1 Our Process

1.1 Loading Images

First, we start by mounting our drive and loading the image paths from it to be stored in an array named **image_paths**. We also have **load_images** function that load the images using cv2 library.

```
import os
import os.path as op
import numpy as np
import cv2
import matplotlib.pylab as plt
from google.colab.patches import cv2_imshow
from google.colab import drive
drive.mount('/content/drive')

image_paths = [
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code1.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code2.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code3.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code4.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code5.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code6.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code7.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code8.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code9.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code10.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code11.png",
    "/content/drive/MyDrive/Colab Notebooks/Barcodes/bar_code12.png"
]
# Function to load images
def load_images(image_paths):
    images = []
    invalid_paths = []

    for path in image_paths:
        img = cv2.imread(path)
        if img is not None:
            images.append(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        else:
            invalid_paths.append(path)
    # Error handling
    if invalid_paths:
        print(f"Warning: Unable to load the following images:\n{invalid_paths}")

    return images
```

1.2 Displaying images

Second, We have a simple function that displays the given array of images and their corresponding titles. We are customizing the function to display a maximum of 12 images in a grid. The function also takes cols, whitch is the number of columns in the grid. The default is set to be 4 and the figsize size for the plot. The default is set to be (15, 10).

```
# Function to display images with titles
def display_images(images, titles, cols=4, figsize=(15, 10), max_per_page=12):

    # Limit the number of images to 12 "max_per_page"
    total_images = min(len(images), max_per_page)

    # Create a subset of images for display
    subset_images = images[:total_images]
    subset_titles = titles[:total_images]

    # Dynamically calculate rows for this subset
    rows = (total_images + cols - 1) // cols

    # Create the figure
    fig, axes = plt.subplots(rows, cols, figsize=figsize)
    axes = axes.flatten()

    # Display each image in the grid
    for i, img in enumerate(subset_images):
        ax = axes[i]
        ax.imshow(img, cmap="gray" if len(img.shape) == 2 else None)
        ax.set_title(subset_titles[i], size=12)
        ax.axis("off")

    # Turn off unused axes if there are fewer than cols * rows images
    for i in range(total_images, len(axes)):
        axes[i].axis("off")

    # Adjust layout and display
    plt.tight_layout()
    plt.show()
```

1.3 Our prepossessing functions

1.3.1 Convert to grayscale

Then, We implemented the **convert_to_grayscale** function to convert the passed list of images to grayscale.

```
def convert_to_grayscale(images):
    grayscale_images = [cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) for img in images]
    return grayscale_images
```

1.3.2 Blurring

Next, we made **remove_salt_pepper** function that apply 2 filters, vertical blur and median filter, both help with minimizing the noises like salt and pepper noise.

```
    def remove_salt_pepper(images):
    filtered_images = []
    for img in images:
        vertical_blur = cv2.blur(img, (1, 7))  # Vertical blur
        filtered_img = cv2.medianBlur(vertical_blur, 3)  # Median filter
        filtered_images.append(filtered_img)
    return filtered_images
```

1.3.3 Sobel gradients

Moreover, we have a function named **calculate_gradients** that calculate sobel gradients for the received grayscaled images.

```
def calculate_gradients(grayscale_images, ksize=3):
    gradient_images = []
    for img in grayscale_images:
        grad_x = cv2.Sobel(img, cv2.CV_8UC1, 1, 0, ksize=ksize)
        grad_y = cv2.Sobel(img, cv2.CV_8UC1, 0, 1, ksize=ksize)
        gradient = cv2.subtract(grad_x, grad_y)
        gradient = cv2.convertScaleAbs(gradient)
        gradient_images.append(gradient)
    return gradient_images
```

1.3.4 Threshold using Otsu

Then we made the **threshold_images** function that apply threshold to the images using Otsu's method. What is Otsu? Well, The Otsu method uses the histogram of the image to find the optimal threshold value. It analyzes the histogram to minimize the variance within the foreground and background pixel groups.

```
def threshold_images(gradient_images):
    otsu_thresh_images = []
    for img in gradient_images:
        _, otsu_thresh = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        otsu_thresh_images.append(otsu_thresh)
    return otsu_thresh_images
```

1.3.5 Morphological operations

Then, we perform morphological operations for cleanup. We start be closing then erosion then dilation.

```
def clean_images(threshold_images):
    cleaned_images = []
```

```
    struct = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
    for img in threshold_images:
        closed_img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, struct)
        erosion_img = cv2.erode(closed_img, None, iterations=5)
        dilation_img = cv2.dilate(erosion_img, None, iterations=10)
        cleaned_images.append(dilation_img)
    return cleaned_images
```

1.3.6 High-pass filter

Then, we have high-pass filter that removes low-frequency components from the image by transforming the grayscale images to the frequency domain using FFT, applying a mask to block low frequencies, and then converting the image back to the spatial domain. We have **limit** as the radius of the high-pass filter mask, **gaussian** which is a boolean flag to apply Gaussian blur to the mask which its default is False and **keep_dc** which is a boolean flag to keep the DC component "AKA the center of the frequency domain" which its default is False.

```
    def try_highpass(dft_img, limit, gaussian: bool = False, keep_dc: bool = False):
    # List to store the filtered images in the spatial domain
    fft_filtered_images = []

    for gray_image in dft_img:
        # Step 1: # Apply FFT and shift low frequencies to the center
        dft_img_shift = np.fft.fftshift(np.fft.fft2(gray_image))

        # Step 2: Create the high-pass filter mask
        mask = ~give_me_circle_mask_nowww(dft_img_shift.shape, limit)

        # Optional Gaussian smoothing on the mask
        if gaussian:
            mask = cv2.GaussianBlur(mask, (21, 21), 0)

        # Optionally keep the DC component
        if keep_dc:
            mask[dft_img_shift.shape[0] // 2, dft_img_shift.shape[1] // 2] = 255

        # Step 3: Apply the mask to the shifted FFT of the image
        dft_img_shifted_highpass = np.multiply(dft_img_shift, mask)

        # Step 4: Inverse FFT to return to spatial domain
        spatial_img = np.abs(np.fft.ifft2(np.fft.ifftshift(dft_img_shifted_highpass)))

        # Step 5: Append the filtered image to the list
        fft_filtered_images.append(spatial_img)

    # Return the list of filtered images in the spatial domain
    return fft_filtered_images
```

