Ain Shams University – Faculty of Engineering

Computer Engineering and Software Systems Program

CSE483 - Computer Vision

Barcode Extraction

Milestone 1

[Barcode-Scanner Github Repo]

[Project Discussion Video]

Name	ID
Youssef Adel Albert	21P0258
Mark Saleh Sobhi	21P0206
Ahmed Tarek Mahmoud	2100561

Contents

Introduction and overview 3

Proposed Solution 3

Function Overview 4

Generic Preprocessing of captured image 5

Function Overview 6

Image Copy and Conversion 6

Color Detection and Removal 7

Morphological Operations 8

Noise Reduction 8

Sinusoidal Wave Noise Detection and Removal 9

Find barcode contours 9

Function Overview 10

Preprocessing the Image 10

Binarization and Contour Detection 11

Drawing and Returning the Bounding Box 12

Handling No Contours Found 12

Cropping and straightening the image 12

Perspective Transformation 12

Orientation Correction 13

Binarization 13

Cleaning the Barcode (Postprocessing) 14

Return 16

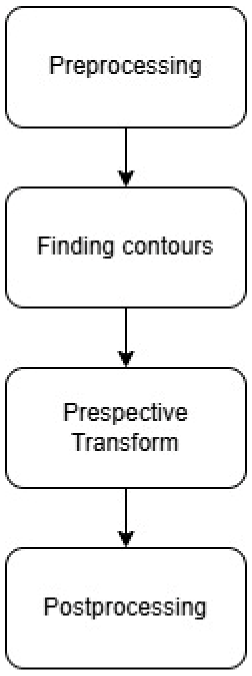
Introduction and overview

This project aims to incorporate the material discussed in the course in a practical application. The application is to apply classical computer vision algorithms to captured image of a barcode to automatically scan the bars on the label emulating the behaviour of real-life barcode scanners. By preprocessing the captured barcode label image, an undistorted, noiseless, binarized (grayscale), clear barcode be obtained and decoded as shown in the linked guide above.

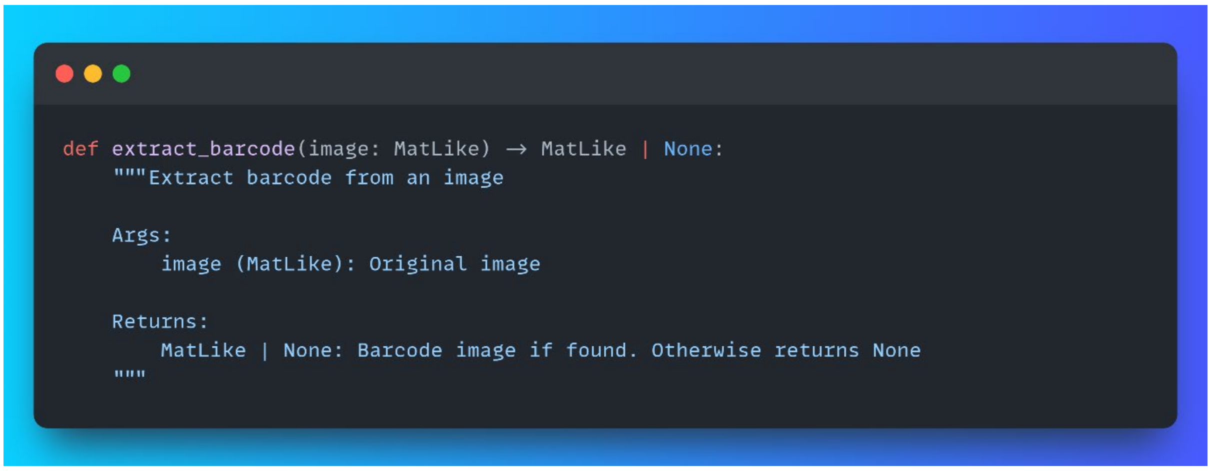
It aims to expand the knowledge of students, and how they can apply their obtained knowledge on basics of classical computer vision to implement a real-life standard through simple logic; in this case that is to decode the position of pixels (representing the bars of the barcode) and their varying widths to decode the data represented in the barcode. You should not attempt to read the text representation of the data at the bottom of the barcode with machine/deep learning and should not use any external libraries other than those discussed in the course labs.

Your final pipeline needs to be generic and robust enough, capable of passing as many test cases as possible without requiring any modifications or fine-tuning on-the-spot. Your code should only try to figure out the problem(s) in the current test case (e.g., salt-and-pepper noise, rotated barcode, etc.) and apply the appropriate fix(es) when applicable.

