VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

TRAN THI THE NHAN – 523K0047

MIDTERM ESSAY

Introduction to Digital Image Processing

HO CHI MINH CITY, 2025

VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

TRAN THI THE NHAN – 523K0047

MIDTERM ESSAY

Introduction to Digital Image Processing

Advised by

Dr. Trinh Hung Cuong

HO CHI MINH CITY, 2025

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to Mr. Trinh Hung Cuong, our instructor and mentor, for his valuable guidance and support throughout the mid-term report of our project on Building a book management and ordering system on the MERN stack platform. He has been very helpful and patient in providing us with constructive feedback and suggestions to improve our work. He has also encouraged us to explore new technologies and techniques to enhance our system's functionality and performance. We have learned a lot from his expertise and experience in web development and software engineering. We are honored and privileged to have him as our teacher and supervisor.

Ho Chi Minh city, 2rd November 2025
Author
(Signature and full name)

Nhan

Tran Thi The Nhan

DECLARATION OF AUTHORSHIP

We hereby declare that this is our own project and is guided by Mr. Trinh Hung Cuong; The content research and results contained herein are central and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the main author from different sources, which are clearly stated in the reference section.

In addition, the project also uses some comments, assessments as well as data of other authors, other organizations with citations and annotated sources.

If something wrong happens, We'll take full responsibility for the content of my project. Ton Duc Thang University is not related to the infringing rights, the copyrights that We give during the implementation process (if any).

Ho Chi Minh city, 2rd November 2025

Author

(Signature and full name)

Nhan

Tran Thi The Nhan

TABLE OF CONTENTS

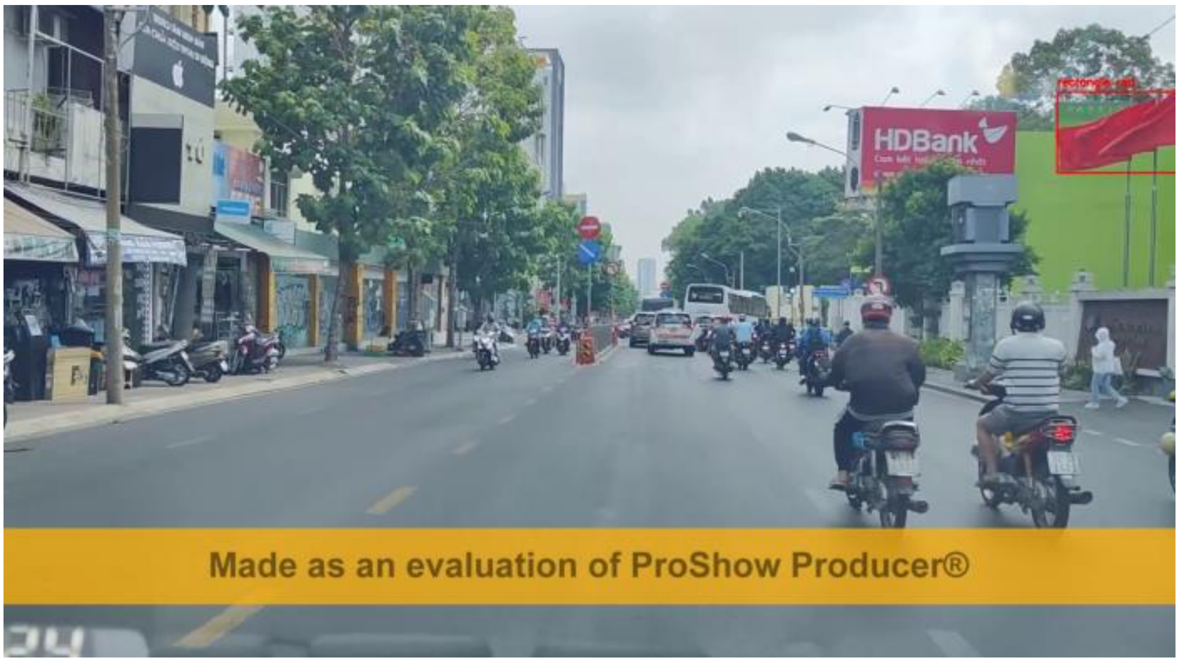
Chapter 1: Methodology of Solving Tasks 2
1.1 Task 1:.. 2
1.2 Task 2:.. 3
Chapter 2: Task results 14
2.1 Task 1:.. 14
2.2 Task 2:.. 16

Chapter 1: Methodology of Solving Tasks

1.1 Task 1:

1.1.1 Main Programming Steps

1. The traffic sign detection system follows a multi-stage pipeline.
2. Frame Preprocessing - Enhance image quality and convert to suitable color space
3. Color Segmentation - Isolate red and blue regions (common traffic sign colors)
4. Shape Detection - Analyze contours to identify geometric shapes. About ‘analyze_contour_curvature’ function aim to analyze contour curvature to detect wavy edges, return avg_curvature < 150. Then i call the function and assign it to has_straight_edges variable to check if the region seem as a rectangle is an actual rectangle or not. For example, the flag may be catch as a rectangle traffic sign because it 'nearly' in rectangle but it has wavy edges and more deformed . Returns True if edges are relatively straight, False if wavy.