We also have the **give_me_circle_mask_nowww** that creates a circular mask of a given radius. This mask is used to filter out low-frequency components in the frequency domain in High-pass filter step.

```
def give_me_circle_mask_nowww(mask_size, radius):
    # Ensure the mask is a single-channel grayscale image, with dtype=np.uint8
    mask = np.zeros(mask_size, dtype=np.uint8)  # Single channel mask

    # Find the center of the mask
    cy = mask.shape[0] // 2
    cx = mask.shape[1] // 2

    # Draw a filled white circle
    mask = cv2.circle(mask, (cx, cy), radius, 255, -1)
    # 255 is the white color in grayscale

    return mask
```

1.3.7 Threshold

Next, we used the **threshold_images1** function to manually apply a threshold value of 110 to the images after performing FFT.

```
def threshold_images1(fft_filtered_images_normalized):
    manual_thresh_img = []
    for img in fft_filtered_images_normalized:
        # Manual Thresholding
        _, manual_threshed = cv2.threshold(img, 110, 255, cv2.THRESH_BINARY)
        manual_thresh_img.append(manual_threshed)
    return manual_thresh_img
```

1.4 Detect and Straighten barcodes from images

We have **detect_barcodes_with_orientation_and_axis** function that detects the barcodes in a list of cleaned (After morphological operations) images and return the bounding box coordinates along with the final images.
We also have **detect_barcodes_after_rotation** function that repeat the previous prepossessing steps on the rotated images for better cropping.

```python
def detect_barcodes_with_orientation_and_axis(cleaned_images, original_images):
    final_images = []
    bounding_boxes = []
    orientations = []  # To store barcode orientation information

    for i, img in enumerate(cleaned_images):
        # Find contours in the cleaned image
        contours, _ = cv2.findContours(img.copy(), cv2.RETR_EXTERNAL,
                      cv2.CHAIN_APPROX_SIMPLE)

        # Check if any contours were found
        if not contours:
            print(f"No contours found in image {i+1}. Skipping this image.")
            final_images.append(original_images[i])
            bounding_boxes.append(None)
            orientations.append("No Barcode")
            continue

        # Sort contours by area and use the largest one
        contours = sorted(contours, key=cv2.contourArea, reverse=True)
        rotated_bounding_box = cv2.minAreaRect(contours[0])
        bounding_box_points = cv2.boxPoints(rotated_bounding_box)
        bounding_box_points = np.intp(bounding_box_points)

        # Calculate the orientation angle of the bounding box
        center = tuple(map(int, rotated_bounding_box[0]))  # Center of the bounding box
        width, height = rotated_bounding_box[1]
        angle = rotated_bounding_box[-1]

        # Correct the angle to straighten the barcode properly
        if width > height:
            angle = angle + 90 if angle < -45 else angle
        else:
            angle = angle - 90 if angle > 45 else angle

        # Determine the orientation of the barcode
        # Treat angles close to 0° or ±90° as straight
        if abs(angle) <= 5 or abs(angle) >= 85:
            orientation = "Straight"
            final_images.append(original_images[i])
```

```
            bounding_boxes.append(cv2.boundingRect(contours[0]))  # Get bounding box

        else:
            orientation = "Tilted"

            # Straighten the tilted image
            rotation_matrix = cv2.getRotationMatrix2D(center, angle, 1.0)
            rotated_img = cv2.warpAffine(
                original_images[i],
                rotation_matrix,
                (original_images[i].shape[1], original_images[i].shape[0]),
                borderValue=(255, 255, 255),
            )

            # Detect barcodes again on the rotated-cleaned image so that
            # the barcode is detected and cropped correctly
            rotated_final, rotated_bbox = detect_barcodes_after_rotation(rotated_img)
            display_images([rotated_final], [f"rotated_final {i+1}"], figsize=(25, 15))

            final_images.append(rotated_final)
            bounding_boxes.append(rotated_bbox)

        orientations.append(f"Orientation: {orientation}, Angle: {angle:.2f}°")

    return final_images, bounding_boxes, orientations

def detect_barcodes_after_rotation(rotated_image):

    # Convert the rotated image to grayscale
    rotated_grayscale = cv2.cvtColor(rotated_image, cv2.COLOR_BGR2GRAY)
    display_images([rotated_grayscale], [f"rotated_grayscale"], figsize=(25, 15))

    # Apply salt and pepper noise removal
    rotated_filtered = remove_salt_pepper([rotated_grayscale])[0]
    display_images([rotated_filtered], [f"rotated_filtered"], figsize=(25, 15))

    # Calculate gradients to highlight edges
    rotated_gradient = calculate_gradients([rotated_filtered])[0]
    display_images([rotated_gradient], [f"rotated_gradient"], figsize=(25, 15))

    # Apply thresholding to create a binary image
    _, rotated_threshold = cv2.threshold(rotated_gradient, 0, 255,
                           cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    display_images([rotated_threshold], [f"rotated_threshold"], figsize=(25, 15))

    # Clean the thresholded image using morphological operations to highlight
    #the barcode boarders
```

```
    rotated_cleaned = clean_images([rotated_threshold])[0]
    display_images([rotated_cleaned], [f"rotated_cleaned"], figsize=(25, 15))

    # Find contours in the cleaned image
    contours, _ = cv2.findContours(rotated_cleaned.copy(),
                  cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Check if any contours were found
    if not contours:
        print("No contours found in the rotated image.")
        return rotated_image, None

    # Sort contours by area and get the largest one assuming it is the barcode
    contours = sorted(contours, key=cv2.contourArea, reverse=True)
    rotated_bounding_box = cv2.minAreaRect(contours[0])
    bounding_box_points = cv2.boxPoints(rotated_bounding_box)
    bounding_box_points = np.intp(bounding_box_points)

    # Return the rotated image with bounding box and the bounding box coordinates
    return rotated_image, rotated_bounding_box
```