Proposed Solution



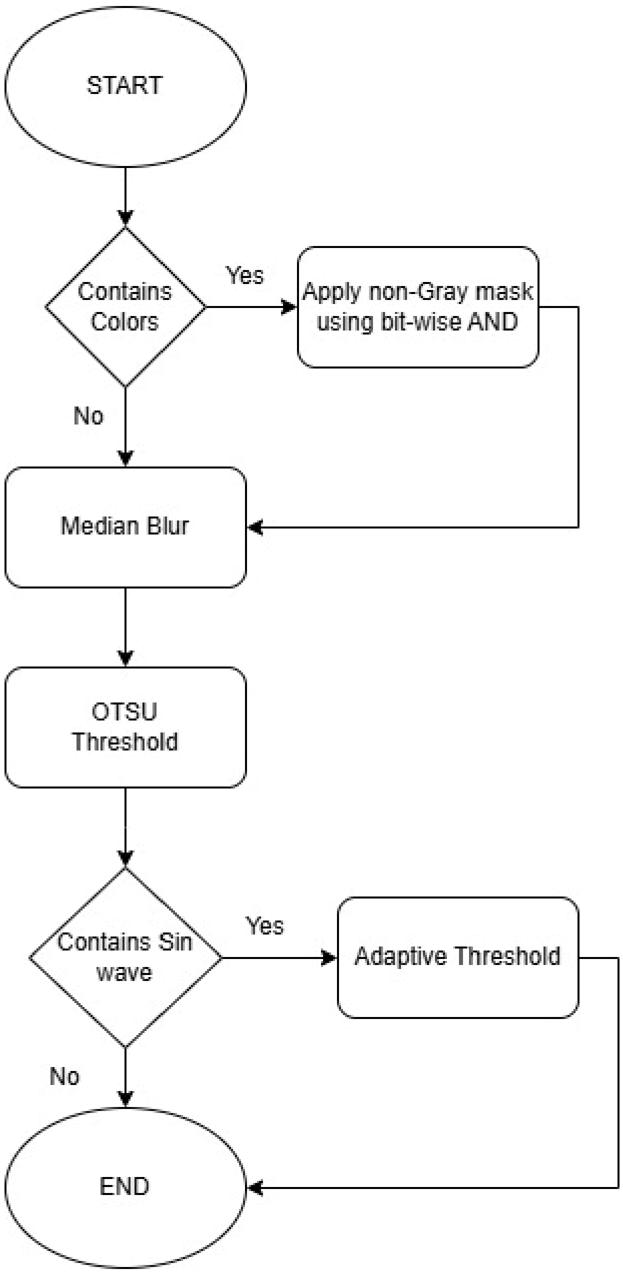
Function Overview

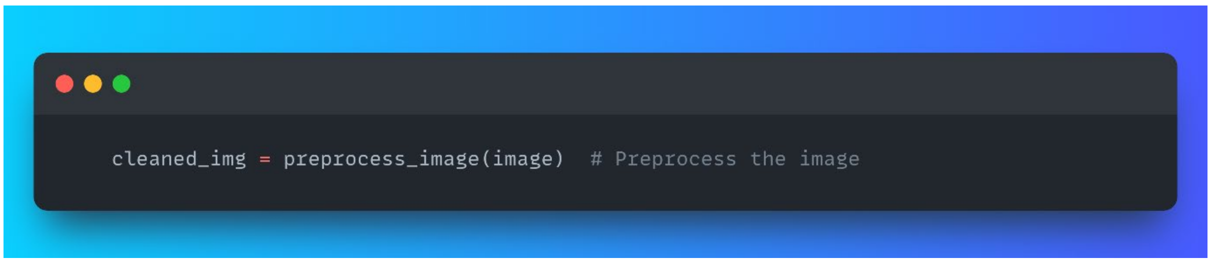


```python
def extract_barcode(image: MatLike) → MatLike | None:
    """Extract barcode from an image

    Args:
        image (MatLike): Original image

    Returns:
        MatLike | None: Barcode image if found. Otherwise returns None
    """
```

The extract_barcode function is designed to extract a barcode from an input image. It preprocesses the image, identifies the barcode's contours, applies a perspective transformation to isolate the barcode, and then cleans the barcode image for optimal decoding. The function takes a MatLike object (the original image) as input and returns a MatLike object containing the extracted barcode if found. If no barcode is detected, the function returns None.

Generic Preprocessing of captured image

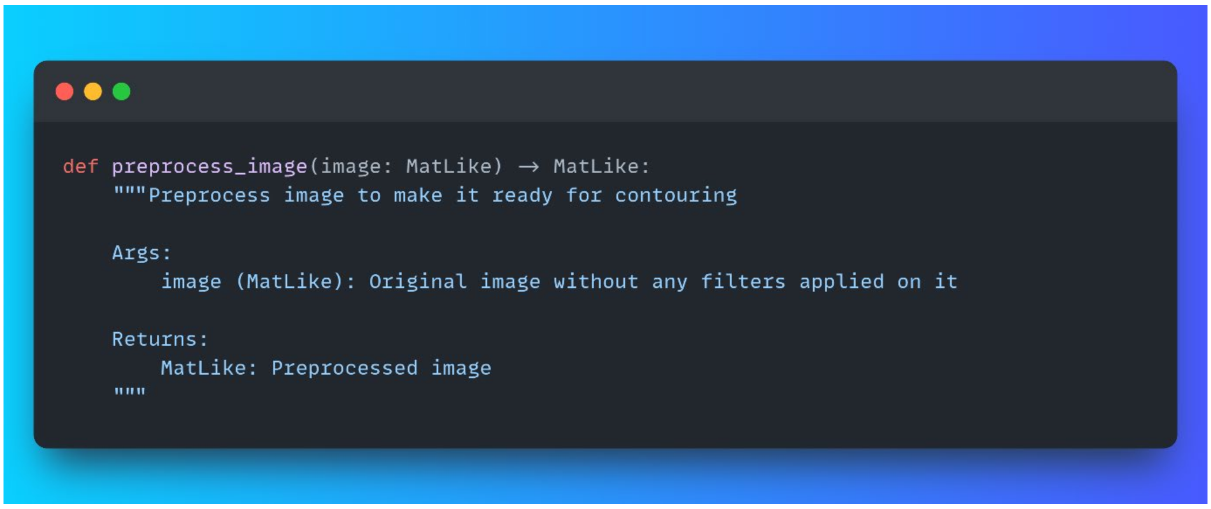




```
cleaned_img = preprocess_image(image)  # Preprocess the image
```

The function begins by preprocessing the input image using the preprocess_image function. This preprocessing step typically involves converting the image to grayscale, reducing noise, and enhancing contrast to make the barcode more distinguishable.

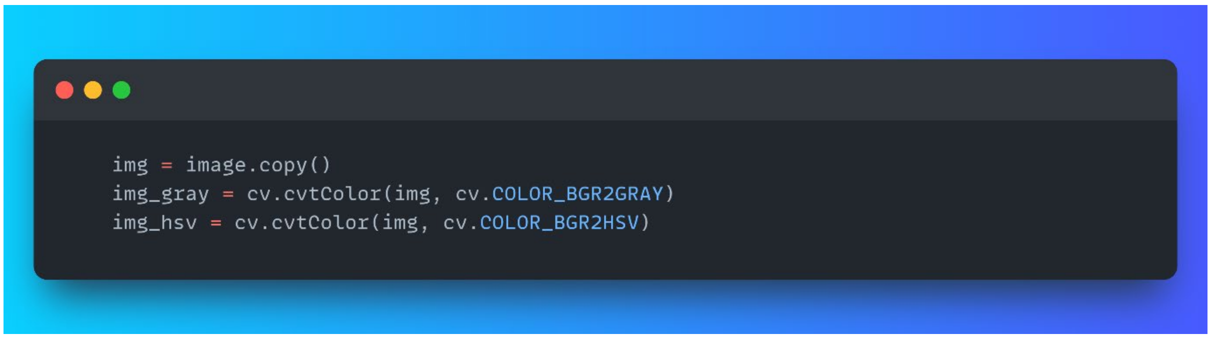
Function Overview



```python
def preprocess_image(image: MatLike) -> MatLike:
    """Preprocess image to make it ready for contouring

    Args:
        image (MatLike): Original image without any filters applied on it

    Returns:
        MatLike: Preprocessed image
    """
```

The preprocess_image function is designed to prepare an image for contour detection by addressing color content, noise, and distortions that may affect barcode recognition. The function identifies whether the image contains colors, removes them if necessary, reduces noise, and handles specific distortions such as sinusoidal wave noise. Below is a detailed breakdown of the preprocessing steps. The function takes an image in its original form without filters applied and returns a binary (black-and-white) image optimized for contour detection.