Before apply has_straight_edges condition

After apply has_straight_edges condition

5. Shape Validation - Verify detected shapes meet traffic sign criteria
6. Border Detection - Special handling for signs with red borders
7. Final Verification - Combine color and shape information for accurate detection
8. Video Processing - Apply detection to all frames and generate output

1.1.2 Detailed Code Explanation

1. Import Libraries and Initial Setup

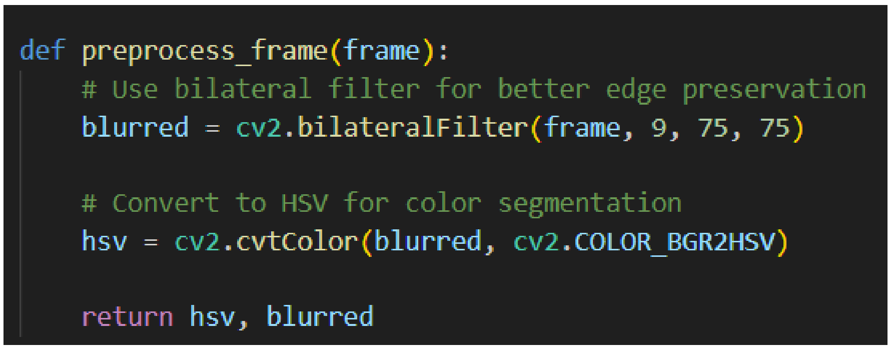
- cv2: OpenCV for computer vision operations
- numpy: Numerical computing for array operations
- math: Mathematical functions for geometric calculations
- matplotlib: Visualization and plotting
- typing: Type hints for better code documentation
- dataclasses: For creating structured data objects

2. Noise Removal Function

- Purpose: Remove small noisy regions from binary images
- Process:

- Threshold the image to create binary mask
- Use connected components analysis to identify separate regions
- Calculate area of each connected component
- Keep only components larger than min_size
- Create output mask with filtered components

3. Frame Preprocessing



```python
def preprocess_frame(frame):
    # Use bilateral filter for better edge preservation
    blurred = cv2.bilateralFilter(frame, 9, 75, 75)

    # Convert to HSV for color segmentation
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

    return hsv, blurred
```

- Bilateral Filter: Reduces noise while preserving edges
- Parameters: diameter=9, sigmaColor=75, sigmaSpace=75
- HSV Conversion: Converts from BGR to HSV color space
- HSV is better for color segmentation because it separates color (Hue) from brightness (Value)

4. Color Segmentation

- Color Ranges: Define HSV ranges for red and blue colors
- Red has two ranges due to its position at the extremes of HSV spectrum
- Blue has one primary range
- Morphological Operations:
- Opening: Removes small noise (erosion followed by dilation)
- Closing: Fills small holes (dilation followed by erosion)
- Median Blur: Further reduces noise while preserving edges

5. Shape Detection and Analysis

- Morphological Processing: Clean up the mask for better contour detection
- Contour Finding: Identify all closed shapes in the processed mask

6. Geometric Property Analysis

- Circularity: Measures how close a shape is to a perfect circle (1.0 = perfect circle)
- Solidity: Ratio of contour area to convex hull area (measures convexity)
- Aspect Ratio: Width to height ratio of bounding rectangle
- Vertices Count: Number of corners in polygon approximation

7. Shape Classification Logic

- Multi-level Approximation: Uses different epsilon values (0.01, 0.02, 0.03) for polygon approximation to handle scale variations
- Confidence Scoring: Each shape gets a confidence score based on geometric properties
- Weighted Criteria: Different properties contribute differently to shape classification

8. Red Border Detection

I create a handling for traffic signs have borders:

- Color Ratio Analysis: Checks if red pixels are moderately distributed (not filling entire area)
- Inner Region Analysis: For circular borders, verifies inner region has low red content
- Border-specific Criteria: More lenient aspect ratio and area requirements

9. Main Detection Pipeline

Two-Pronged Approach:

- Standard Detection: Process contours from color segmentation

- Border Detection: Special handling for signs with red borders
- Overlap Checking: Prevents duplicate detections

10. Validation and Verification

Validation Criteria:

- Color-Shape Validation: Ensures detected combinations match real traffic signs
- Geometric Validation: Uses compactness and area ratio to verify shape regularity
- Size Constraints: Minimum and maximum area limits to filter noise and large objects

11. Video Processing

- Video Processing Steps:
- Input Setup: Open video file and read properties
- Output Setup: Create VideoWriter with same properties as input
- Frame Processing: Apply detection to each frame
- Annotation: Draw green rectangles around detected signs
- Student ID: Add identification text to each frame
- Progress Tracking: Print processing status every 30 frames

Key Technical Attempt and Innovations

- Multi-stage color segmentation uses multiple HSV ranges and morphological operations for robust color detection
- Geometric Property Analysis: Combines multiple shape descriptors for accurate classification
- I tried enhance contrast of each frame use Adaptive Histogram Equalization (CLAHE), cv2.equalizeHist but it **not work well**.
- Set up ‘area’ limitation is also improtant to reduce small wrong detection that not in standard traffic signs size.

- Multi-level Polygon Approximation: Handles shape detection at different scales
- Border-specific Detection: Specialized algorithm for hollow traffic signs
- Comprehensive Validation: Multiple verification steps to reduce false positives

1.2 Task 2:

1.2.1 Summary of the main programming steps

· **Load image** (expects a grayscale image file).

· **Remove table/grid lines** (horizontal/vertical) so they don’t break digit shapes.

· **Enhance contrast and denoise** (Gaussian blur + CLAHE) and produce a robust binary map using *adaptive thresholding* combined with *Otsu*.

· **Morphological cleanup**: I removed tiny noise components, erode/dilate in a carefully ordered sequence to join digit strokes and remove spurs/noise.

