HO CHI MINH UNIVERSITY OF SCIENCE

APPLIED DIGITAL IMAGE AND VIDEO PROCESSING
LAB01

April 6, 2024

Author

Nguyen Viet Kim - 21127333

Contents

1 Self-Assessment

No.	Tasks	Percentage
1	Binary Dilation	100%
2	Binary Erosion	100%
3	Binary Opening	100%
4	Binary Closing	100%
5	Binary Hit-or-miss	100%
6	Binary Boundary Extraction	100%
7	Binary Thinning	100%
8	Grayscale Dilation	100%
9	Grayscale Erosion	100%
10	Grayscale Opening	100%
11	Grayscale Closing	100%
11	Grayscale Gradient	100%
11	Grayscale Top-hat	100%
11	Grayscale Black-hat	100%

2 Algorithm Description

2.1 Required Libraries

2.2 Binary Dilation

Given a binary image A and a structuring element B, the dilation operation $A \oplus B$ results in a new binary image where each foreground pixel (usually represented as 1) in A is replaced by 1 if there is at least one foreground pixel under the structuring element B.
In mathematical terms, dilation can be defined as the union of A with the translated kernel B, where the translation is performed over all possible positions in the image. This can be expressed as:

$$(A \oplus B)(x, y) = \bigcup_{(i,j)\in B} A(x-i, y-j)$$

In words, this equation states that the value of the pixel at position (x, y) in the dilated image is determined by taking the union (or maximum) of the pixel values in the input image A at positions $(x-i, y-j)$ for all points (i, j) in the structuring element B.
Alternatively, dilation can be defined as the set of all points (x, y) such that for at least one point (i, j) in the structuring element B, the pixel at $(x+i, y+j)$ in A is foreground:

$$A \oplus B = \{(x, y) \mid \text{for some } (i, j) \in B, A(x-i, y-j) = 1\}$$

The pseudo code for the custom dilation function:

```
function dilation(input_image, kernel):
    // Define output image
    output_image = create_empty_image_like(input_image)

    // Get kernel offset
    offset_x = kernel.width / 2
    offset_y = kernel.height / 2
```

```

    // Loop through each pixel
    for each white pixel (x, y) in input_image:
        for each kernel element (i, j):
            new_x = x + i - offset_x
            new_y = y + j - offset_y

            if kernel[i, j] != 0:
                output_image[new_x, new_y] = 255

    return output_image
```

2.3 Binary Erosion

Given a binary image A and a structuring element B, the erosion operation $A \ominus B$ results in a new binary image where each foreground pixel (usually represented as 1) in A is replaced by 0 unless all the pixels under the structuring element B are also foreground pixels.
In mathematical terms, erosion can be defined as the intersection of A with the translated kernel B, where the translation is performed over all possible positions in the image. This can be expressed as:

$$(A \ominus B)(x, y) = \bigcap_{(i,j)\in B} A(x+i, y+j)$$

Alternatively, erosion can be defined as the set of all points (x, y) such that for all points (i, j) in the structuring element B, the pixel at $(x+i, y+j)$ in A is foreground:

$$A \ominus B = \{(x, y) \mid \text{for all } (i, j) \in B, A(x+i, y+j) = 1\}$$

In practical terms, erosion removes pixels from the boundaries of objects in the image, effectively reducing the size of the objects. It is useful for operations such as noise reduction, object separation, and image segmentation.

```
function erosion(input_image, kernel):
    // Define output image
    output_image = create_empty_image_like(input_image)

    // Get kernel offset
    offset_x = kernel.width / 2
    offset_y = kernel.height / 2

    // Loop through each pixel
    for each white pixel (x, y) in input_image:
        // Check if the center pixel is white (255)
        if input_image[y][x] == 255:
            // Check if all neighboring pixels covered by the kernel are white
            erosion_flag = True
            for each kernel element (i, j):
                new_x = x + i - offset_x
                new_y = y + j - offset_y

                if kernel[i, j] == 1 and input_image[new_y][new_x] != 255:
                    erosion_flag = False
                    break
```

```
            // If any neighboring pixel is not white, set the center pixel to black
    (0) in the output image
            if erosion_flag:
                output_image[y][x] = 255

    return output_image
```

2.4 Binary Opening

Given a binary image A and a structuring element B, the opening operation $A \circ B$ results in a new binary image where each pixel is set to 1 if it remains unchanged under the erosion operation followed by the dilation operation.
Mathematically, opening can be defined as:

$$A \circ B = (A \ominus B) \oplus B$$

Alternatively, opening can be expressed as the set of all points (x, y) such that for every point (i, j) in the structuring element B, the pixel at $(x+i, y+j)$ in A is foreground:

$$A \circ B = \{(x, y) \mid \text{for all } (i, j) \in B, A(x+i, y+j) = 1\}$$

Binary opening is useful for removing small objects, smoothing object contours, and breaking narrow bridges or connections between objects.

```
function open(input_image, kernel):
    // Perform erosion using custom implementation
    eroded_img_custom = erosion(input_image, kernel)

    // Perform dilation using custom implementation on the eroded image
    opened_img_custom = dilation(eroded_img_custom, kernel)

    return opened_img_custom
```