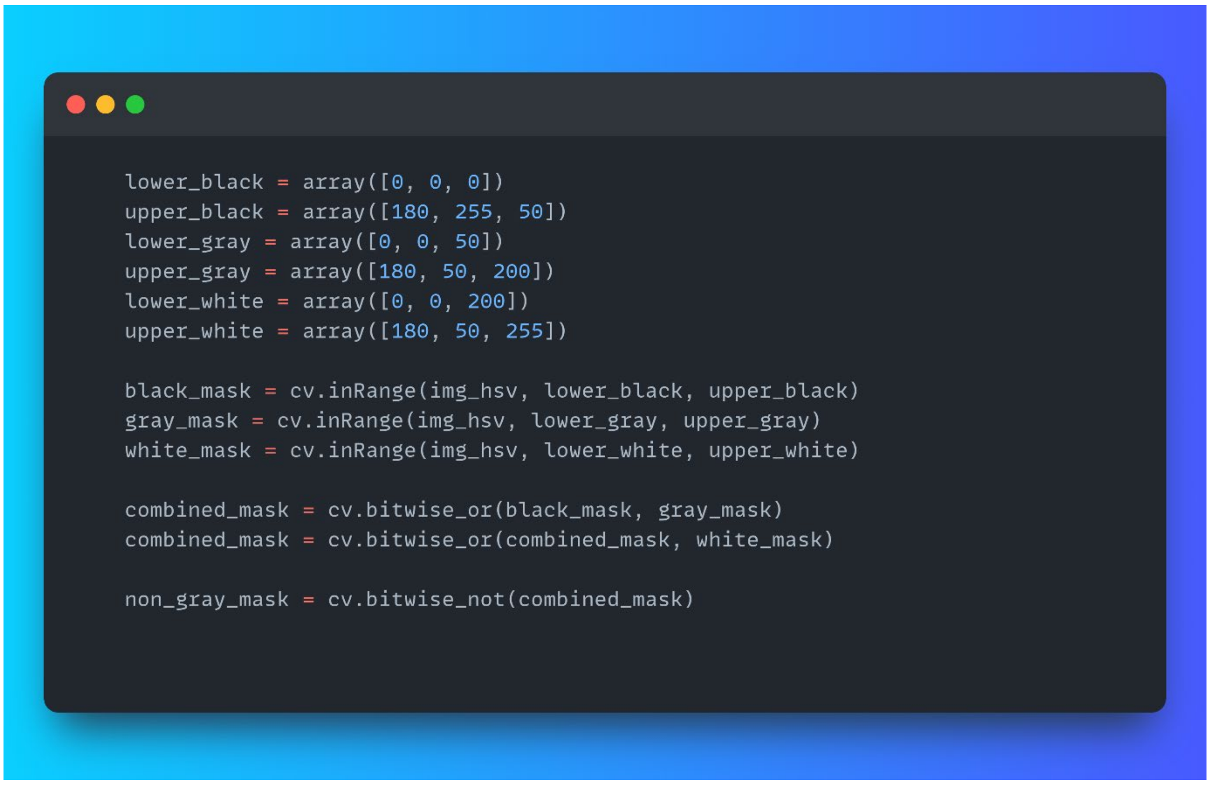
Image Copy and Conversion



```python
    img = image.copy()
    img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    img_hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
```

- The input image is copied to avoid modifying the original image.
- The copied image is converted to grayscale (img_gray) and HSV (img_hsv) color spaces for subsequent processing.

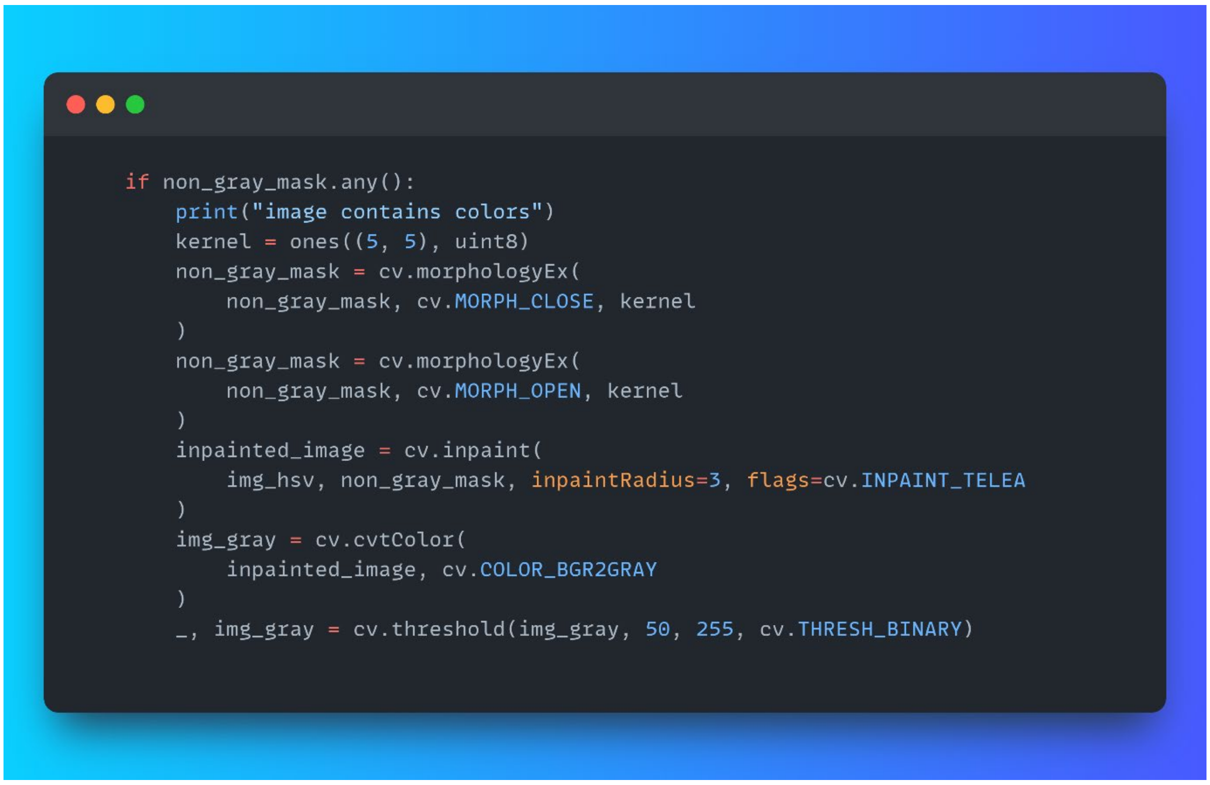
Color Detection and Removal



```
lower_black = array([0, 0, 0])
upper_black = array([180, 255, 50])
lower_gray = array([0, 0, 50])
upper_gray = array([180, 50, 200])
lower_white = array([0, 0, 200])
upper_white = array([180, 50, 255])

black_mask = cv.inRange(img_hsv, lower_black, upper_black)
gray_mask = cv.inRange(img_hsv, lower_gray, upper_gray)
white_mask = cv.inRange(img_hsv, lower_white, upper_white)

combined_mask = cv.bitwise_or(black_mask, gray_mask)
combined_mask = cv.bitwise_or(combined_mask, white_mask)

non_gray_mask = cv.bitwise_not(combined_mask)
```

- Purpose: Identify and remove non-gray colors (e.g., colored artifacts) to simplify the image.
- Color Ranges in HSV:
  - Black: Pixels with low saturation and brightness.
  - Gray: Pixels with moderate brightness and low saturation.
  - White: Pixels with high brightness and low saturation.
- Masks:
  - Create masks for black, gray, and white regions.
  - Combine these masks to form a single mask for all gray-scale regions.
  - Invert the combined mask to isolate non-gray colors.