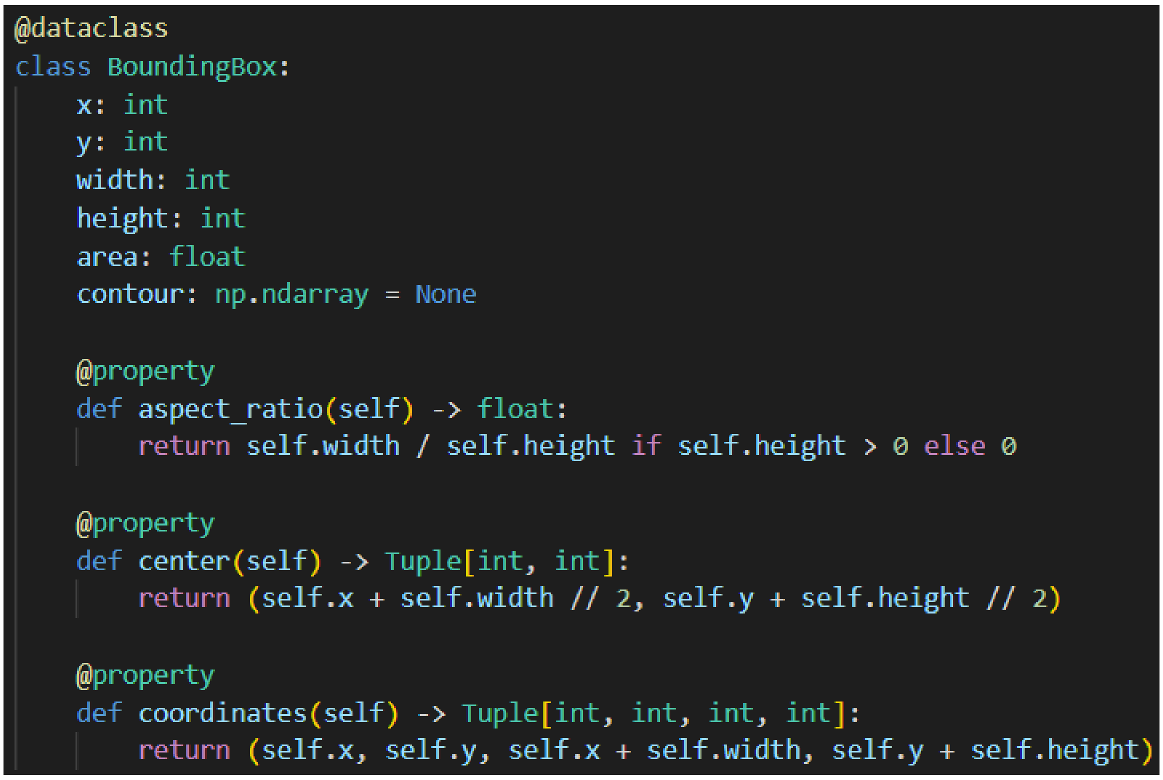
· **Connected-component / contour analysis**: extract contours, compute bounding rectangles and geometric filters (area, aspect ratio, solidity, extent, border checks) to keep likely digit regions.

· **Create a presentation frame**: draw bounding boxes on original image, add top/bottom bars and student ID stamp.

· **Save final frame** and return a dictionary with images and detection results.

1.2.2 Detailed explanation

First, I imported the necessary libraries, created a BoundingBox dataclass

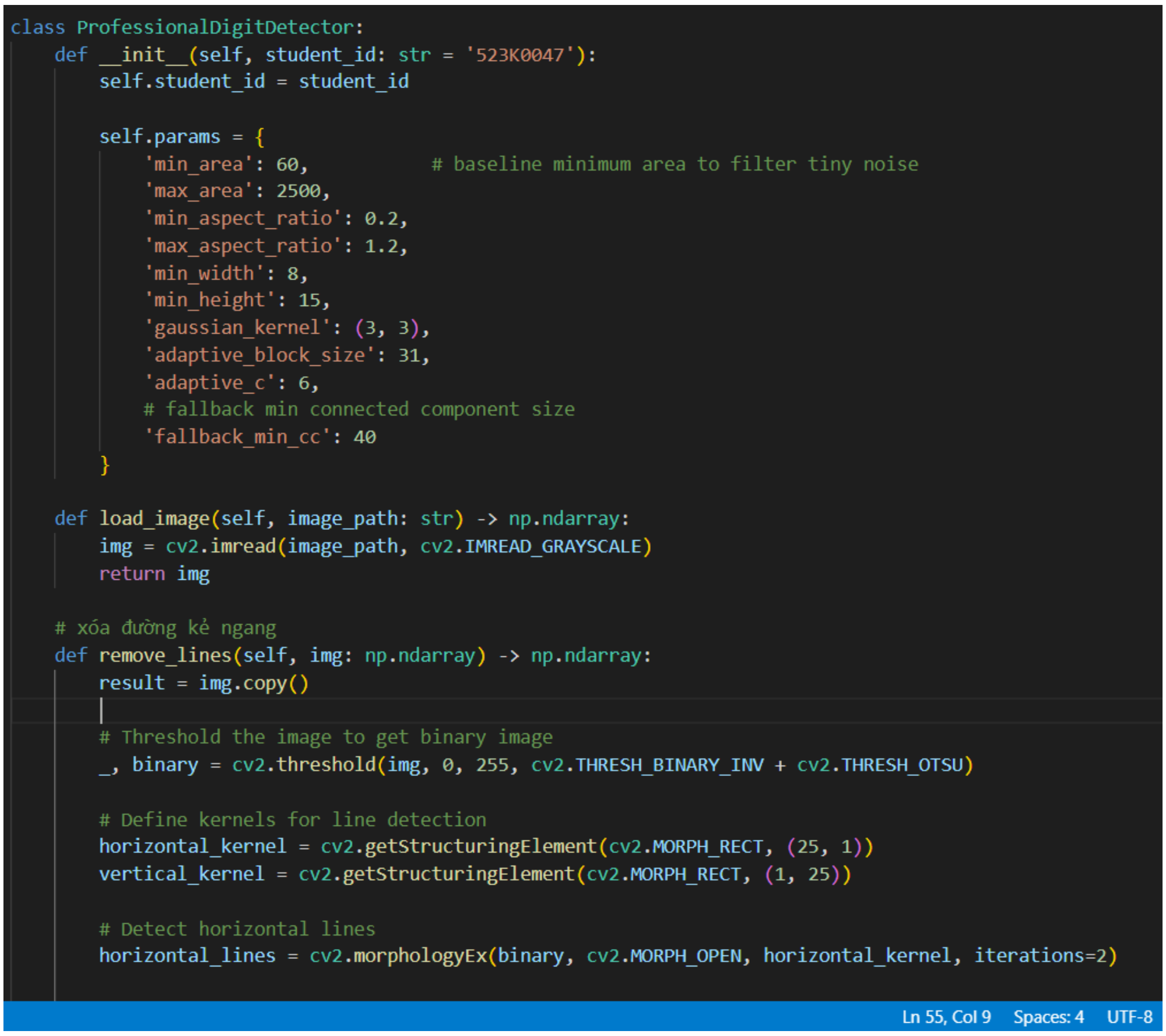


```python
@dataclass
class BoundingBox:
    x: int
    y: int
    width: int
    height: int
    area: float
    contour: np.ndarray = None

    @property
    def aspect_ratio(self) -> float:
        return self.width / self.height if self.height > 0 else 0

    @property
    def center(self) -> Tuple[int, int]:
        return (self.x + self.width // 2, self.y + self.height // 2)

    @property
    def coordinates(self) -> Tuple[int, int, int, int]:
        return (self.x, self.y, self.x + self.width, self.y + self.height)
```

