or products may be limited. It may not be suitable for all types of inventory.

10. Environmental Factors: Environmental conditions such as extreme temperatures, humidity, or exposure to dust and debris can affect the performance and durability of cameras and sensors.

11. Data Storage and Processing: Storing and processing large amounts of image data can strain computational resources and require substantial storage capacity.

12. Training and User Adoption: Personnel may need training to operate and maintain the system effectively. Resistance to change among employees can be a barrier to successful adoption.

13. Regulatory Compliance: Depending on the region and industry, there may be regulatory requirements related to surveillance and data handling that need to be addressed.

* 1. **Conclusion**

the implemented Warehouse Stocks Monitoring System using image processing presents a promising solution for businesses and organizations engaged in warehouse management and inventory tracking. This system leverages image processing techniques to offer several benefits, including efficient inventory management, accuracy in stock tracking, automation of monitoring processes, and real-time updates on stock levels.

By quickly detecting empty slots within the warehouse and automating the identification of available and unavailable stock, the system can enhance operational efficiency and reduce costs. It provides a mechanism for precise inventory management, ensuring that businesses can optimize their storage space and minimize stockouts or overstock situations.

Moreover, the system offers data analytics capabilities, allowing organizations to gain valuable insights into their warehouse operations, track trends, and make data-driven decisions for demand forecasting and inventory optimization. This can lead to improved customer service, cost savings, and enhanced competitiveness in the market.

However, it's essential to acknowledge and address potential challenges and issues, such as accuracy limitations, complexity of implementation, maintenance requirements, scalability concerns, and privacy considerations. A well-planned strategy for system deployment, regular maintenance, and adherence to privacy regulations can mitigate these challenges and ensure the system's long-term success.

# REFERENCES

The image processing pipeline

https://subscription.packtpub.com/book/big-data-and-business-intelligence/9781789343731

/1/ch01lvl1sec12/the-image-processing-pipeline

Mordvintsev, A. and Abid, K., 2014. Opencv-python tutorials documentation.

https://www.kaggle.com/blobs/download/forum-message-attachment-files/16192/OpenCVPython%20Tutorials-2017.pdf. pdf.

NumPy

https://www.w3schools.com/python/numpy/numpy\_intro.asp

Matplotlib Tutorial

https://www.w3schools.com/python/matplotlib\_intro.asp

Matplotlib Pyplot

https://www.w3schools.com/python/matplotlib\_pyplot.asp

OpenCV center of contour

https://pyimagesearch.com/2016/02/01/opencv-center-of-contour/#:~:text=%23%20find%2

0contours%20in%20the%20thresholded%20image%20cnts,each%20of%20the%20white%

20blobs%20on%20the%20image.

Adobe Photoshop

https://www.adobe.com