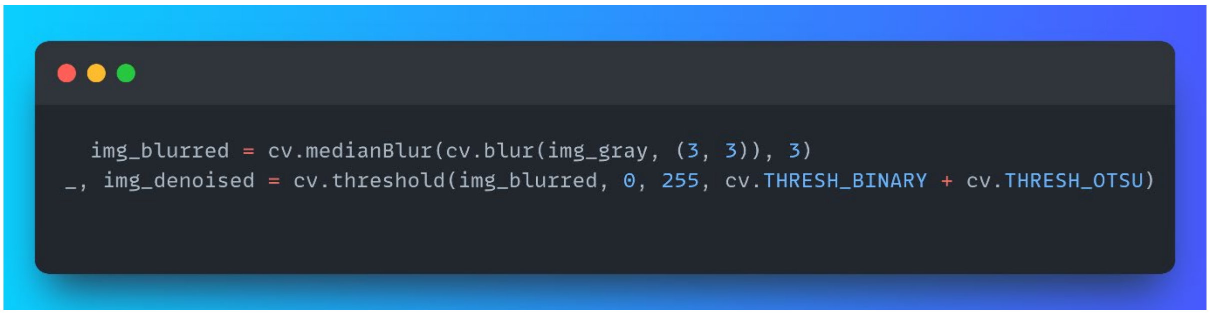
Morphological Operations



```
if non_gray_mask.any():
    print("image contains colors")
    kernel = ones((5, 5), uint8)
    non_gray_mask = cv.morphologyEx(
        non_gray_mask, cv.MORPH_CLOSE, kernel
    )
    non_gray_mask = cv.morphologyEx(
        non_gray_mask, cv.MORPH_OPEN, kernel
    )
    inpainted_image = cv.inpaint(
        img_hsv, non_gray_mask, inpaintRadius=3, flags=cv.INPAINT_TELEA
    )
    img_gray = cv.cvtColor(
        inpainted_image, cv.COLOR_BGR2GRAY
    )
    _, img_gray = cv.threshold(img_gray, 50, 255, cv.THRESH_BINARY)
```

- If the image contains colors, morphological closing and opening operations are applied to refine the mask by:
  - Filling small holes in detected regions.
  - Removing noise in the mask background.
- **Inpainting**:
  - Non-gray regions are removed by inpainting, filling those areas with surrounding pixel values.
  - The resulting image is converted back to grayscale and thresholded to a binary form.

Noise Reduction

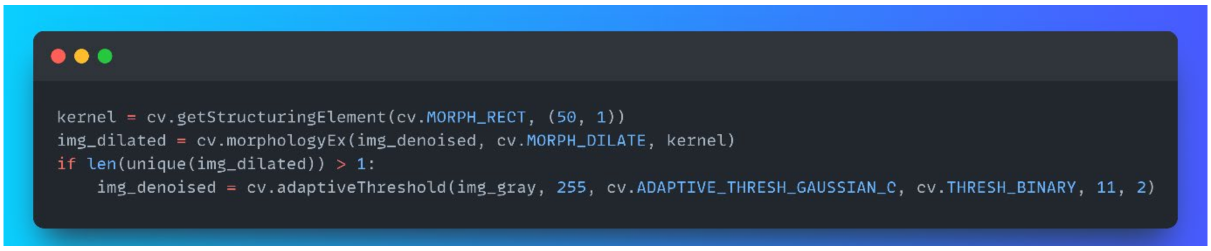


```
  img_blurred = cv.medianBlur(cv.blur(img_gray, (3, 3)), 3)
_, img_denoised = cv.threshold(img_blurred, 0, 255, cv.THRESH_BINARY + cv.THRESH_OTSU)
```

- A combination of median blurring and a box filter is applied to reduce "salt-and-pepper" noise.

- Otsu's thresholding is used to binarize the image while addressing brightness and contrast variations.

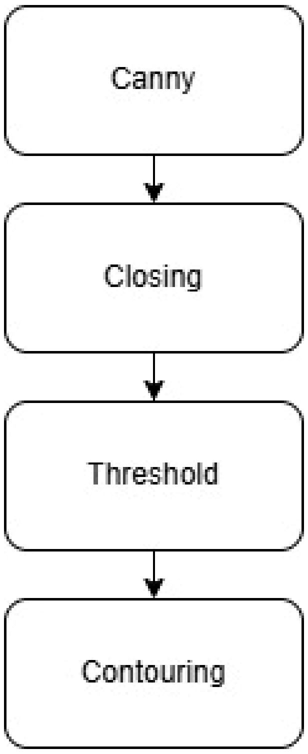
Sinusoidal Wave Noise Detection and Removal

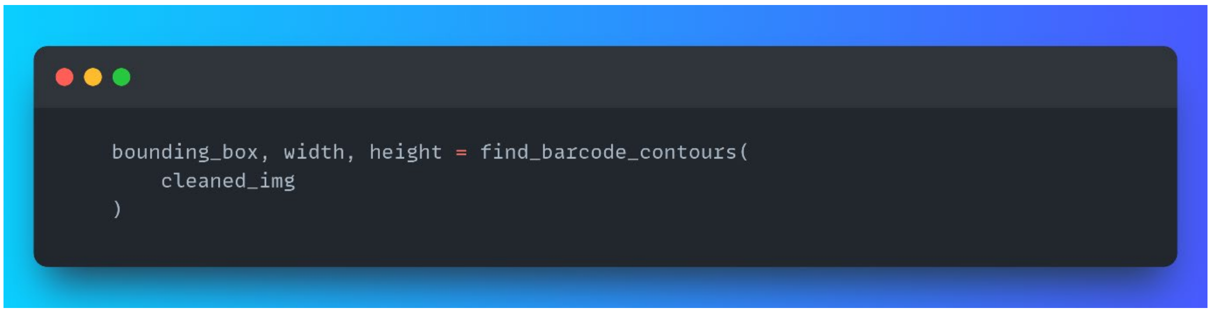


```
kernel = cv.getStructuringElement(cv.MORPH_RECT, (50, 1))
img_dilated = cv.morphologyEx(img_denoised, cv.MORPH_DILATE, kernel)
if len(unique(img_dilated)) > 1:
    img_denoised = cv.adaptiveThreshold(img_gray, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 11, 2)
```

- A horizontal structuring element (kernel) is used to dilate the binary image, removing horizontal bars while leaving sinusoidal patterns intact.
- If the dilation does not result in a uniform image (indicating the presence of sinusoidal noise), adaptive Gaussian thresholding is applied to counteract this distortion.

Find barcode contours

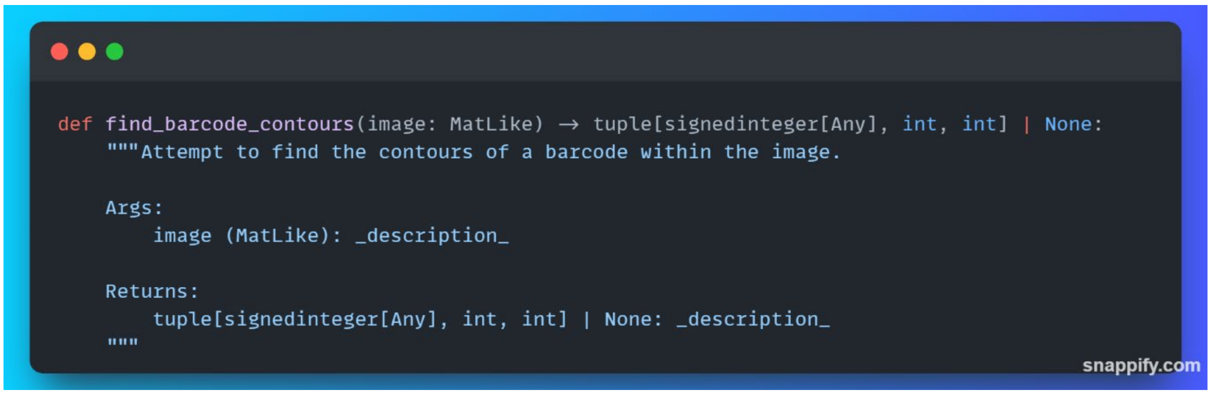




```python
bounding_box, width, height = find_barcode_contours(
    cleaned_img
)
```

Next, the function calls find_barcode_contours on the preprocessed image to detect the contours of the barcode. This function returns the bounding box coordinates, width, and height of the detected barcode. If no contours are found, the function prints a message and returns None.