1.5 Crop barcodes

The **crop_barcodes** function crops the barcodes from the original images using the given bounding box coordinates. We handle the two cases whether the barcode is straight or rotated by checking the format of the bounding box format.

```python
def crop_barcodes(original_images, bounding_boxes):
    cropped_barcodes = []

    for image, bbox in zip(original_images, bounding_boxes):
        # Check if the bounding box is in the format
        (x, y, w, h)  "AKA straight image"
        if len(bbox) == 4:
            x, y, w, h = bbox
            cropped_barcode = image[y:y+h, x:x+w]
            cropped_barcodes.append(cropped_barcode)

        # Check if the bounding box is in the format
        # ((center), (width, height), angle)  "AKA the rotated image"
        elif len(bbox) == 3:
            center, size, angle = bbox
            # Get the rotated rectangle as 4 points
            box_points = cv2.boxPoints(((center), size, angle))
            box_points = np.int32(box_points)  # Convert to integer

            # Get the bounding rectangle of the box
            x, y, w, h = cv2.boundingRect(box_points)

            # Crop using the new bounding box
            cropped_barcode = image[y:y+h, x:x+w]
            cropped_barcodes.append(cropped_barcode)

        else:
            print(f"Skipping invalid bounding box: {bbox}")
            continue

    return cropped_barcodes
```

1.6 Detect barcode from images main function

Now we have the main function **detect_barcode_from_images** that calls almost all the previous functions. It have the flow of our prepossessing of the images, the following steps is applied to the images:

1. Load images from the provided paths.
2. Convert images to grayscale.
3. Remove salt-and-pepper noise from grayscale images.

