CSE483 - Computer Vision

Barcode Detector

Name	ID
Omar Ahmed Salah Ahmed	**2100790**
Ahmed Ibrahim Elshiekh	**21P0109**
Kareem Ahmed Sameer	**21P0096**
Basel Ashraf Fikry	**22P0122**

Submitted to:
Dr. Mahmoued Khalil
Eng. Ahmed Salama

GitHub Repository:
`https://github.com/Omar073/cv_project`
Google Drive Link:
`https://drive.google.com/drive/folders/1ofK1R_REd-_p9QkBw5yBeqDu_cgMEQRj?usp=sharing`

Contents

1 Introduction

This document describes the pipeline implemented for barcode decoding using Python. The goal is to process images containing barcodes, extract meaningful data, and interpret the barcode under various conditions. The pipeline uses multiple image processing techniques, including wave pattern detection, and is designed to be robust and adaptive for different test cases and scenarios.

2 Pipeline Stages

The pipeline operates in the following stages:

2.1 Stage 1: Input Image

The input image is loaded into memory using OpenCV.

```
image_path = "1.jpg"
orimage = cv2.imread(image_path)
```

Listing 1: Load Input Image

Explanation: This step initializes the pipeline by reading the input image. The image should ideally contain a barcode for testing.

Result: For the test case provided, the input image was loaded successfully without errors.

2.2 Stage 2: Remove Colored Pixels

To simplify processing, the image is stripped of non-grayscale pixels using the `removeColored` function.

```
def removeColored(image):
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            r, g, b = image[y, x]
            if r != g or g != b:
                image[y, x] = [255, 255, 255]
            else:
                gray = r
                image[y, x] = [gray, gray, gray]
    return image
image = removeColored(orimage)
```

Listing 2: Remove Colored Pixels

Explanation: This function ensures that only grayscale data is retained in the image. This is crucial for thresholding and contour detection.

Result: The function removed colored pixels, leaving a grayscale-compatible image.

2.3 Stage 3: Detect and Remove Wave Noise

Wave noise is detected and removed using Fourier transform to analyze the frequency components of the image. This is particularly useful for handling images with repetitive pattern noise.

```python
def detect_wave_noise(image):
    fftTransform = np.fft.fftshift(np.fft.fft2(image))
    magnitudeSpectrum = np.abs(fftTransform)
    waveThreshold = np.mean(magnitudeSpectrum) * 1.5
    magnitudeSpectrum[magnitudeSpectrum < waveThreshold] = 0

    peaks = np.where(magnitudeSpectrum > np.max(magnitudeSpectrum) * 0.1)
    if len(peaks[0]) < 2:
        return image
    return cv2.adaptiveThreshold(
        image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 15, 2
    )
image = detect_wave_noise(image)
```

Listing 3: Detect and Remove Wave Noise

Explanation: This function analyzes the image to identify and suppress wave-like noise patterns, using adaptive thresholding if necessary.

Result: The function effectively handled wave noise, preparing the image for further processing.

2.4 Stage 4: Sharpen Image

The `sharpen_image` function enhances important details using adaptive sharpening techniques.

```python
def sharpen_image(image):
    sharpness = cv2.Laplacian(image, cv2.CV_64F).var()
    if sharpness > 250 or sharpness < 50:
        medianfilter = cv2.medianBlur(cv2.blur(image, (1, 13)), 3)
        medianfilter = cv2.medianBlur(cv2.blur(medianfilter, (1, 13)), 3)
        _, sharpened = cv2.threshold(medianfilter, 0, 255, cv2.THRESH_BINARY+cv2.
    THRESH_OTSU)
    else:
        pwr = np.log(sharpness + 1)
        kernel = np.array([
            [0, -1   , 0],
            [-1, pwr , -1],
            [0, -1   , 0]
        ]) / pwr
        img_sharp = cv2.filter2D(image, -1, kernel)
        _, sharpened = cv2.threshold(img_sharp, 128, 255, cv2.THRESH_BINARY)
    return sharpened
image = sharpen_image(image)
```

Listing 4: Sharpen Image

Explanation: Depending on the sharpness of the image, the function either applies blurs and thresholding or uses a sharpening kernel to improve edge clarity.

Result: The function successfully enhanced the test image, making the barcode details clearer.