This class defines a simple container with fields for bounding box origin (x,y), width, height, area and optionally the contour (OpenCV contour array).

There are 3 Propertie functions: aspect_ratio Returns width/height (safe against division by zero), ‘center’ Returns integer center coordinate of the bounding box and ‘coordinates’ returns (x1, y1, x2, y2) rectangle coordinates convenient for drawing.

Then I created a big class ‘ProfessionalDigitDetector’ to wrap all detections pipeline inside.



```python
class ProfessionalDigitDetector:
    def __init__(self, student_id: str = '523K0047'):
        self.student_id = student_id

        self.params = {
            'min_area': 60,           # baseline minimum area to filter tiny noise
            'max_area': 2500,
            'min_aspect_ratio': 0.2,
            'max_aspect_ratio': 1.2,
            'min_width': 8,
            'min_height': 15,
            'gaussian_kernel': (3, 3),
            'adaptive_block_size': 31,
            'adaptive_c': 6,
            # fallback min connected component size
            'fallback_min_cc': 40
        }

    def load_image(self, image_path: str) -> np.ndarray:
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        return img

    # xóa đường kẻ ngang
    def remove_lines(self, img: np.ndarray) -> np.ndarray:
        result = img.copy()

        # Threshold the image to get binary image
        _, binary = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

        # Define kernels for line detection
        horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (25, 1))
        vertical_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 25))

        # Detect horizontal lines
        horizontal_lines = cv2.morphologyEx(binary, cv2.MORPH_OPEN, horizontal_kernel, iterations=2)
```

First, self.params centralizes thresholds controlling filtering and preprocessing:

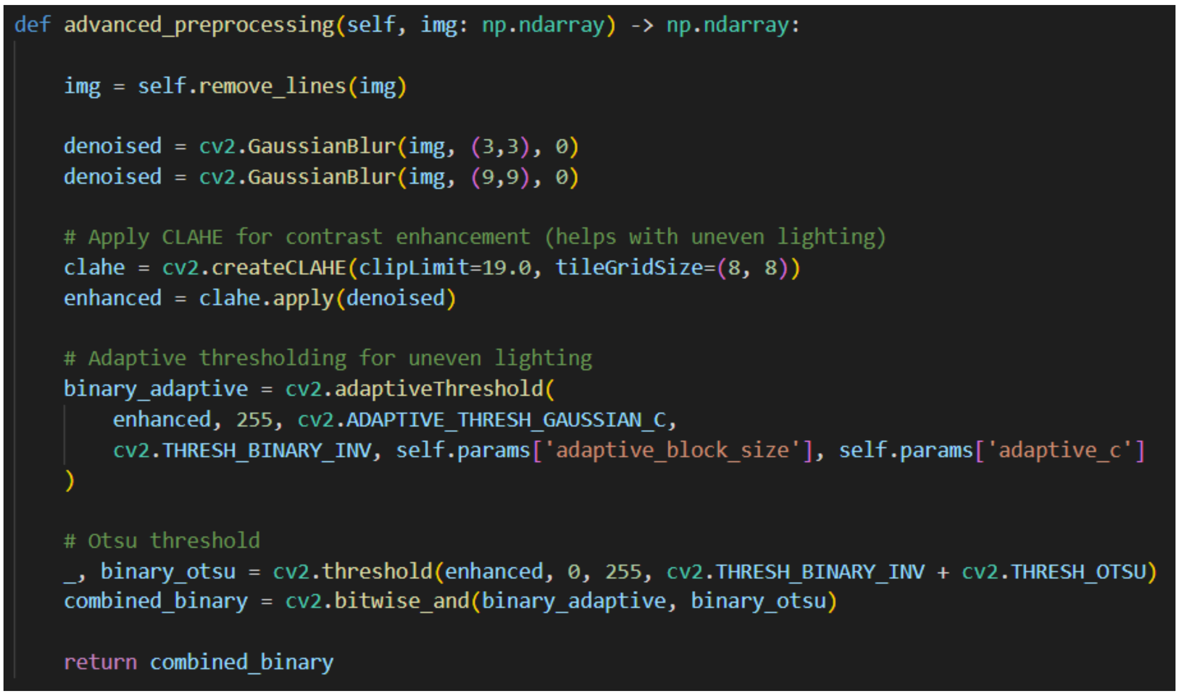
- min_area / max_area used in contour filtering (area in pixels).
- aspect ratio bounds assume digits are roughly taller than wide but allow near-square (0.2–1.2).
- min_width/min_height minimum bounding box size to avoid tiny noise.
- adaptive_block_size & adaptive_c control adaptive thresholding behavior.
- store ‘gaussian_kernel’: declared but in code a different kernel is used too.

I load image and create a function to remove lines in the image, so that it can easier to detect the first number 4 in line 4 because before that i found that the number was stick to the line and hard to detect. It improved.