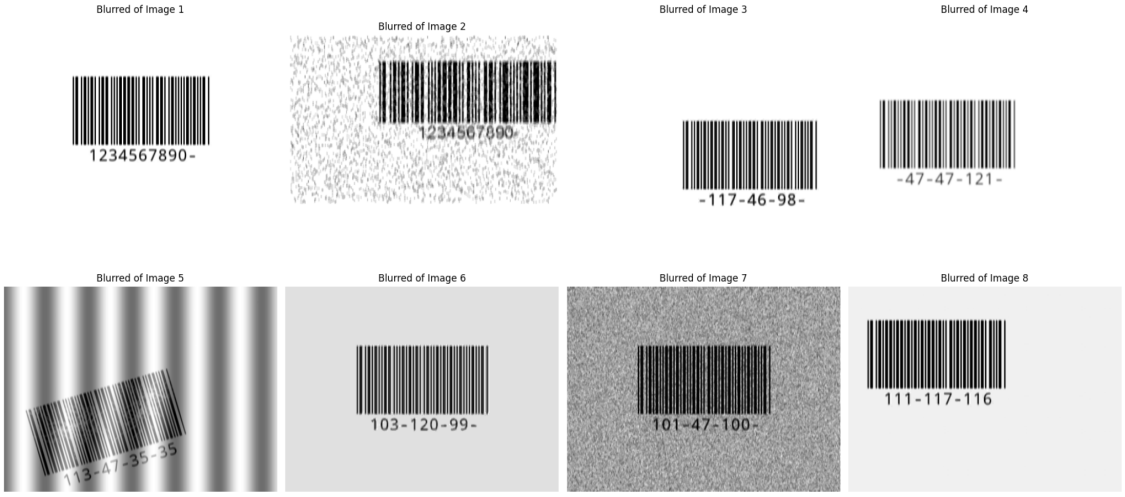

Figure 1: Our blur Results

4. Calculate gradients to highlight edges.

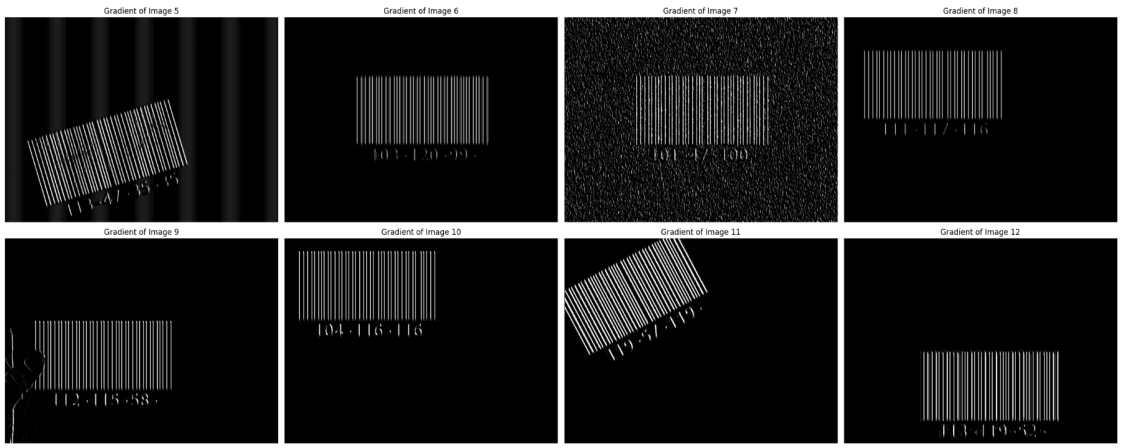

Figure 2: Our gradients Results

5. Apply thresholding to the gradient images.

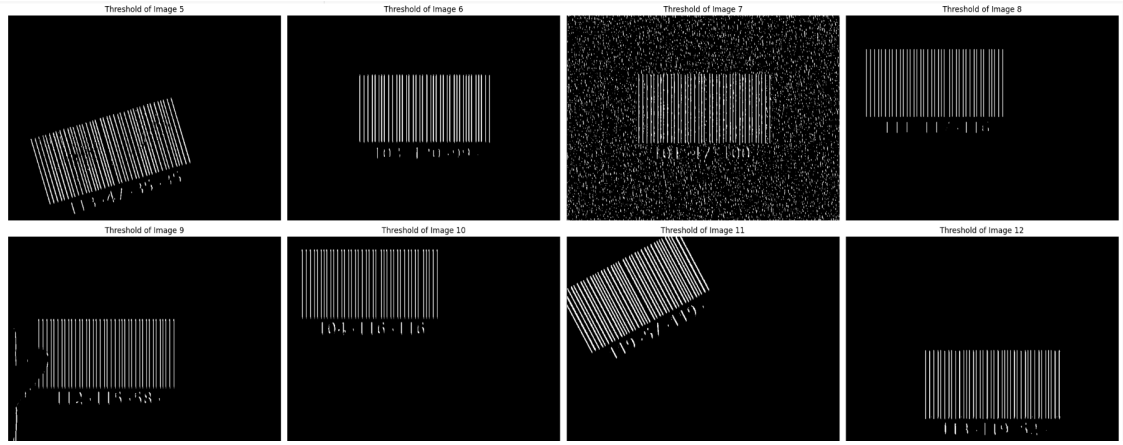

Figure 3: Our threshold Results

6. Remove salt-and-pepper noise again after thresholding.

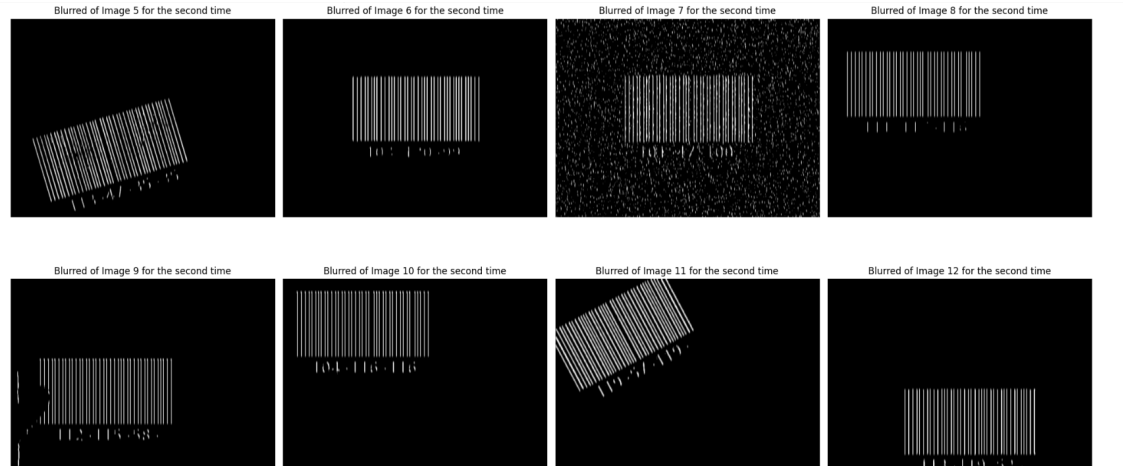

Figure 4: Our removing salt-and-pepper noise for the second time Results

7. Clean the images to remove small noise (Apply Morphological operations).

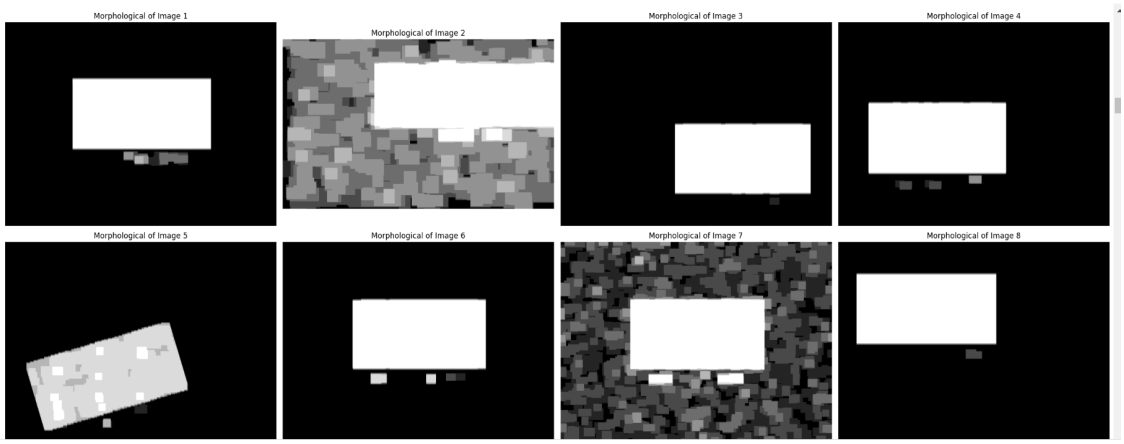

Figure 5: Our Morphological Results

8. Apply thresholding again to the cleaned images.

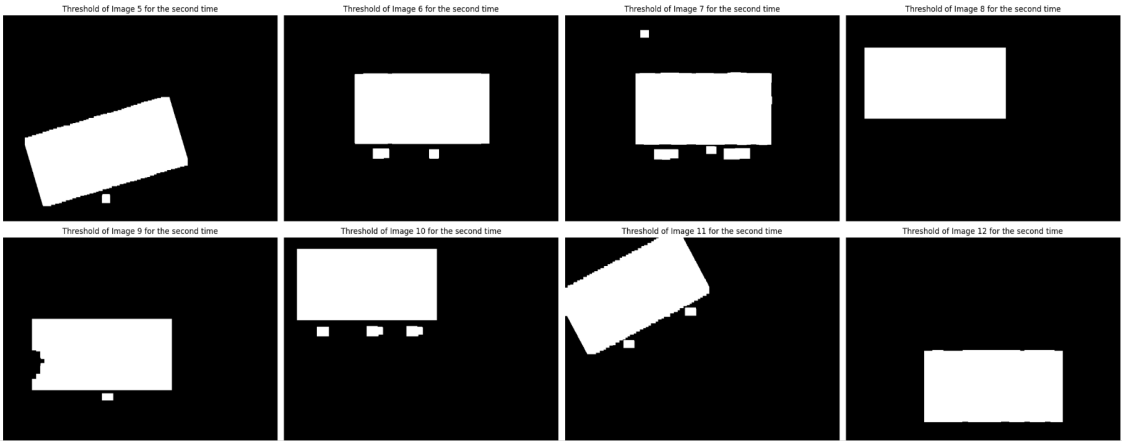

Figure 6: Our threshold for the second time Results

9. Detect barcodes and get bounding boxes from the thresholded images.
10. Convert the final images with detected barcodes to grayscale for FFT filtering