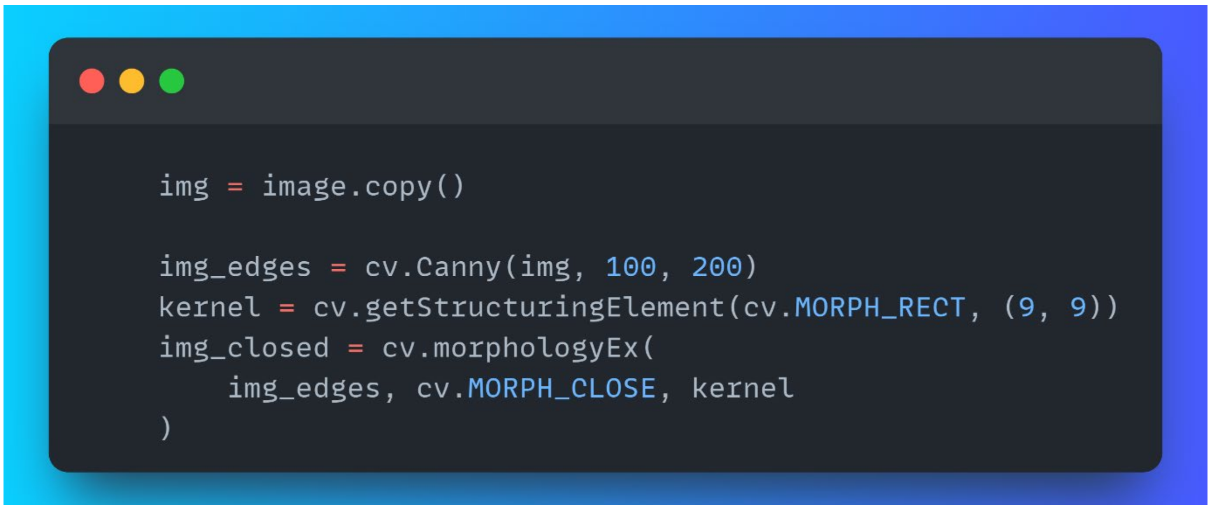
Function Overview



```python
def find_barcode_contours(image: MatLike) -> tuple[signedinteger[Any], int, int] | None:
    """Attempt to find the contours of a barcode within the image.

    Args:
        image (MatLike): _description_

    Returns:
        tuple[signedinteger[Any], int, int] | None: _description_
    """
```

The `find_barcode_contours` function is designed to locate the contours of a barcode within a given image. It takes a grayscale image as input and returns the coordinates of the bounding box around the barcode, along with its width and height. If no barcode is found, the function returns `(None, None, None)`.

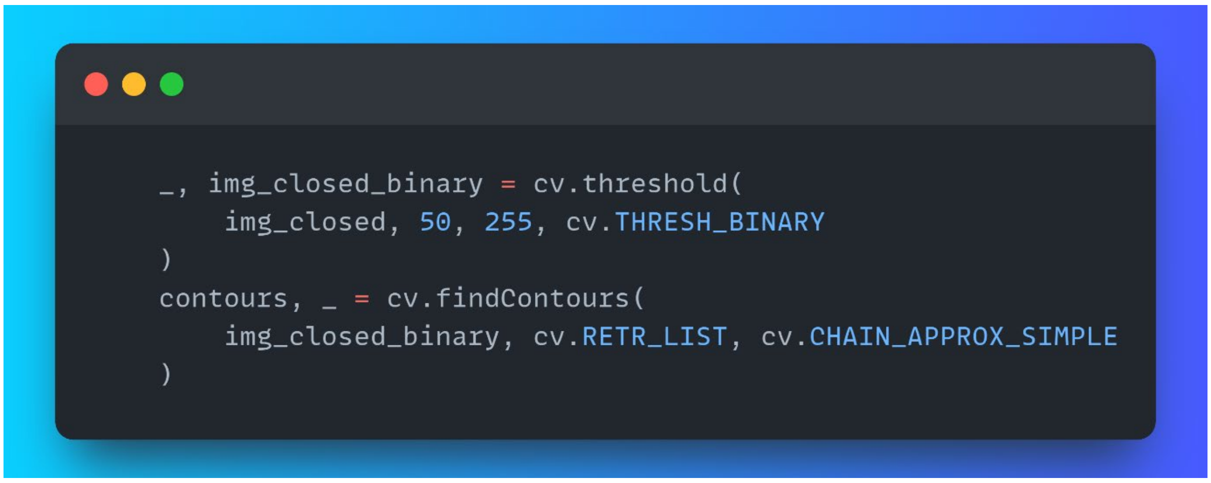
Preprocessing the Image



```python
img = image.copy()

img_edges = cv.Canny(img, 100, 200)
kernel = cv.getStructuringElement(cv.MORPH_RECT, (9, 9))
img_closed = cv.morphologyEx(
    img_edges, cv.MORPH_CLOSE, kernel
)
```

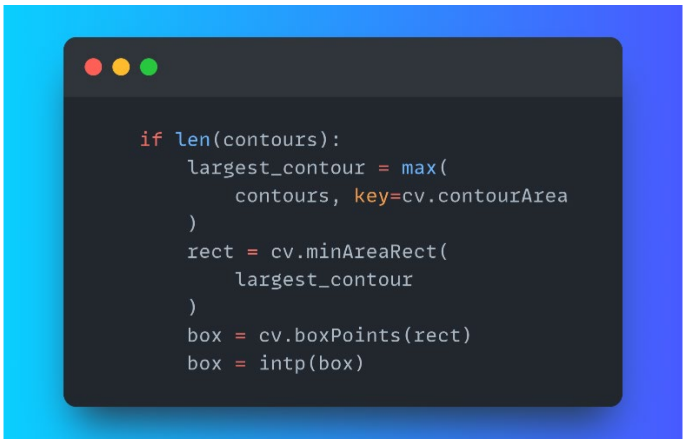
The function begins by creating a copy of the input image to avoid modifying the original. It then applies the Canny edge detection algorithm to highlight the edges within the image. This is achieved using the `cv.Canny` function, which takes the image and two threshold values as parameters. The resulting edge-detected image is then processed using morphological transformations to close any gaps in the edges. This is done using the `cv.morphologyEx` function with a square structuring element created by `cv.getStructuringElement`. The morphological closing operation helps in creating a continuous edge around the barcode.