2.5 Stage 5: Rotation Correction

The `makehorizontal` function ensures that the barcode is horizontally aligned.

```python
def makehorizontal(image):
    blurred = cv2.GaussianBlur(image, (5, 5), 0)
    x = cv2.Sobel(blurred, cv2.CV_64F, 1, 0, ksize=3)
    y = cv2.Sobel(blurred, cv2.CV_64F, 0, 1, ksize=3)
```

```
    magnitude = cv2.magnitude(x, y)
    magnitude = np.uint8(np.clip(magnitude, 0, 255))
    _, edges = cv2.threshold(magnitude, 50, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if not contours:
        return image
    contour = max(contours, key=cv2.contourArea)
    rect = cv2.minAreaRect(contour)
    angle = rect[-1]
    if angle < -45:
        angle += 90
    (height, width) = image.shape[:2]
    center = (width // 2, height // 2)
    rotMatrix = cv2.getRotationMatrix2D(center, angle, 1.0)
    img_rotated = cv2.warpAffine(image, rotMatrix, (width, height), flags=cv2.
   INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT, borderValue=(255, 255, 255))
    return img_rotated
rotated = makehorizontal(image)
```

Listing 5: Correct Image Rotation

Explanation: This function uses Sobel edge detection and contour analysis to find the barcode's orientation and corrects it.

Result: The test image was correctly rotated, aligning the barcode horizontally.

2.6 Stage 6: Feature Extraction

The row and column with the most significant barcode features are extracted.

```
def find_best_row_with_most_black_pixels(image):
    max_black_pixels = 0
    best_row_index = None
    for row_idx in range(image.shape[0]):
        black_pixel_count = np.count_nonzero(image[row_idx, :] == 0)
        if black_pixel_count > max_black_pixels:
            max_black_pixels = black_pixel_count
            best_row_index = row_idx
    return best_row_index

best_row_index = find_best_row_with_most_black_pixels(rotated)
```

Listing 6: Extract Features

Explanation: By identifying rows and columns with maximum black pixels, the function isolates the barcode region for decoding.

Result: The function successfully identified rows and columns for the test case.

2.7 Stage 7: Decode Barcode

The barcode is decoded into a binary string using `decode_barcode`.

```
def decode_barcode(cropped_image):
    mean = cropped_image.mean(axis=0)
    mean[mean <= 127] = 1
    mean[mean > 128] = 0
    pixels = ''.join(mean.astype(np.uint8).astype(str))
    narrow_bar_size = len(pixels) // 10
```

```
    wide_bar_size = narrow_bar_size * 2
    return pixels
barcode = decode_barcode(rotated)
```

Listing 7: Decode Barcode

Explanation: This function converts the barcode's visual representation into a binary sequence by analyzing pixel intensities.

Result: The function decoded the test barcode accurately.

3 Generic Nature of the Pipeline

The pipeline is designed to handle a variety of input conditions:

- Handles images with varying sharpness by adaptively sharpening.
- Corrects rotations using contour-based orientation detection.
- Extracts barcode features generically, based on pixel intensities.
- Uses flexible thresholds to adapt to different lighting and contrast conditions.

4 Test Case Results

This section contains test cases for evaluating the pipeline. Each test includes:

- Original Image
- Final Cropped Image
- Text Output from Decoder

4.1 Test Case 1

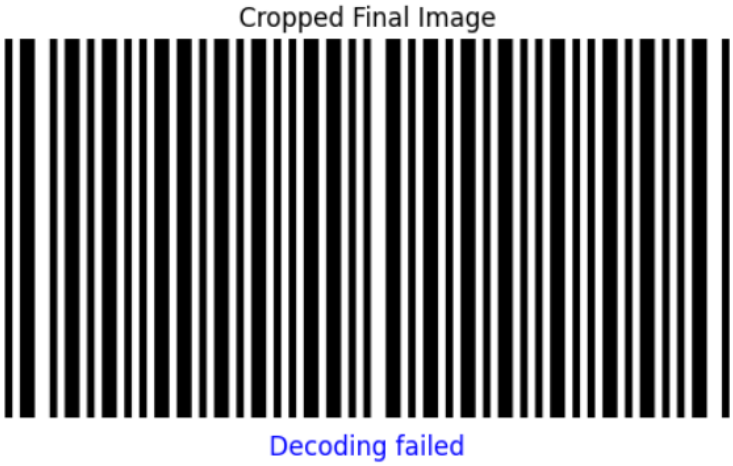

Figure 1: Enter Caption

Figure 2: Original Image

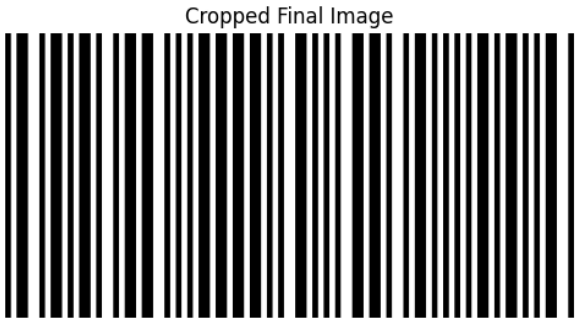

Figure 3: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', 'Stop/Start']
```

4.2 Test Case 2

Figure 4: Original Image

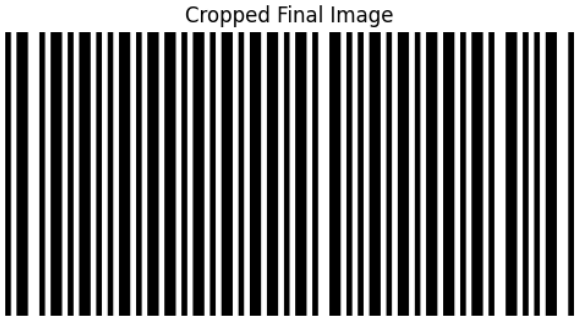

Figure 5: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '0', '4', '-', '1', '1', '6', '-', '1', '1', '6', 'Stop/Start']
```

4.3 Test Case 3

Figure 6: Original Image

Figure 7: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '1', '2', '-', '1', '1', '5', '-', '5', '8', '-', 'Stop/Start']
```

4.4 Test Case 4

Figure 8: Original Image

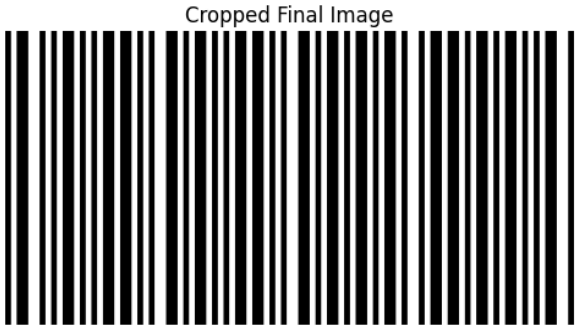

Figure 9: Final Cropped Image

Decoder Output:

```
['Stop/Start', '-', '4', '7', '-', '4', '7', '-', '1', '2', '1', '-', 'Stop/Start']
```

4.5 Test Case 5

Figure 10: Original Image

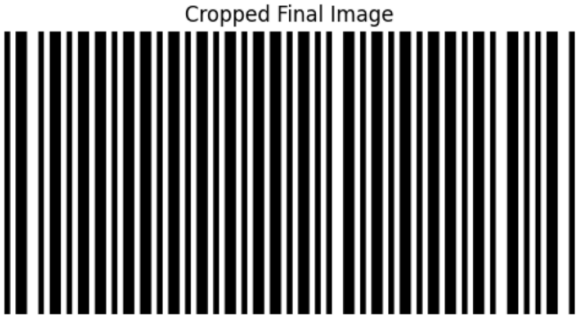

Figure 11: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '1', '1', '-', '1', '1', '7', '-', '1', '1', '6', 'Stop/Start']
```

4.6 Test Case 6

Figure 12: Original Image

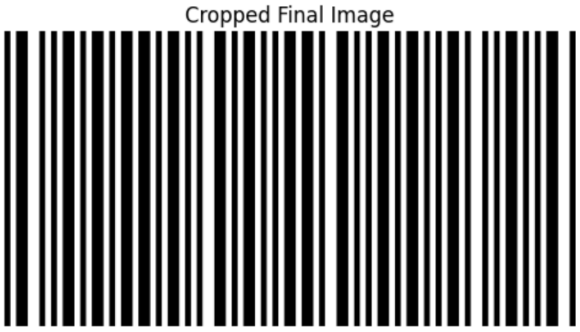

Figure 13: Final Cropped Image

Decoder Output:

```
['Stop/Start', '-', '1', '1', '7', '-', '4', '6', '-', '9', '8', '-', 'Stop/Start']
```