‘remove_lines’ method tries to remove both horizontal and vertical straight lines (e.g., table/grid) while keeping digits intact:

- 'result' is a copy used for final inpainting.
- cv2.threshold with THRESH_BINARY_INV + OTSU automatically chooses a threshold; inverted binary makes foreground white (255) where originally darker pixels (this code assumes digits/lines are dark on light background).
- Create long thin rectangular kernels: (25,1) for horizontal lines, (1,25) for vertical lines. Sizes tuned to expected line lengths.
- MORPH_OPEN (erosion followed by dilation) with line-shaped kernels isolates line-like components matching kernel shape. iterations=2 increases effect.
- Then, combine both masks to get a unified lines mask.
- Dilation widens the mask to ensure line pixels are fully covered (useful before inpainting).
- Final result is cv2.inpaint filled regions where lines_mask is set, using Telea algorithm with radius=3. This removes the detected lines and attempts to reconstruct background/texture around them.

Next, i create a function to process image is ‘advanced_preprocessing’ with:



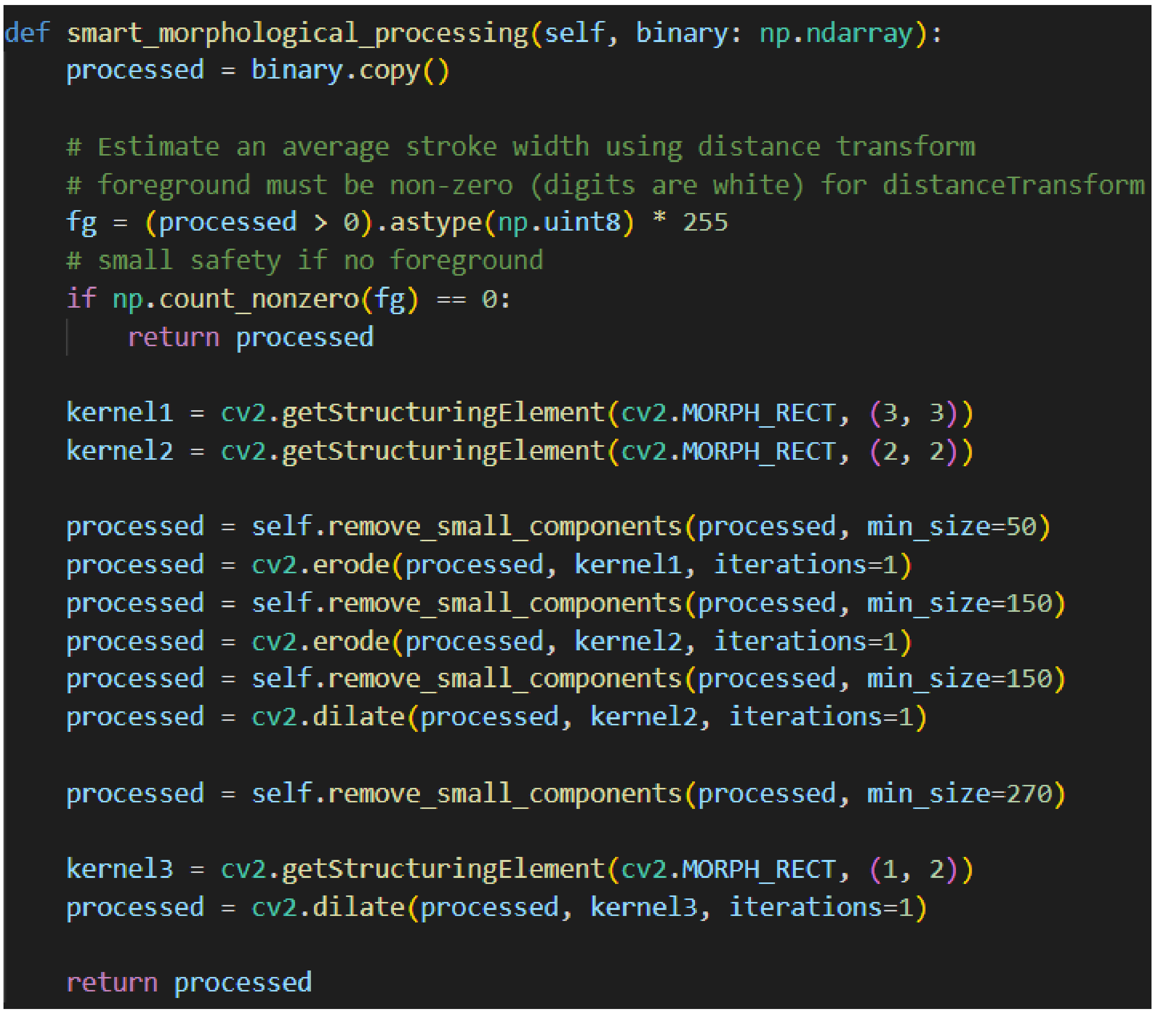
```python
def advanced_preprocessing(self, img: np.ndarray) -> np.ndarray:

    img = self.remove_lines(img)

    denoised = cv2.GaussianBlur(img, (3,3), 0)
    denoised = cv2.GaussianBlur(img, (9,9), 0)

    # Apply CLAHE for contrast enhancement (helps with uneven lighting)
    clahe = cv2.createCLAHE(clipLimit=19.0, tileGridSize=(8, 8))
    enhanced = clahe.apply(denoised)

    # Adaptive thresholding for uneven lighting
    binary_adaptive = cv2.adaptiveThreshold(
        enhanced, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY_INV, self.params['adaptive_block_size'], self.params['adaptive_c']
    )

    # Otsu threshold
    _, binary_otsu = cv2.threshold(enhanced, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    combined_binary = cv2.bitwise_and(binary_adaptive, binary_otsu)

    return combined_binary
```

- First removes lines, then applies Gaussian blur twice. (Note: second line reassigns from img again — as written denoised = cv2.GaussianBlur(img, (9,9),

0) ignores the (3,3) blur result; probably intended to chain both but effectively only the last blur is used. This is a minor bug/inconsistency.)