Binarization and Contour Detection



```
_, img_closed_binary = cv.threshold(
    img_closed, 50, 255, cv.THRESH_BINARY
)
contours, _ = cv.findContours(
    img_closed_binary, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE
)
```

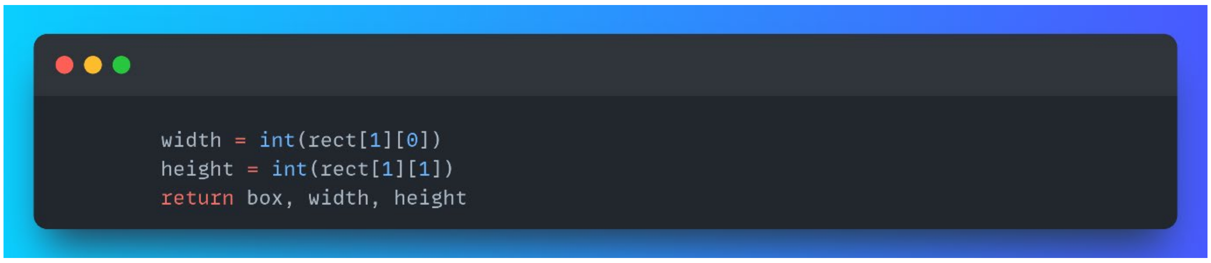
Next, the function binarizes the processed image using the `cv.threshold` function. This converts the image into a binary format where the pixel values are either 0 or 255, making it easier to detect contours. The `cv.findContours` function is then used to find all the contours in the binary image. Contours are simply the boundaries of objects within the image.



```
if len(contours):
    largest_contour = max(
        contours, key=cv.contourArea
    )
    rect = cv.minAreaRect(
        largest_contour
    )
    box = cv.boxPoints(rect)
    box = intp(box)
```

If any contours are found, the function identifies the largest contour, assuming it corresponds to the barcode. This is done using the `max` function with `cv.contourArea` as the key. The largest contour is then used to compute the minimum area rotated bounding rectangle using `cv.minAreaRect`. This rectangle is the smallest rectangle that can enclose the contour, and it can be rotated. The four corners of this rectangle are obtained using `cv.boxPoints` and converted to integer coordinates.

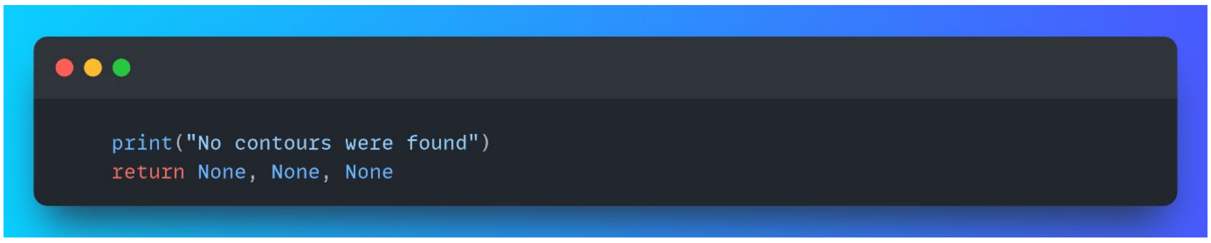
Drawing and Returning the Bounding Box



```python
    width = int(rect[1][0])
    height = int(rect[1][1])
    return box, width, height
```

For debugging purposes, the function converts the grayscale image to a color image and draws the bounding box around the detected barcode using `cv.drawContours`. This visual representation can be useful for verifying the accuracy of the detection for debugging. Finally, the function extracts the width and height of the bounding box from the rectangle and returns the coordinates of the bounding box, the width, and the height.

Handling No Contours Found

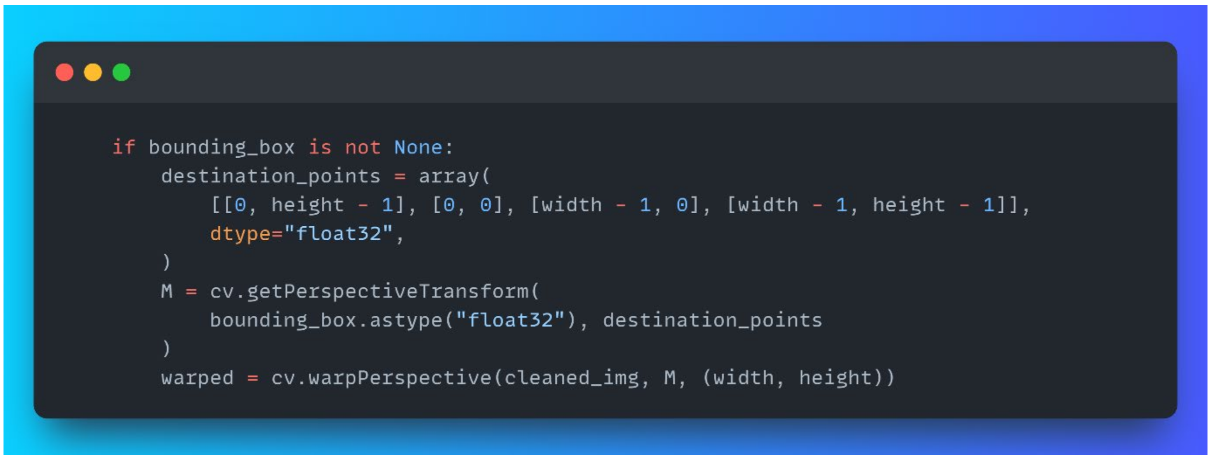


```python
print("No contours were found")
return None, None, None
```

If no contours are found in the image, the function prints a message indicating this and returns `(None, None, None)`.

Cropping and straightening the image

Perspective Transformation

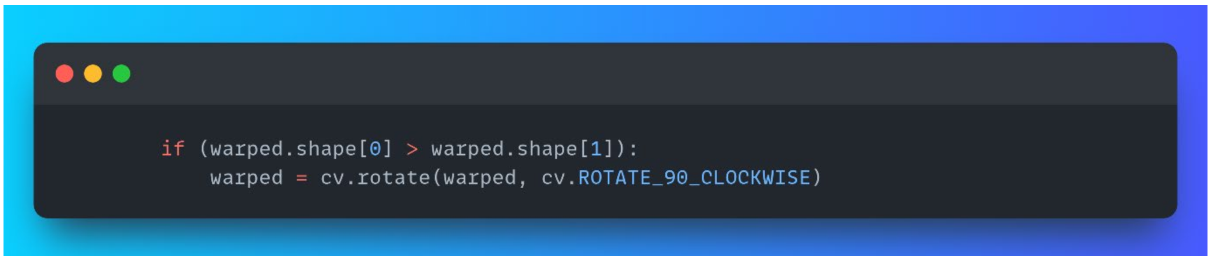


```python
if bounding_box is not None:
    destination_points = array(
        [[0, height - 1], [0, 0], [width - 1, 0], [width - 1, height - 1]],
        dtype="float32",
    )
    M = cv.getPerspectiveTransform(
        bounding_box.astype("float32"), destination_points
    )
    warped = cv.warpPerspective(cleaned_img, M, (width, height))
```

If the bounding box is found, the function proceeds with the perspective transformation:

- **Destination Points**: It defines the destination points as a rectangle with the same dimensions as the detected bounding box. These points are specified in a clockwise order starting from the bottom-left corner.
- **Transformation Matrix**: The `cv.getPerspectiveTransform` function computes the transformation matrix `M` that maps the bounding box coordinates to the destination points.
- **Warping the Image**: The `cv.warpPerspective` function applies the perspective transformation to the preprocessed image, resulting in a warped image where the barcode is aligned and centered.

