Faculty of Engineering

Credit Hours Engineering Programs

Computer Engineering and SoftwareSystems Program

Academic Year 2024/2025 – Fall 2024

CSE-483 Computer Vision

Milestone 1

....

Name	ID
Omar Sameh Mohammed	21P0204
Youssef Mohamed Zaki	21P0079
Michael Joseph Adeeb	1901075
Adel Mohamed Adel	21P0113
Mostafa Hassan Mohamed	21P0349

Contents

1. Introduction

Project Background

This project focuses on implementing a robust system to detect and decode barcodes from images under various conditions, including rotation, noise, and varying lighting. The system leverages computer vision techniques to preprocess, detect, and decode barcodes, ensuring accuracy and reliability.

Purpose

The purpose of this documentation is to outline the objectives, design, and implementation details for Milestone 1 of the project. This milestone includes the successful implementation of preprocessing steps, barcode detection, and decoding functions.

2. Project Objectives

Primary Objectives

- Develop a system to detect barcodes in images with varying orientations and noise levels.
- Decode barcodes into their respective sequences.

Specific Goals

- Preprocess images to handle noise and rotation.
- Implement algorithms to detect salt-and-pepper noise.
- Decode barcodes accurately and efficiently.

3. System Overview

Architecture

The system consists of the following key modules:

1. **Preprocessing Module:** Handles image transformations, noise detection, and corrections.
2. **Detection Module:** Identifies barcode regions within the image.
3. **Decoding Module:** Interprets the detected barcode into readable data.

Flow Diagram

1. Load Image.
2. Preprocess Image:
    - Grayscale Conversion.
    - Separate from background .
    - Noise Detection and Removal.
    - Rotation Correction.
3. Detect Barcode Region.
4. Decode Barcode.

4. Functional Requirements

Preprocessing

- Convert input images to grayscale.
- Detect and fix salt-and-pepper noise.
- Correct image orientation to align barcode horizontally.

Detection

- Use morphological operations to isolate the barcode region.
- Identify the largest contour representing the barcode.

Decoding

- Extract barcode patterns from the detected region.
- Interpret patterns into alphanumeric sequences using Code 11 standards.

5. Technical Design

Preprocessing

Grayscale Conversion

- Convert images to grayscale using OpenCV's cvtColor function.

Noise Detection

- Threshold images to detect salt-and-pepper noise.
- Calculate the percentage of noisy pixels relative to the image size.

Noise Removal

- Apply median filtering to eliminate detected noise.

Rotation Correction

- Use contour detection and minAreaRect to estimate the rotation angle of the barcode.
- Correct the orientation using affine transformations.

Barcode Detection

Morphological Operations

- Apply morphological closing with a kernel proportional to the barcode's bar width.

Contour Analysis

- Find and select the largest contour corresponding to the barcode region.

Decoding

Pattern Analysis

- Analyze bar widths to distinguish between narrow and wide bars.
- Match patterns against the Code 11 standard to retrieve the barcode sequence.

6. Implementation

Core Functions

Preprocessing Functions

```python
def preprocess(image):
    #Step 1: Extract grayscale regions from an image using low saturation in the HSV color space
    isolated = extract_grayscale_regions_hsv(image, saturation_threshold=30)
    #Step 2: Convert the image to grayscale
    gray = cv2.cvtColor(isolated, cv2.COLOR_BGR2GRAY)
    
    #Step 3: Blur the image to reduce noise using an average 3x1 kernel.
    kernel = np.array([[1/3], [1/3], [1/3]])
    blurred = cv2.filter2D(gray,-1,kernel)

    #Step 4: Detect and Fix Salt and Pepper.
    if detect_salt_and_pepper(blurred):
        blurred = fix_salt_and_pepper(blurred)
    
    #Step 5: Thresholding to separate the barcode from the background.
    _, binary_img = cv2.threshold(blurred, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

    #Step 6: Detect the rotaion angle of the barcode and rotate accordingly
    binary_img = Handle_rotation(binary_img)

    #Step 7: Extract the barcode from the image
    barcode = detectBarcode(binary_img)

    #Step 8: Apply median bluring and morphological opening vertically to optimize extracted barcode
    w, h = barcode.shape[:2]
    blured_barcode = cv2.medianBlur(cv2.blur(barcode,(1,h)),1) #average in vertical direction to dillute white pixels in bars, black pixels in spaces
    _, binary_barcode = cv2.threshold(blured_barcode, 200, 255, cv2.THRESH_BINARY) #threshold to remove gray (dilluted) pixels
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, h))
    final = cv2.morphologyEx(binary_barcode, cv2.MORPH_OPEN, kernel) #connect bars
    
    return final
```