- CLAHE (Contrast Limited Adaptive Histogram Equalization) improves local contrast. clipLimit=19.0 is high (strong contrast). tileGridSize defines local regions.
- adaptiveThreshold uses a local neighborhood (block size adaptive_block_size) and subtracts adaptive_c to compute threshold per block; THRESH_BINARY_INV makes foreground white (useful for later morphological ops).
- Otsu global thresholding is also computed; bitwise_and of adaptive and Otsu reduces false positives from either method and yields a conservative binary mask (only pixels classified as foreground by both methods remain). This combination helps robustness under uneven illumination.

After that is apply morphology by *smart_morphological_processing*. This "smart" pipeline attempts to make digit shapes stable and blob-like for contour analysis:



```python
def smart_morphological_processing(self, binary: np.ndarray):
    processed = binary.copy()

    # Estimate an average stroke width using distance transform
    # foreground must be non-zero (digits are white) for distanceTransform
    fg = (processed > 0).astype(np.uint8) * 255
    # small safety if no foreground
    if np.count_nonzero(fg) == 0:
        return processed

    kernel1 = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    kernel2 = cv2.getStructuringElement(cv2.MORPH_RECT, (2, 2))

    processed = self.remove_small_components(processed, min_size=50)
    processed = cv2.erode(processed, kernel1, iterations=1)
    processed = self.remove_small_components(processed, min_size=150)
    processed = cv2.erode(processed, kernel2, iterations=1)
    processed = self.remove_small_components(processed, min_size=150)
    processed = cv2.dilate(processed, kernel2, iterations=1)

    processed = self.remove_small_components(processed, min_size=270)

    kernel3 = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 2))
    processed = cv2.dilate(processed, kernel3, iterations=1)

    return processed
```

- Copy the previous processed image to preserve original.
- Convert to binary 0/255, fg required by some OpenCV functions.
- If no foreground present, return immediately.

- Then come to a sequence of morphological operations interleaved with connected-component cleanup:

- remove_small_components(..., 50) removes tiny noise.
- erode(kernel1) thins components slightly (remove spurs), then remove components smaller than 150, etc.

- Erosion then dilation pattern is typical to remove small noise and close small gaps.
- Final remove_small_components(..., 270) ensures only reasonably sized components survive.
- Kernels are small — tuned for digit stroke widths.

The sequence is aggressive, it removes more and more small components across stages, aiming to keep well-formed digit blobs and discard spurious artifacts. After trying many different value numbers for min size and kernel size, and also the order of dilate, erode steps, i've found that the above sequence and their values of parameters are the best.

- 'remove_small_components' function used to remove noise regions. Returns an image containing only components meeting the minimum size threshold.

Afterwarsd, I created 'contour_analysis' function. This method is the main filtering stage that uses geometric heuristics to keep likely digits. When image pass through this function, it performs final connected-component filtering with min_size=85 before contour extraction.

Summary step by step i've used to solve the task:

- Line removal: remove_lines
- Contrast & binarization: advanced_preprocessing (CLAHE + adaptive + Otsu)
- Morphological cleanup: smart_morphological_processing + remove_small_components
- Contour detection & filtering: contour_analysis
- Presentation & drawing: create_professional_frame
- Run end-to-end and return the final result: process and main

- ‘create_frame’ function is for operation such as image styling and draws green rectangles for each detected bounding box. Coordinates adjusted by the offsets used when placing the original image.

Then I created a process function to run main detection process.

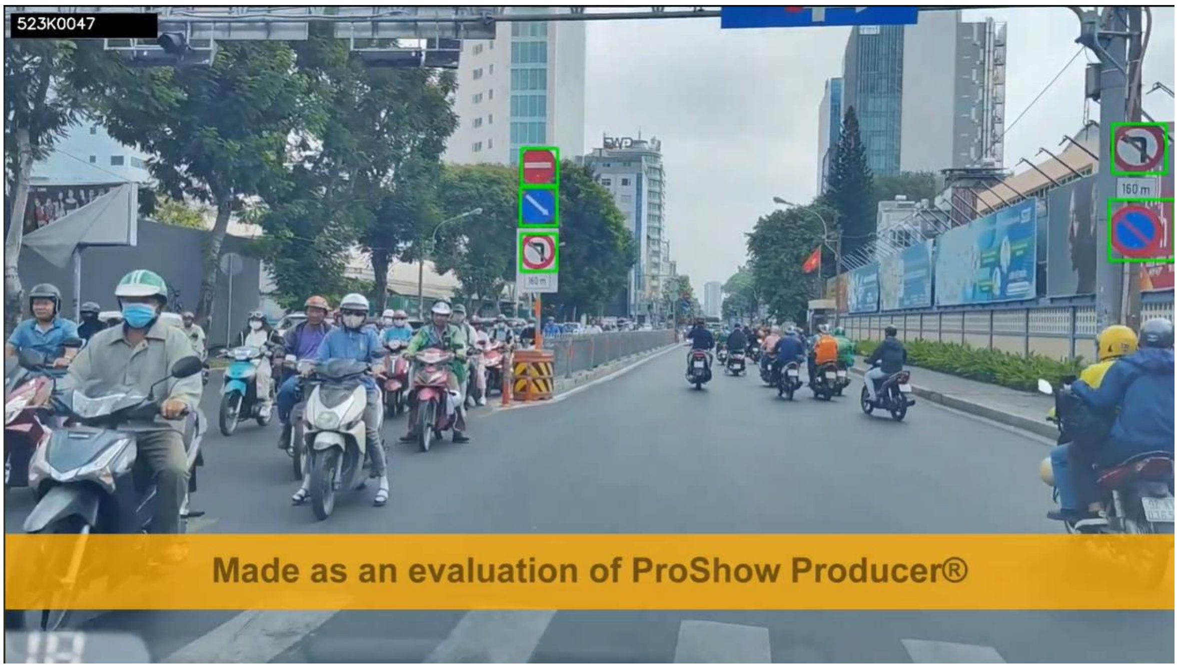
Finally, outside the main class, I created main function to set default filenames and student ID, instantiates the detector, calls process() and prints the detection count. The ‘if __name__ == "__main__"’ ensures main() runs only when script is executed directly.