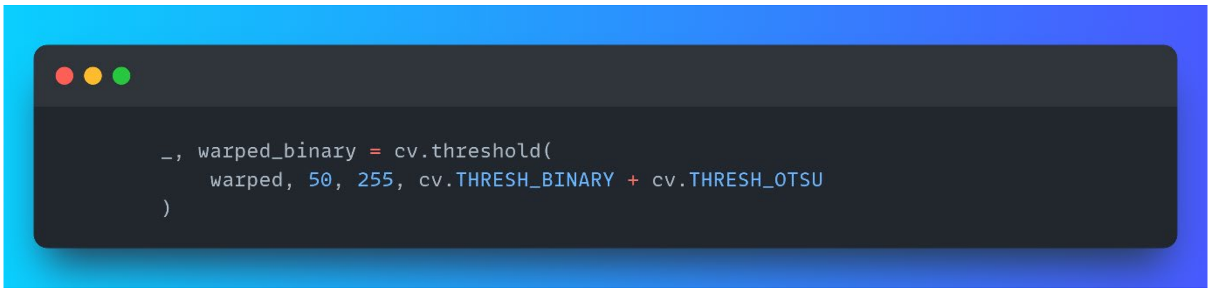
Orientation Correction



```
if (warped.shape[0] > warped.shape[1]):
    warped = cv.rotate(warped, cv.ROTATE_90_CLOCKWISE)
```

After warping, the function checks if the resulting image is in portrait orientation (height greater than width). If so, it rotates the image 90 degrees clockwise using `cv.rotate` to ensure the barcode is in landscape orientation, which is typically required for decoding.

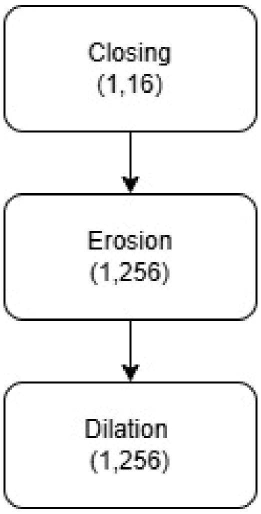
Binarization

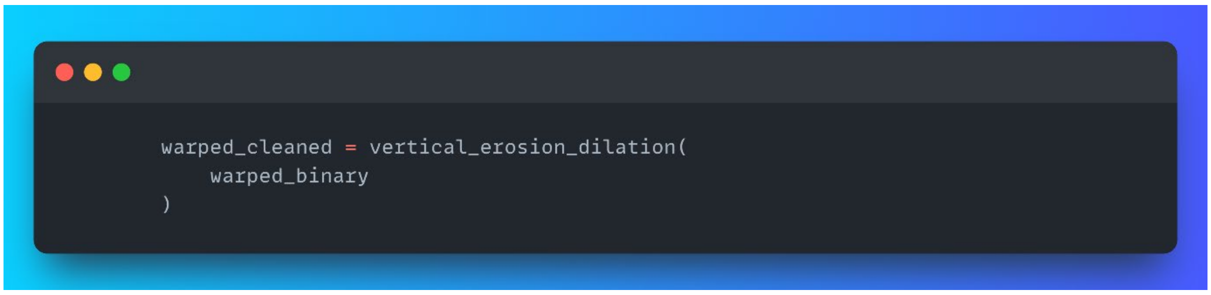


```
_, warped_binary = cv.threshold(
    warped, 50, 255, cv.THRESH_BINARY + cv.THRESH_OTSU
)
```

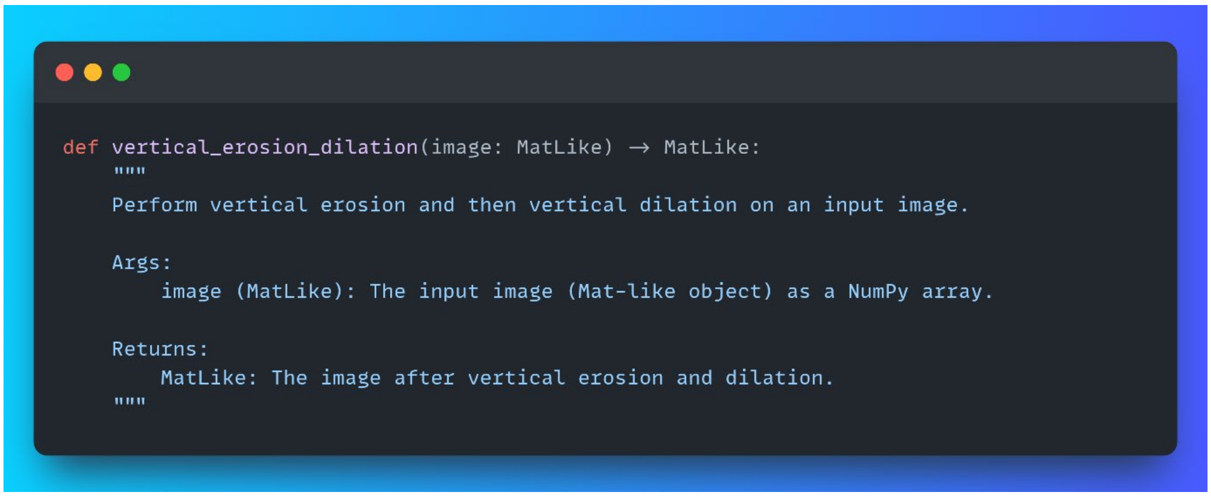
The warped image is then binarized using the `cv.threshold` function with Otsu's method. This step converts the image to a binary format, which is optimal for barcode decoding. Binarization helps in distinguishing the barcode's black and white bars clearly.