11. Remove salt-and-pepper noise from the grayscale images.

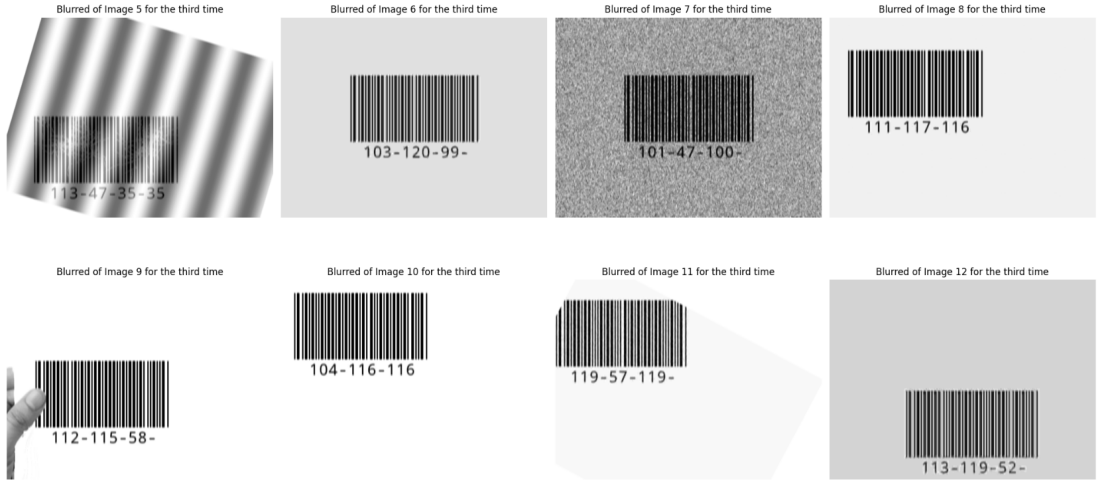

Figure 7: Our removing salt-and-pepper noise for the third and final time Results