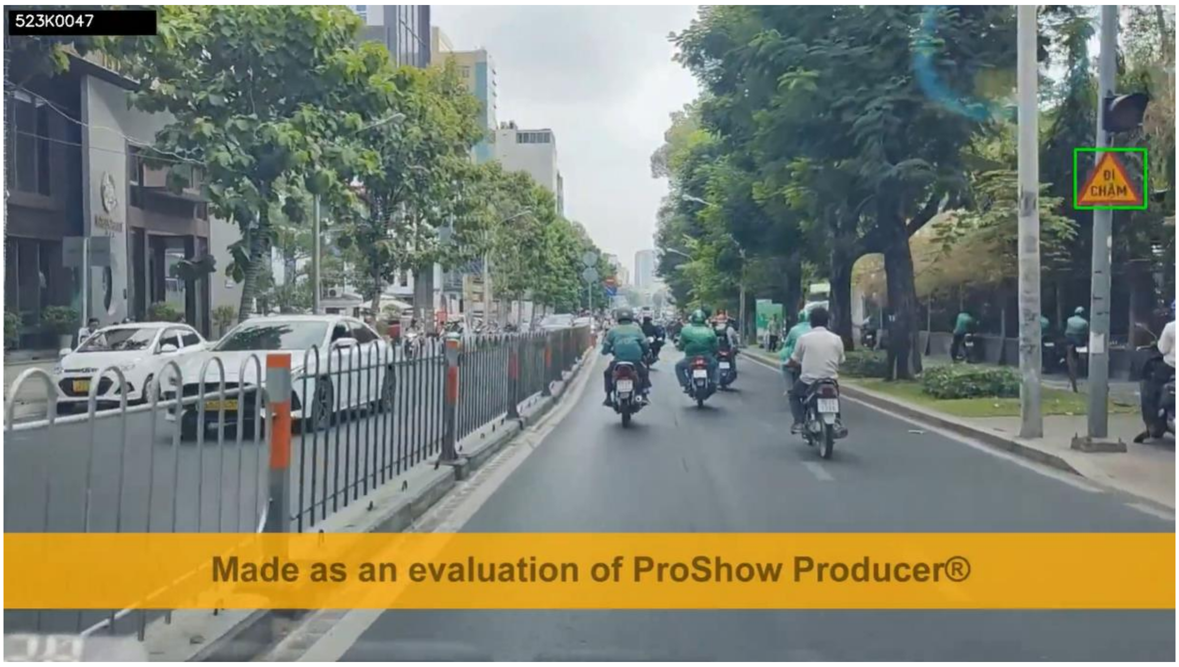
Chapter 2: Task results

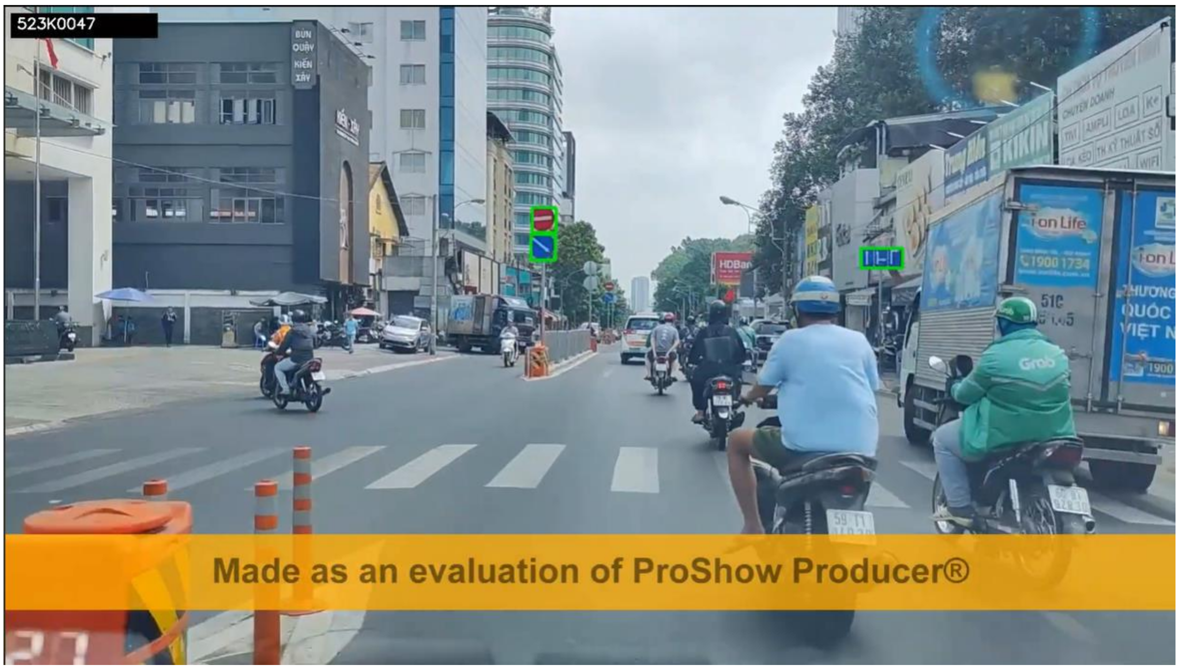
2.1 Task 1:

Overall, the code solution can **detected all major observable traffic signs** of the given video. But there are still some objects not even being a traffic sign have been catched such as billborad,etc. But the number of false positives is small so overall this coding solution solves the problem quite well.