- extract_grayscale_regions_hsv(input_img): Extracts grayscale regions using HSV color space.

```
def extract_grayscale_regions_hsv(input_img, saturation_threshold=30):
    """ Extract grayscale regions from an image using low saturation in the HSV color space."""
    # Convert the image to HSV color space
    hsv_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2HSV)

    # Split the HSV channels
    h, s, v = cv2.split(hsv_img)

    # Create a mask for low-saturation regions
    low_saturation_mask = s < saturation_threshold

    # Convert the mask to binary
    binary_mask = low_saturation_mask.astype(np.uint8) * 255

    # Create a white background
    white_background = np.ones_like(input_img, dtype=np.uint8) * 255

    # Use the mask to isolate low-saturation regions (grayscale)
    grayscale_regions = cv2.bitwise_and(input_img, input_img, mask=binary_mask)

    # Invert the mask
    inverted_mask = cv2.bitwise_not(binary_mask)
    grayscale_regions += cv2.bitwise_and(white_background, white_background, mask=inverted_mask)

    return grayscale_regions
```

- Blur the image to reduce noise using an average 3x1 kernel.

```
kernel = np.array([[1/3], [1/3], [1/3]])
blurred = cv2.filter2D(gray,-1,kernel)
```

- detect_salt_and_pepper(image): Identifies salt-and-pepper noise.

```
def detect_salt_and_pepper(image):
    '''
        Input: image
            detects salt and pepper regardless of the image's brigtness
        output: boolean indicating if there is salt and pepper nosie
    '''
    height, width = image.shape[:2]
    noise_pixels = 0
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    _, image = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

    #the idea is to loop through the pixels and compare each to the surrounding. this is probably not optimal performance wise but makes the program
    # generic. accurately detecting salt and pepper in images regardless of brightness and other noise (like sin)
    for i in range(1, height - 1):
        for j in range(1, width - 1):

            center_pixel = image[i, j]

            surrounding_pixels = [
                image[i-1, j-1], image[i-1, j], image[i-1, j+1], image[i, j-1],
                image[i, j+1], image[i+1, j-1], image[i+1, j], image[i+1, j+1]]

            x=0
            #stack-like way to compare the pixel to its surroundings. remember the image was thresholded so the pixels are black and white only
            for k in range(0,7):
                if surrounding_pixels[k] == center_pixel:
                    x+=1
                else:
                    x-=1

            if x < 0:
                noise_pixels += 1

            if x < 0:
                noise_pixels += 1

    image_size = height*width
    percentage_noise = 100*noise_pixels / (height*width) #compare salt/pepper pixels to total pixels. to calculate an approximate percentage
    print(f"Total pixels in image:  {image_size}")
    print(f"Total noisy pixels: {noise_pixels}")
    print(f"Salt and Pepper approximate percentage: {percentage_noise} %")

    return percentage_noise > 2
```

- fix_salt_and_pepper(image): Removes salt-and-pepper noise using median filtering.

```python
def fix_salt_and_pepper(image):

    kernel = np.array([[1/5], [1/5], [1/5]])
    image = cv2.filter2D(image,-1,kernel)
    _, image = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

    #image = cv2.medianBlur(cv2.blur(image,(1,h)),1)
    #Q_, image = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)

    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 10))
    image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel, iterations = 3)
    #display(image, "1")

    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (2, 1))
    image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
    #display(image, "2")

    return image
```

- Thresholding to separate the barcode from the background.

```python
_, binary_img = cv2.threshold(blurred, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)
```

- Detect the rotation angle of the barcode and rotate accordingly

```python
binary_img = Handle_rotation(binary_img)
```

- Extract the barcode from the image

```python
barcode = detectBarcode(binary_img)
```

- Apply median bluring and morphological opening vertically to optimize extracted barcode

```
w, h = barcode.shape[:2]
blured_barcode = cv2.medianBlur(cv2.blur(barcode,(1,h)),1) #average in vertical direction to dillute white pixels in bars, black pixels in spaces
_, binary_barcode = cv2.threshold(blured_barcode, 200, 255, cv2.THRESH_BINARY) #threshold to remove gray (dilluted) pixels
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, h))
final = cv2.morphologyEx(binary_barcode, cv2.MORPH_OPEN, kernel) #connect bars

return final
```

Detection:

- detect_distance_between_lines(gray): Calculates the maximum distance between bars.

```
def detect_distance_between_lines(gray):
    '''
        Input: image
            finds the barcode from the image to be ready for decoding
        output: max distance between bars. used to dilate the barcode to make it 1 object
    '''

    #Thresholding the grayscale image to create a binary image. Inverts the binary image
    # This is done because we are interested in detecting the white bars on a dark background (barcode).
    _, binary_image = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY_INV)

    # apply canny edge detection
    edges = cv2.Canny(binary_image, 50, 150, apertureSize=3)

    # Use Hough Line Transform to detect lines in the edge-detected image
    # HoughLinesP is a probabilistic version of the Hough Line Transform that returns a list of lines.
    # 1 is the resolution of the accumulator in pixels.
    # np.pi / 180 is the resolution of the angle in radians.
    # The threshold is the minimum number of intersections in the accumulator to detect a line.
    # minLineLength: minimum length of a line to be detected.
    # maxLineGap: maximum gap between lines to be considered as a single line.
    lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=100, minLineLength=100, maxLineGap=10)

    # Make a copy of the grayscale image to draw lines on
    line_image = np.copy(gray)

    if lines is not None:
        # Sort by x
        lines = sorted(lines, key=lambda x: x[0][0])

        distances = []
        maxdistance = 0

        # loop through the sorted lines and calculate distances between consecutive lines
```

```
        # loop through the sorted lines and calculate distances between consecutive lines
        for i in range(1, len(lines)):
            x1, y1, x2, y2 = lines[i-1][0]
            x3, y3, x4, y4 = lines[i][0]

            # Check if the current line is approximately parallel to the previous line to filter irregular lined
            if abs(x3 - x1) == abs(x4 - x2):

                distance = abs(x3 - x1)

                if distance > maxdistance:
                    maxdistance = distance

                distances.append(distance)

            #cv2.line(line_image, (x1, y1), (x2, y2), 1)
            #cv2.line(line_image, (x3, y3), (x4, y4), 1)

        print(f"Distance between barcode bars: {maxdistance}")
        return maxdistance

    else:
        print("No lines detected.")
        return -1
```

- detectBarcode(img): Detects and isolates the barcode region.

```
def detectBarcode(img):
    '''
        Input: image
            finds the barcode from the image to be ready for decoding
        output: extracted barcode
    '''
    #maxdistance  is max distance between bars. this is used to open the barcode to make is 1 object (largest contour)
    maxdistance = detect_distance_between_lines(img)
    #print(maxdistance)

    #use a structuring element based on the distance between bars to open the barcode horizontally. maxdistance is required for the code to not be test cae deppendant.
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (maxdistance+1, 1)) #+1 3shan maxdistance is 1 pixel short
    closed_image = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)

    #apply thresholding
    _, closed_image_2 = cv2.threshold(closed_image, 150, 255, cv2.THRESH_BINARY_INV)
    contours, _ = cv2.findContours(closed_image_2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    largest_contour = None
    max_area = 0

    #loop through contours to find the largest one which is the barcode
    for contour in contours:
        area = cv2.contourArea(contour)
        if area > max_area:
            max_area = area
            largest_contour = contour

    #use the dimensions of the contour to crop the barcode from the original image
    x, y, deltax, deltay = cv2.boundingRect(largest_contour)
    mabrook = img[y:y+deltay, x:x+deltax]

    return mabrook
```

Decoding Functions

- decode(out): Decodes the extracted barcode region into alphanumeric sequences.

6. Testing Cases:

1-Test case1 :

```
Test Case 1
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 447
Salt and Pepper approximate percentage: 0.093125 %
Distance between barcode bars: 8
```

Test Case 2:

```
Test Case 2
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 349
Salt and Pepper approximate percentage: 0.07270833333333333 %
Distance between barcode bars: 8
```

Test case 3:

```
Test Case 3
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 377
Salt and Pepper approximate percentage: 0.07854166666666666 %
Distance between barcode bars: 8
```

Test Case 4:

```
Test Case 4
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 461
Salt and Pepper approximate percentage: 0.09604166666666666 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '-', '4', '7', '-', '4', '7', '-', '1', '2', '1', '-', 'Stop/Start']
Correct code:
['Stop/Start', '-', '4', '7', '-', '4', '7', '-', '1', '2', '1', '-', 'Stop/Start']
*****************************************************************************
```

Test Case5:

```
Test Case 5
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 302
Salt and Pepper approximate percentage: 0.06291666666666666 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '1', '1', '1', '-', '1', '1', '7', '-', '1', '1', '6', 'Stop/Start']
Correct code:
['Stop/Start', '1', '1', '1', '-', '1', '1', '7', '-', '1', '1', '6', 'Stop/Start']
```