12. Apply FFT filtering to each cropped barcode image.

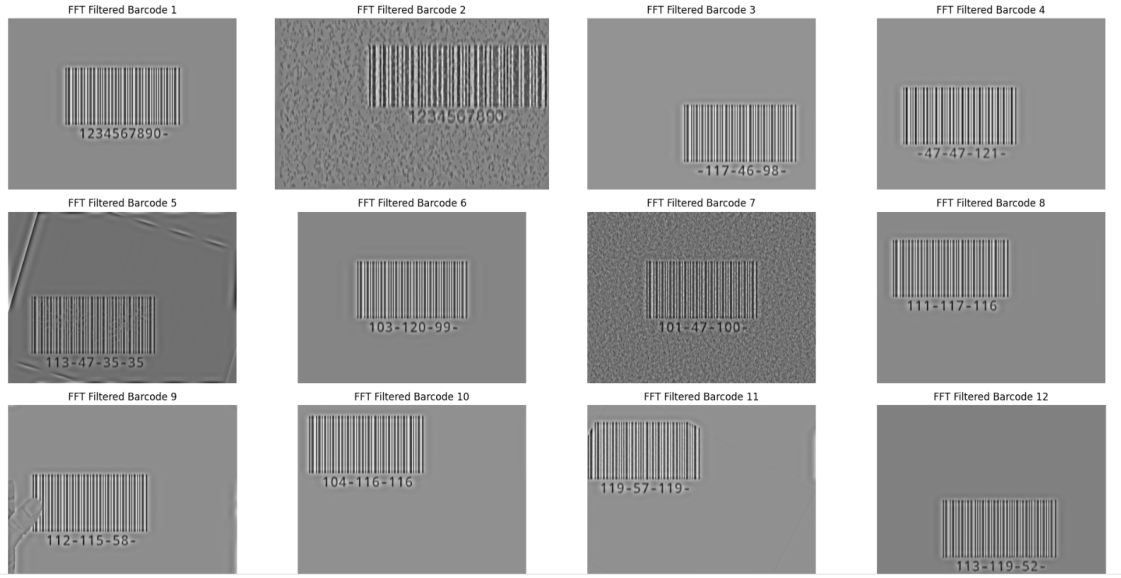

Figure 8: Our High-pass filter Results

13. Normalize and convert FFT-filtered images to uint8 before thresholding.

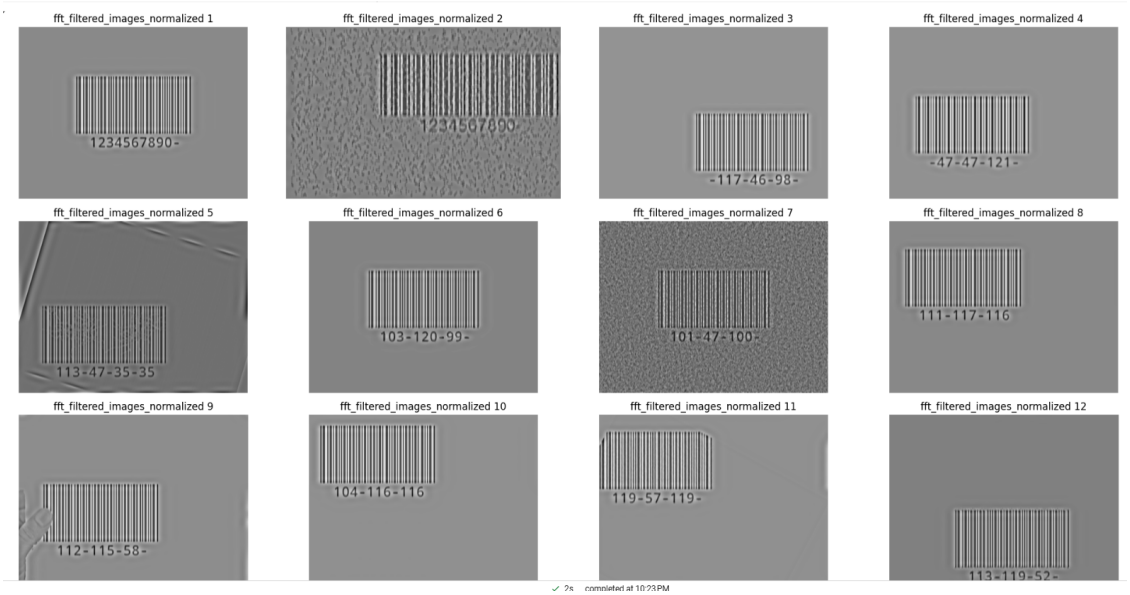

Figure 9: Our Normalized images Results

14. Apply thresholding to the normalized FFT-filtered images.

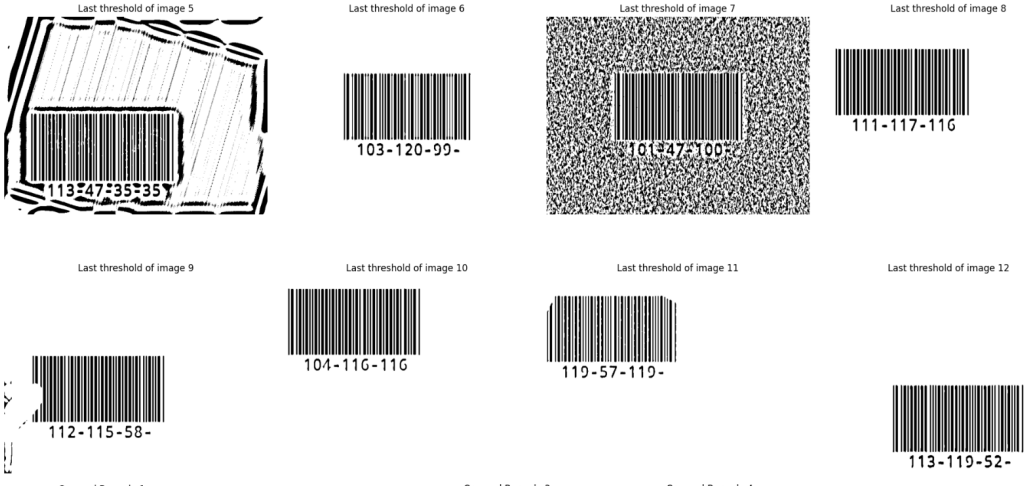

Figure 10: Our threshold for the third time Results

15. Crop barcodes using the bounding box coordinates.

16. Display the cropped barcode images.

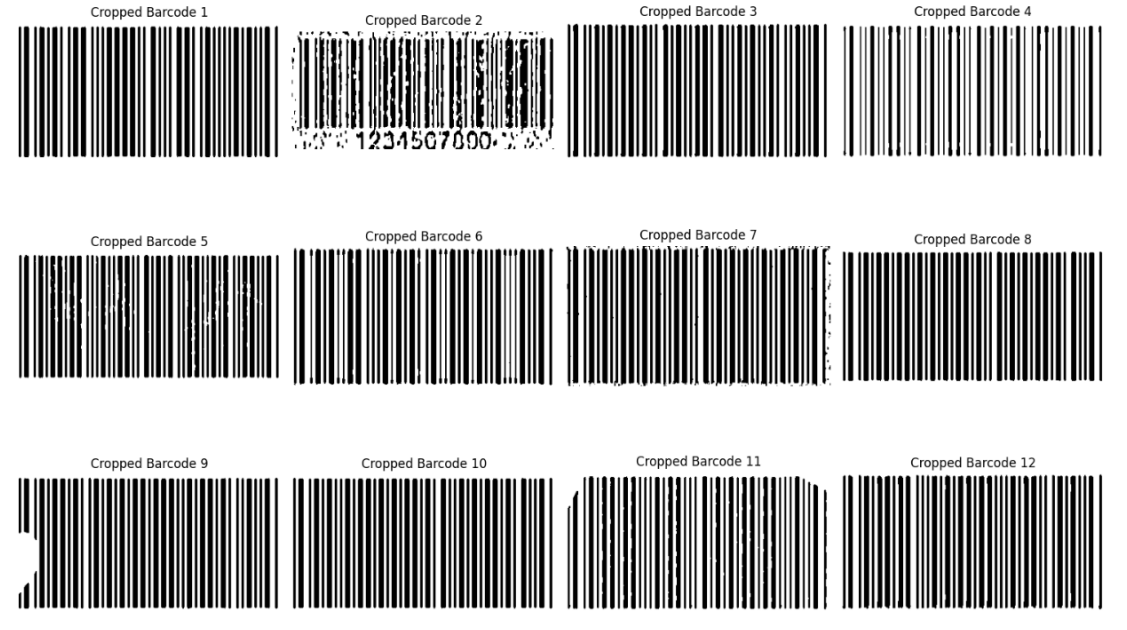

Figure 11: Our final cropped imaged

```python
def detect_barcode_from_images(image_paths, fft_limit=20, fft_gaussian=True, fft_keep_dc=False):
    # Load images
    images = load_images(image_paths)

    # Convert to grayscale
    grayscale_images = convert_to_grayscale(images)

    # Remove salt-and-pepper noise
    filtered_images = remove_salt_pepper(grayscale_images)
    display_images(filtered_images, [f"Blurred of Image {i+1}" for i in range(len(filtered_images))], figsize=(20, 15))

    # Calculate gradients
    gradient_images = calculate_gradients(filtered_images)
    display_images(gradient_images, [f"Gradient of Image {i+1}" for i in range(len(gradient_images))], figsize=(25, 15))

    # Threshold images
    threshold_images_set = threshold_images(gradient_images)
    display_images(threshold_images_set, [f"Threshold of Image {i+1}" for i in range(len(threshold_images_set))], figsize=(25, 15))

    # Remove salt-and-pepper noise agian
    filtered_images2 = remove_salt_pepper(threshold_images_set)
    display_images(filtered_images2, [f"Blurred of Image {i+1} for the second time" for i in range(len(filtered_images2))], figsize=(20, 15))

    # Morphological of image
    morph_images = clean_images(filtered_images2)
    display_images(morph_images, [f"Morphological of Image {i+1}" for i in range(len(morph_images))], figsize=(25, 15))

    # Threshold images again
    threshold_images_set2 = threshold_images(morph_images)
    display_images(threshold_images_set2, [f"Threshold of Image {i+1} for the second time" for i in range(len(threshold_images_set2))], figsize=(25, 15))

    # Detect barcodes and get bounding boxes
    final_images, bounding_boxes, _ = detect_barcodes_with_orientation_and_axis(threshold_images_set2, images)
    display_images(final_images, [f"Detected Barcodes {i+1}" for i in range(len(final_images))], figsize=(20, 15))

    # Convert detected barcodes to grayscale for FFT filtering
    grayscale_images2 = convert_to_grayscale(final_images)
    display_images(grayscale_images2, [f"grayscale for FFT filtering {i+1}" for i in range(len(grayscale_images2))], figsize=(20, 15))

    # remove salt and pepper
    filtered_images3 = remove_salt_pepper(grayscale_images2)
    display_images(filtered_images3, [f"Blurred of Image {i+1} for the third time" for i in range(len(filtered_images3))], figsize=(20, 15))

    # Apply FFT filtering for each cropped barcode
    fft_filtered_images = try_highpass(filtered_images3, fft_limit, fft_gaussian, fft_keep_dc)

    # Display FFT-filtered results
    fft_titles = [f"FFT Filtered Barcode {i+1}" for i in range(len(fft_filtered_images))]
    display_images(fft_filtered_images, fft_titles, cols=4, figsize=(20, 10))
```