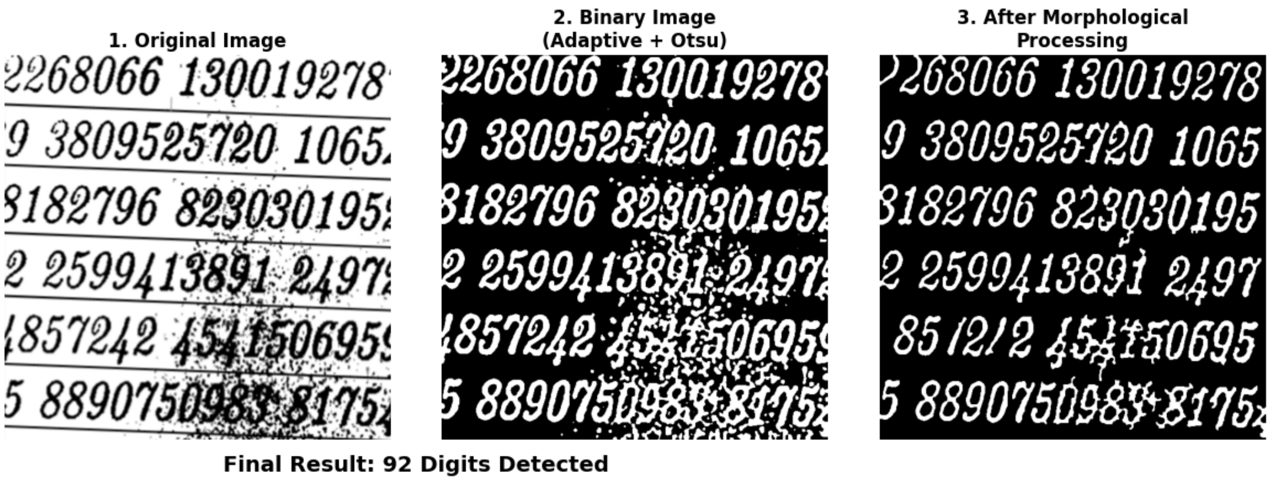
Below are three output frames with rectangles surrounding each traffic sign in three different time of the output video:




523K0047
ĐI CHẬM
Made as an evaluation of ProShow Producer®

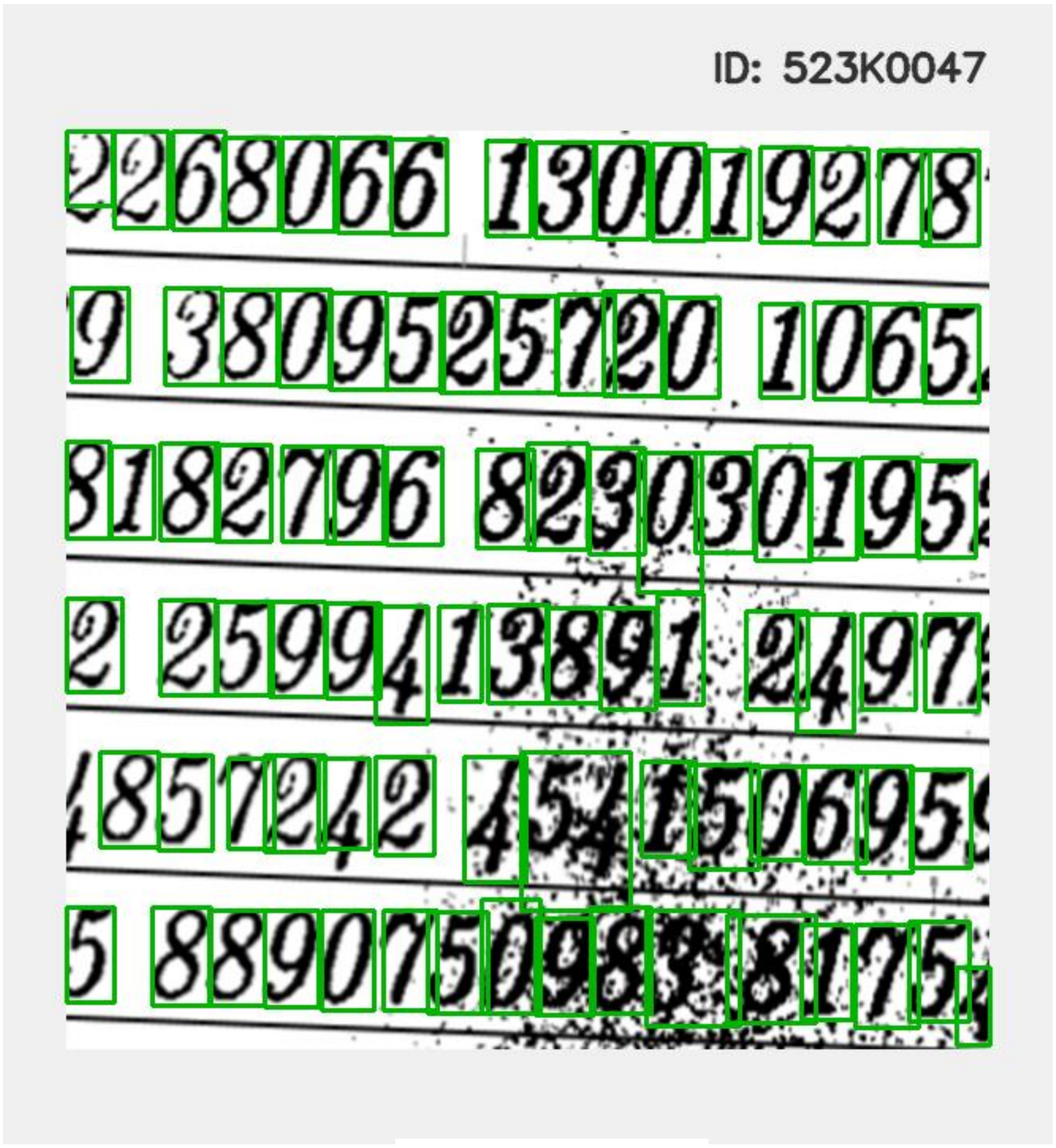

523K0047
Made as an evaluation of ProShow Producer®

2.2 Task 2:


1. Original Image
2. Binary Image (Adaptive + Otsu)
3. After Morphological Processing
Final Result: 92 Digits Detected

Processing steps

In general, the code program detected all numbers. But number 4 and 5 in line 5 still stick together.



The final output image