Cleaning the Barcode (Postprocessing)



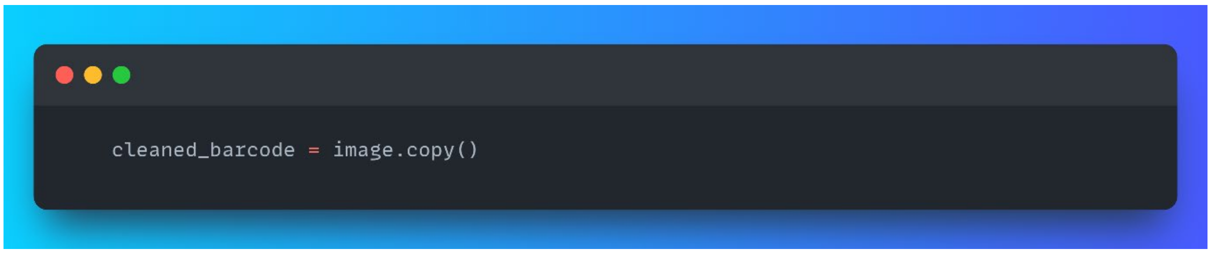


```python
warped_cleaned = vertical_erosion_dilation(
    warped_binary
)
```



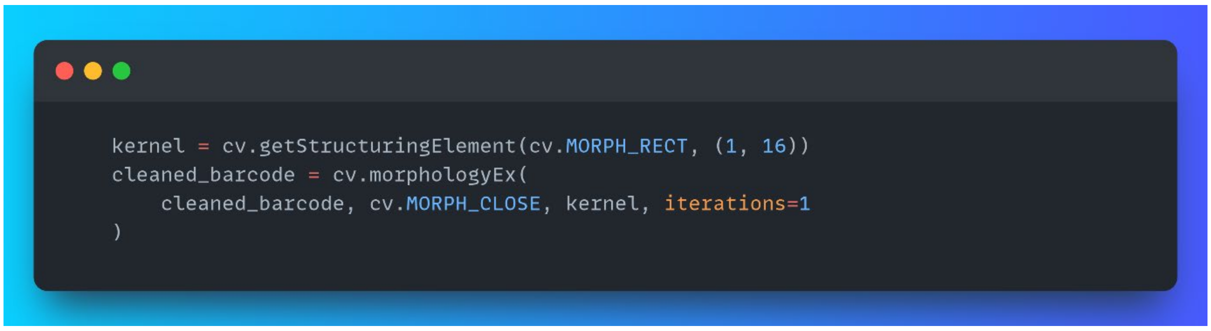
```python
def vertical_erosion_dilation(image: MatLike) -> MatLike:
    """
    Perform vertical erosion and then vertical dilation on an input image.

    Args:
        image (MatLike): The input image (Mat-like object) as a NumPy array.

    Returns:
        MatLike: The image after vertical erosion and dilation.
    """
```

To further enhance the barcode image, the function calls `vertical_erosion_dilation` on the binarized image. This step performs vertical erosion followed by vertical dilation to clean and restore the barcode after various transformations. It ensures that the vertical bars of the barcode are well-defined and free from noise.



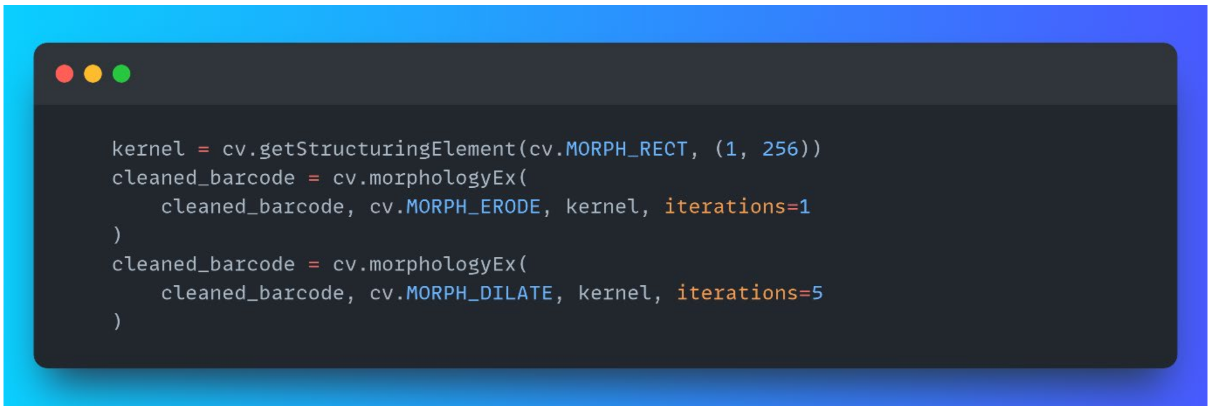
```
cleaned_barcode = image.copy()
```

The function begins by creating a copy of the input image to ensure that the original image remains unaltered. This is done using the image.copy() method, which creates a duplicate of the input image.



```
kernel = cv.getStructuringElement(cv.MORPH_RECT, (1, 16))
cleaned_barcode = cv.morphologyEx(
    cleaned_barcode, cv.MORPH_CLOSE, kernel, iterations=1
)
```

- **Kernel Creation**: The function creates a vertical rectangular structuring element using `cv.getStructuringElement`. The first kernel has a size of (1, 16), which is designed to close small gaps and remove black dots from the image. This kernel is vertical because the goal is to enhance vertical structures.
- **Morphological Closing**: The `cv.morphologyEx` function is used with the `cv.MORPH_CLOSE` operation and the first kernel. This operation helps in filling small black holes in the vertical white bars, making the structures more continuous. Closing is used instead of opening because aggressive vertical erosion will be applied before vertical dilation.

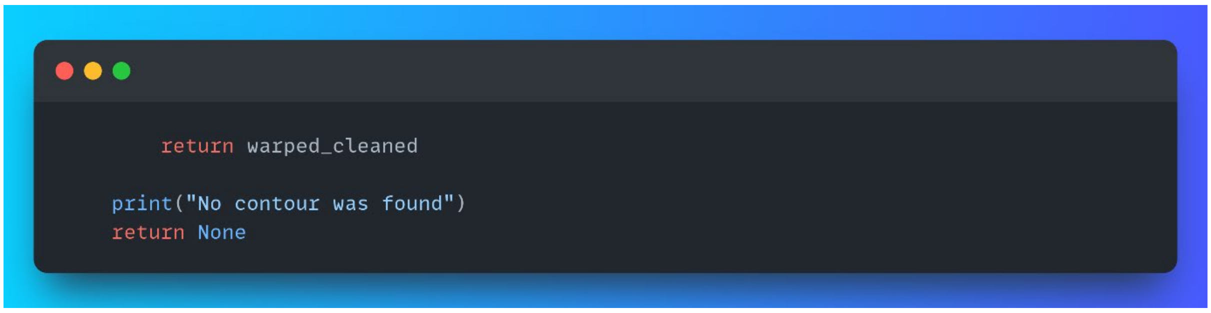


```
kernel = cv.getStructuringElement(cv.MORPH_RECT, (1, 256))
cleaned_barcode = cv.morphologyEx(
    cleaned_barcode, cv.MORPH_ERODE, kernel, iterations=1
)
cleaned_barcode = cv.morphologyEx(
    cleaned_barcode, cv.MORPH_DILATE, kernel, iterations=5
)
```

- **Vertical Erosion**: A second, larger vertical kernel of size (1, 256) is created. The `cv.morphologyEx` function is then applied with the `cv.MORPH_ERODE` operation using this larger kernel. Erosion reduces the thickness of the vertical structures, effectively removing small noise elements and isolating the main vertical components.
- **Vertical Dilation**: Finally, the `cv.morphologyEx` function is applied with the `cv.MORPH_DILATE` operation using the same large kernel. Dilation increases the thickness of

the vertical structures, restoring them to their original size or even making them slightly thicker. This step ensures that the vertical bars are prominent and well-defined.

Return



```
        return warped_cleaned

    print("No contour was found")
    return None
```

Finally, the function returns the cleaned barcode image. If no barcode contours were found during the contour detection step, the function prints a message indicating this and returns `None`.