Figure 12: detect_barcode_from_images

```
    # remove salt and pepper
    filtered_images3 = remove_salt_pepper(grayscale_images2)
    display_images(filtered_images3, [f"Blurred of Image {i+1} for the third time" for i in range(len(filtered_images3))], figsize=(20, 15))

    # Apply FFT filtering for each cropped barcode
    fft_filtered_images = try_highpass(filtered_images3, fft_limit, fft_gaussian, fft_keep_dc)

    # Display FFT-filtered results
    fft_titles = [f"FFT Filtered Barcode {i+1}" for i in range(len(fft_filtered_images))]
    display_images(fft_filtered_images, fft_titles, cols=4, figsize=(20, 10))

    # Normalize and convert FFT-filtered images to uint8 before thresholding
    fft_filtered_images_normalized = [
        cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
        for img in fft_filtered_images
    ]
    filtered_images_titles = [
        f"fft_filtered_images_normalized {i+1}" for i in range(len(fft_filtered_images_normalized))
    ]
    display_images(fft_filtered_images_normalized, filtered_images_titles, cols=4, figsize=(20, 10))

    # Pass the normalized images to the thresholding function
    threshold_images_set3 = threshold_images1(fft_filtered_images_normalized)
    display_images(threshold_images_set3, [f"Last threshold of image {i+1}" for i in range(len(threshold_images_set3))],cols=4, figsize=(20, 15))

    # Crop barcodes using bounding box coordinates
    cropped_barcodes = crop_barcodes(threshold_images_set3, bounding_boxes)
    cropped_titles = [f"Cropped Barcode {i+1}" if crop is not None else f"No Barcode {i+1}" for i, crop in enumerate(cropped_barcodes)]
    display_images([crop for crop in cropped_barcodes if crop is not None], cropped_titles)

    return cropped_barcodes


# Start detecting barcodes from images
preprocessed = detect_barcode_from_images(image_paths, fft_limit=20, fft_gaussian=True, fft_keep_dc=True)
```

Figure 13: detect_barcode_from_images

2 Our Final Result

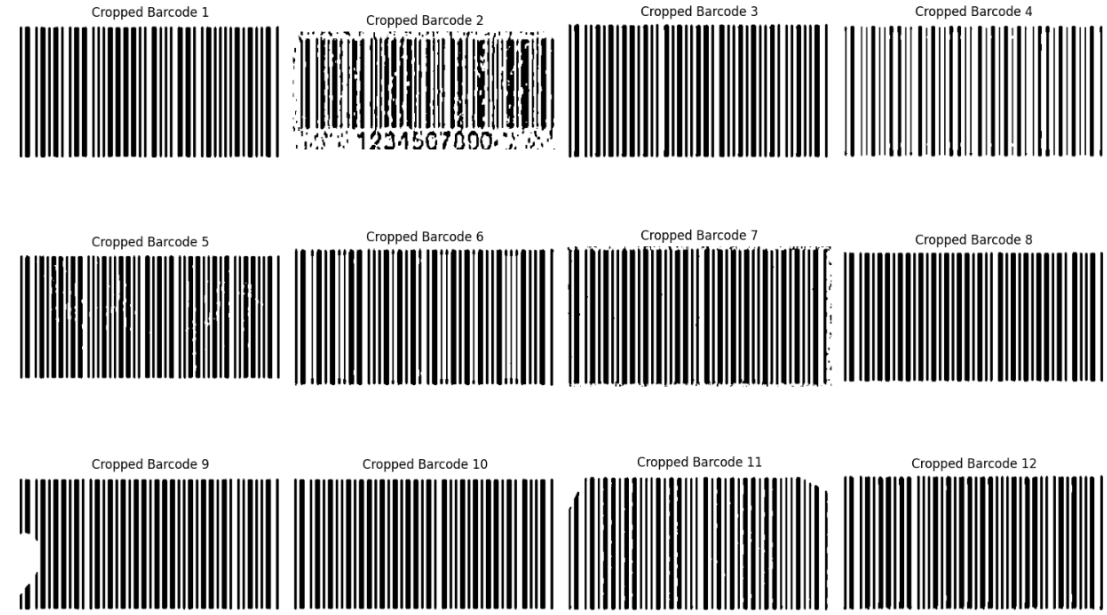

Figure 14: Our final Results

1 Our Decoder:

Before we dive into the structure of the decoder, we introduced the code 11 symbology

```
# Code 11 encoding table
code11_encoding = {
    "00110": "Start/Stop",
    "10001": "1",
    "01001": "2",
    "11000": "3",
    "00101": "4",
    "10100": "5",
    "01100": "6",
    "00011": "7",
    "10010": "8",
    "10000": "9",
    "00001": "0",
    "00100": "-"
}
```

1.1 Detection of Widths

The first method in our code is **detect_bar_widths()**. This method takes an image and converts it into a 2D array. It calculates the mean of column values and according to the result, it determines if this column is a black bar (¿ 128) or wide space (¡ 128). It loops on the columns and checks if we are in the same bar or space to determine its width. when it is not the same, it appends the width and starts to calculate the next width. It returns an array of these widths after reaching the last column in the image.

```
def detect_bar_widths(image):

    num_cols = image.shape[1]  # Get the number of columns in the image
    bar_space_widths = []
    current_width = 1
    # Start by checking if the first column is part of a bar (black)
    is_bar = image[:, 0].mean() < 128

    # Loop through each column of the image to measure widths
    for col in range(1, num_cols):
        # Determine if the current column is a bar or space
        column_is_bar = image[:, col].mean() < 128

        if column_is_bar == is_bar:
```

```
            # Continue counting the width if it's the same type (bar or space)
            current_width += 1
        else:
            # Store the completed width and reset for the next segment
            bar_space_widths.append(current_width)
            current_width = 1
            is_bar = column_is_bar  # Toggle between bar and space

    # Append the width of the final segment
    bar_space_widths.append(current_width)
    return bar_space_widths
```