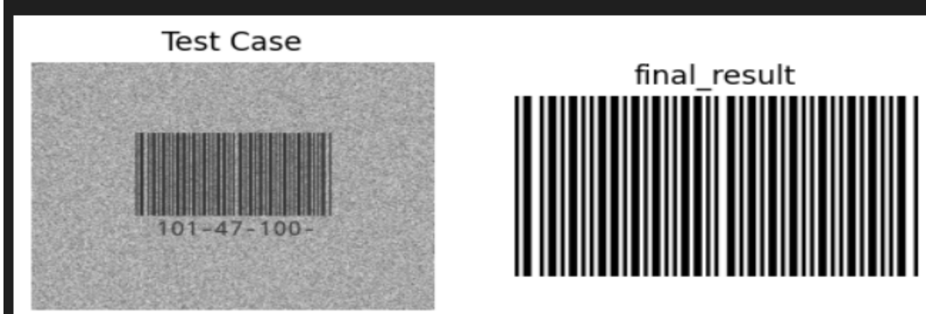
Test Case 6:

```
Test Case 6
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 432
Salt and Pepper approximate percentage: 0.09 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '-', '1', '1', '7', '-', '4', '6', '-', '9', '8', '-', 'Stop/Start']
Correct code:
['Stop/Start', '-', '1', '1', '7', '-', '4', '6', '-', '9', '8', '-', 'Stop/Start']
******************************************************************************
```

Test Case 7:

```
Test Case 7
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 121086
Salt and Pepper approximate percentage: 25.22625 %
Distance between barcode bars: 8
```




```
Detected Barcode:
['Stop/Start', '1', '0', '1', '-', '4', '7', '-', '1', '0', '0', '-', 'Stop/Start']
Correct code:
['Stop/Start', '1', '0', '1', '-', '4', '7', '-', '1', '0', '0', '-', 'Stop/Start']
******************************************************************************
```

Test Case 8:

```
Test Case 8
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 502
Salt and Pepper approximate percentage: 0.10458333333333333 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '1', '1', '3', '-', '1', '1', '9', '-', '5', '2', '-', 'Stop/Start']
Correct code:
['Stop/Start', '1', '1', '3', '-', '1', '1', '9', '-', '5', '2', '-', 'Stop/Start']
```

Test Case 9:

```
Test Case 9
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 5811
Salt and Pepper approximate percentage: 1.210625 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '1', '1', '9', '-', '5', '7', '-', '1', '1', '9', '-', 'Stop/Start']
Correct code:
['Stop/Start', '1', '1', '9', '-', '5', '7', '-', '1', '1', '9', '-', 'Stop/Start']
*******************************************************************************
```

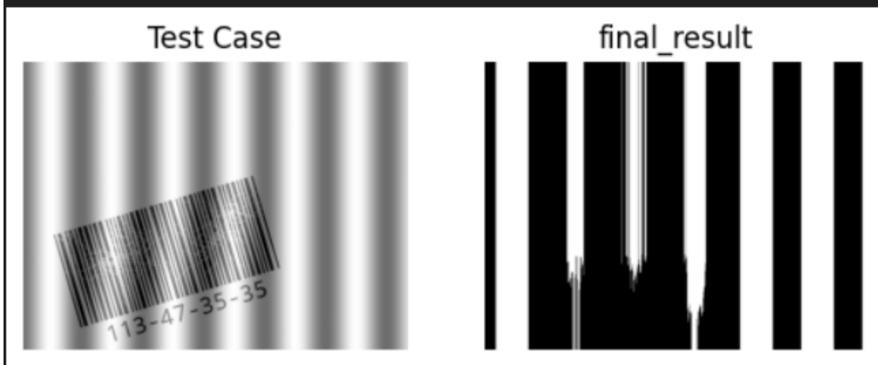
Test case 10:

```
******************************************************************************
Test Case 10
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 502
Salt and Pepper approximate percentage: 0.10458333333333333 %
Distance between barcode bars: 8
```


```
Detected Barcode:
['Stop/Start', '1', '0', '3', '-', '1', '2', '0', '-', '9', '9', '-', 'Stop/Start']
Correct code:
['Stop/Start', '1', '0', '3', '-', '1', '2', '0', '-', '9', '9', '-', 'Stop/Start']
```

Test Case11:

```
*******************************************************************************
Test Case 11
Relevant information:
Total pixels in image:  480000
Total noisy pixels: 3129
Salt and Pepper approximate percentage: 0.651875 %
Distance between barcode bars: 68
```




```
Detected Barcode:
[]
Correct code:
['Stop/Start', '1', '1', '3', '-', '4', '7', '-', '3', '5', '-', '3', '5', 'Stop/Start']
*******************************************************************************
```