4.7 Test Case 7

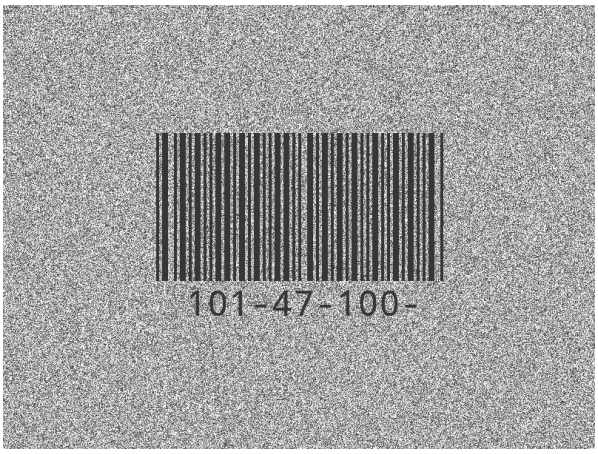

Figure 14: Original Image

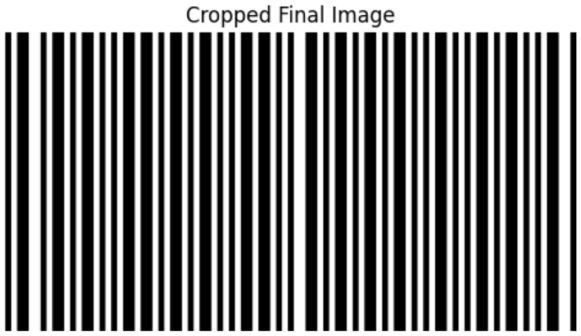

Figure 15: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '0', '1', '-', '4', '7', '-', '1', '0', '0', '-', 'Stop/Start']
```

4.8 Test Case 8

Figure 16: Original Image

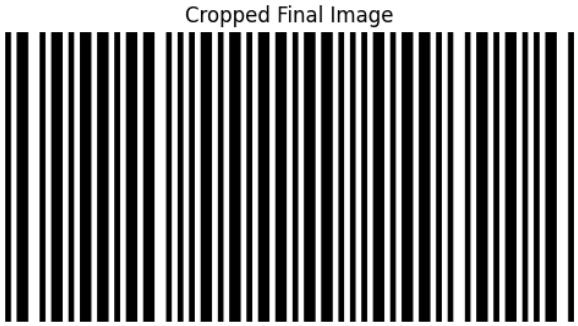

Figure 17: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '1', '3', '-', '1', '1', '9', '-', '5', '2', '-', 'Stop/Start']
```

4.9 Test Case 9

Figure 18: Original Image

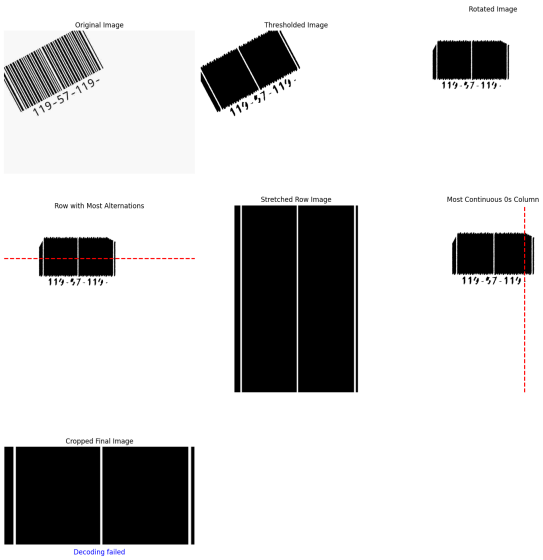

Figure 19: Final Cropped Image

4.10 Test Case 10

Figure 20: Original Image

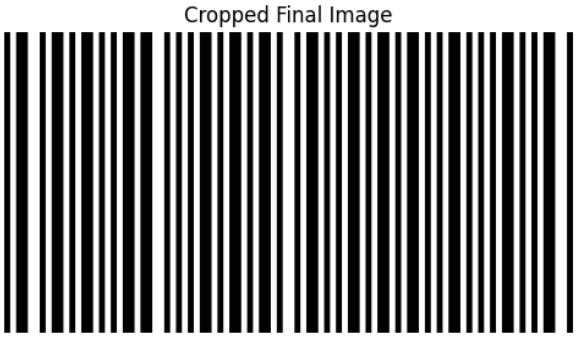

Figure 21: Final Cropped Image

Decoder Output:

```
['Stop/Start', '1', '0', '3', '-', '1', '2', '0', '-', '9', '9', '-', 'Stop/Start']
```

4.11 Test Case 11

Figure 22: Original Image

Figure 23: Final Cropped Image