1.2 Classifying the Widths

The second method is **classify_widths_histogram()**. This method takes the list of widths and computes a histogram of these widths. It then determines the narrow and wide widths according to the first and second peaks in the histogram. It returns these two values.

```
def classify_widths_histogram(bar_space_widths):

    # Compute a histogram of the bar and space widths
    histogram, bin_edges = np.histogram(bar_space_widths, bins='auto')

    # Find the indices of the two most prominent peaks in the histogram
    peak_indices = np.argsort(histogram)[-2:]
    peaks = sorted(bin_edges[peak_indices])

    # Check if two distinct peaks were identified
    if len(peaks) < 2:
        raise ValueError("Unable to determine narrow and wide widths from histogram.")

    # Assign the narrow and wide widths based on the identified peaks
    narrow, wide = peaks
    return int(round(narrow)), int(round(wide))

```

1.3 Decoding the Widths

The Third function is **decode_barcode()**. This is the function responsible for the main functionality of the decoding. It takes the list of widths, and computes narrow and wide. We set a tolerance to handle the noise, and then convert the widths into binary patterns. Then, it looks for the start symbol and star to decode every 5 bits according to code 11 table and skip a space after every 5 bits. If it is unable to decode a pattern in between it appends

an error message. It keeps decoding until it encounters the stop sample. It returns the final decoding.

```
def decode_barcode(bar_space_widths):
    # Call function classify_widths_histogram to determine the narrow and wide bar/space
    try:
        narrow, wide = classify_widths_histogram(bar_space_widths)
    except ValueError:
        return "Error: Histogram analysis failed."

    # Set tolerances
    tolerance_narrow = narrow * 0.5
    tolerance_wide = wide * 0.5

    # Convert widths to binary pattern
    binary_pattern = ""
    for width in bar_space_widths:
        if abs(width - narrow) <= tolerance_narrow:
            binary_pattern += "0"
        elif abs(width - wide) <= tolerance_wide:
            binary_pattern += "1"
        elif abs(width) >= wide:
            binary_pattern += "1"
        elif abs(width) <= narrow:
            binary_pattern += "0"
        else:
            return f"Error: Width {width} not matching narrow ({narrow}) or wide ({wide})."

    # Process the binary pattern by taking 5 bits at a time, skipping one bit after each chunk
    decoded_message = []
    current_pattern = ""
    started = False  # Flag to start decoding
    i = 1  # Start at the second bit (skip the very first bit) due to the noise

    while i + 4 < len(binary_pattern):  # Ensure we have 5 bits to process
        # Take the next 5 bits
        current_pattern = binary_pattern[i:i+5]

        if current_pattern == "00110" and not started:  # Start pattern
            started = True  # Begin decoding after "Start"
            i += 6  # Skip the first 5 bits "Start" + the space bit after it
            continue

        elif started:
            if current_pattern in code11_encoding:
```

```
                decoded_symbol = code11_encoding[current_pattern]
                if decoded_symbol != "Start/Stop":  # Ignore Start/Stop symbol
                    decoded_message.append(decoded_symbol)
            else:
                # Append error message but continue decoding
                decoded_message.append(f"Error: Invalid pattern {current_pattern}")

        i += 6  # Skip the next bit after processing 5 bits and take the next 5 bits

    if not decoded_message:
        return "Error: Unable to decode any characters."

    # Return the decoded message without Start/Stop bits
    return "".join(decoded_message)
```

1.4 Decoding Our Barcodes

These two functions are to process the barcode images and decode them. The first function is process_barcode_image(), which takes a single image. It first detects the widths of bars and spaces and then decodes it. After that, it returns the result of decoding. The second function is decode_multiple_barcodes(), which is just to decode multiple images and handling any errors if there is any.

```
def process_barcode_image(cropped_barcode):

    # Step 1: Detect bar and space widths
    bar_space_widths = detect_bar_widths(cropped_barcode)

    # Step 2: Decode the barcode
    decoded_message = decode_barcode(bar_space_widths)

    return decoded_message

def decode_multiple_barcodes(cropped_barcodes):

    decoded_results = []

    for i, cropped_image in enumerate(cropped_barcodes):
        if cropped_image is None:
            print(f"Image {i + 1}: No barcode detected.")
            decoded_results.append(None)
            continue

        try:
```

```
            print(f"Decoding barcode {i + 1}...")
            # Directly pass the preprocessed image to the function
            decoded_message = process_barcode_image(cropped_image)
            # Append the decoded message for each image
            decoded_results.append(decoded_message)
        except Exception as e:
            print(f"Error decoding barcode {i + 1}: {e}")
            decoded_results.append(None)

    # Return the list of decoded results, not just a single variable
    # as well as the cropped barcodes for displaying
    return decoded_results, cropped_barcodes


```

---

2 Our Results:

These are the results of our test cases after going through our decoder.

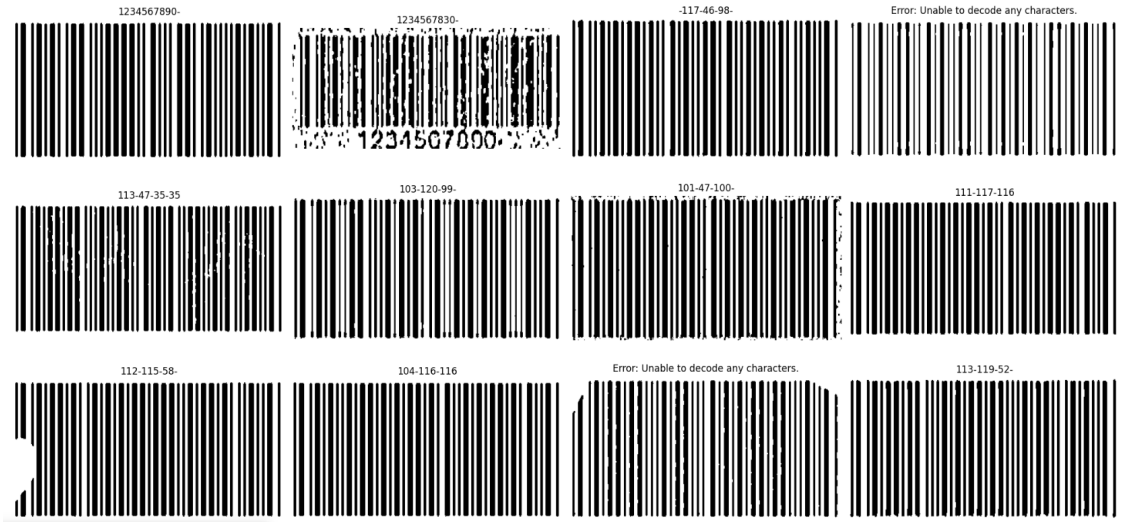

Figure 1: Our Results