2.5 Binary Closing

Given a binary image A and a structuring element B, the closing operation $A \bullet B$ results in a new binary image where each pixel is set to 1 if it remains unchanged under the dilation operation followed by the erosion operation.
Mathematically, closing can be defined as:

$$A \bullet B = (A \oplus B) \ominus B$$

Alternatively, closing can be expressed as the set of all points (x, y) such that for at least one point (i, j) in the structuring element B, the pixel at $(x+i, y+j)$ in A is foreground:

$$A \bullet B = \{(x, y) \mid \text{for at least one } (i, j) \in B, A(x+i, y+j) = 1\}$$

Binary closing is useful for filling small holes in objects, joining narrow gaps or breaks in object contours, and smoothing object boundaries.

```
function close(input_image, kernel):
    // Perform dilation using custom implementation
    dilated_img_custom = dilation(input_image, kernel)

    // Perform erosion using custom implementation on the dilated image
    closed_img_custom = erosion(dilated_img_custom, kernel)

    return closed_img_custom
```

2.6 Grayscale Dilation

The dilation morphological operator applied on the grayscale image is described as the following equation:

$$(f \oplus b)(x, y) = \max_{(s,t)\in B} \{f(x-s, y-t) + b(s, t)\}$$

where x, y are the pixel's position, s, t is the kernel index.

```
function grayscale_dilation(img, kernel):
    for each pixel (x, y) in img:
        max_value = -infinity

        // Apply the kernel
        for each element (i, j) in kernel:
            padded_x = x - center(kernel) + i
            padded_y = y - center(kernel) + j

            if (padded_x, padded_y) is within bounds of img:
                max_value = max(max_value, img[padded_x, padded_y] + kernel[i, j])

        dilated_img[x, y] = max_value

    return dilated_img
```

2.7 Grayscale Erosion

The grayscale image erosion is shown by the following formula:

$$(f \ominus b)(x, y) = \min_{(s,t)\in B} \{f(x+s, y+t) - b(s, t)\}$$

where x, y are the pixel's position, s, t is the kernel index.

```
function grayscale_erosion(img, kernel):
    for each pixel (x, y) in img:
        min_value = +infinity

        // Apply the kernel
        for each element (i, j) in kernel:
            padded_x = x - center(kernel) + i
            padded_y = y - center(kernel) + j

            if (padded_x, padded_y) is within bounds of img:
                min_value = min(min_value, img[padded_x, padded_y] - kernel[i, j])

        eroded_img[x, y] = min_value
```

```python

    return eroded_img
```

2.8 Grayscale Opening

The opening of the grayscale image is the combination of erosion and following dilation:

$$(f \circ b)(x, y) = ((f \ominus b) \oplus b)(x, y)$$

```python
def grayscale_opening(img, kernel):
    # Perform grayscale erosion followed by grayscale dilation
    eroded_img = grayscale_erosion(img, kernel)
    opened_img = grayscale_dilation(eroded_img, kernel)
    return opened_img
```

2.9 Grayscale Closing

The grayscale closing is performed by dilation then erosion the image:

$$(f \bullet b)(x, y) = ((f \oplus b) \ominus b)(x, y)$$

```python
def grayscale_closing(img, kernel):
    # Perform grayscale dilation followed by grayscale erosion
    dilated_img = grayscale_dilation(img, kernel)
    closed_img = grayscale_erosion(dilated_img, kernel)
    return closed_img
```

2.10 Grayscale Gradient

The grayscale gradient is computed by the difference between dilation and erosion of the image:

$$(f \nabla b)(x, y) = ((f \oplus b) - (f \ominus b))(x, y)$$

```python
def grayscale_gradient(img, kernel):
    # Perform grayscale dilation
    dilated_img = grayscale_dilation(img, kernel)
    # Perform grayscale erosion
    eroded_img = grayscale_erosion(img, kernel)
    # Compute the gradient by subtracting the eroded image from the dilated image
    gradient_img = dilated_img - eroded_img
    return gradient_img
```

2.11 Grayscale Top-hat

The Top-hat morphological operator computes the difference between the original image and the opened image:

$$T = f - (f \circ b)$$

```python
def top_hat_transform(img, kernel):
    # Perform grayscale opening
    opened_img = grayscale_opening(img, kernel)
```

```
    # Compute the top-hat transform by subtracting the opened image from the original image
    top_hat_img = img - opened_img
    return top_hat_img
```

2.12 Grayscale Black-hat

The Black-hat has the same concept of the top-hat operator where the image is computed by the difference between original image and closed image:

$$B = (f \bullet b) - f$$

```
def black_hat_transform(img, kernel):
    # Perform grayscale closing
    closed_img = grayscale_closing(img, kernel)
    # Compute the black-hat transform by subtracting the original image from the closed image
    black_hat_img = closed_img - img
    return black_hat_img
```

3 Result

The original image:

Figure 1: Original Image

3.1 Binary Dilation

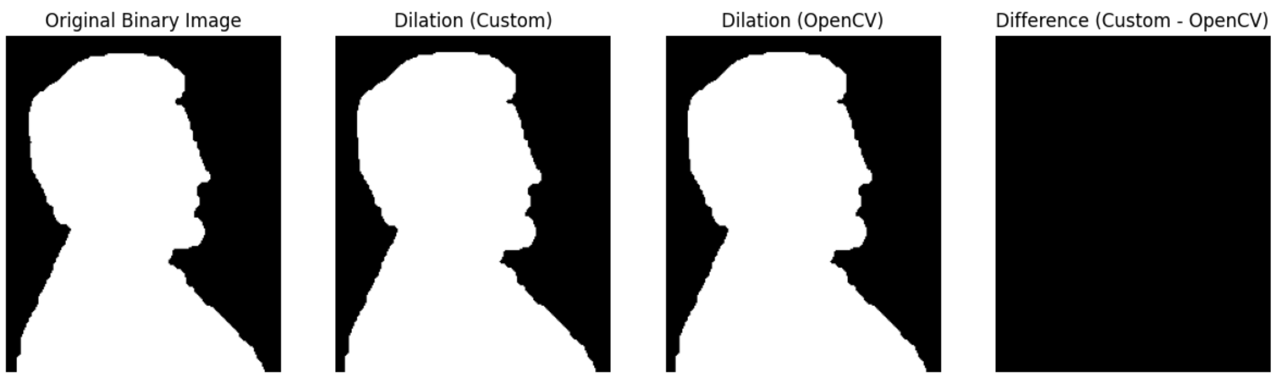

Figure 2: Binary Dilation comparing with openCV

3.2 Binary Erosion

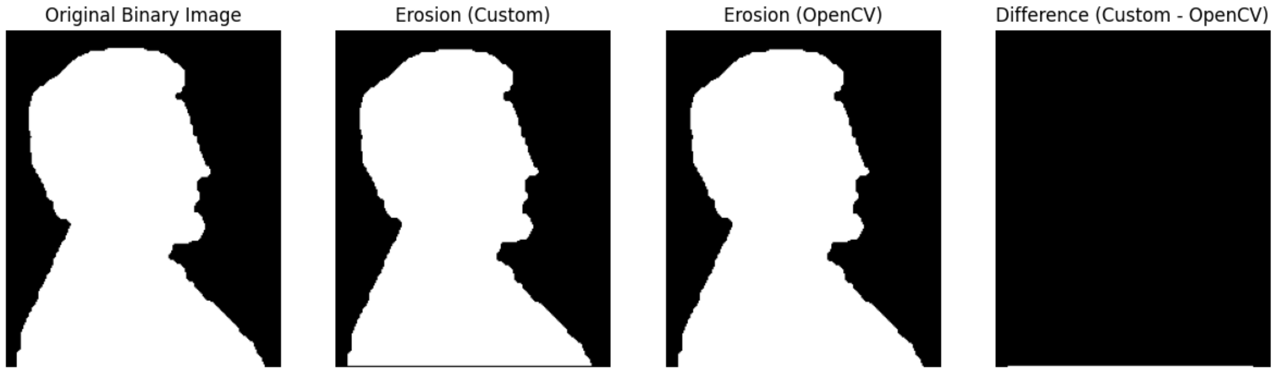

Figure 3: Binary Erosion comparing with openCV

3.3 Binary Opening

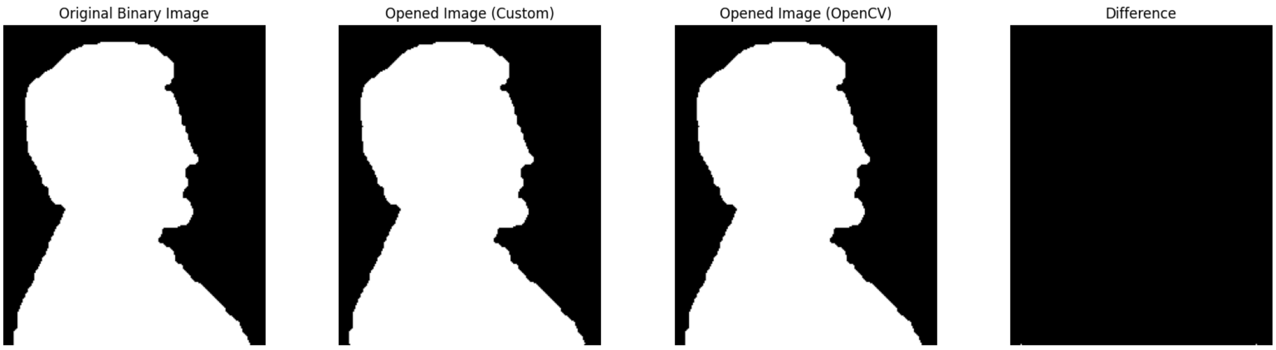

Figure 4: Binary Opening comparing with openCV

3.4 Binary Closing

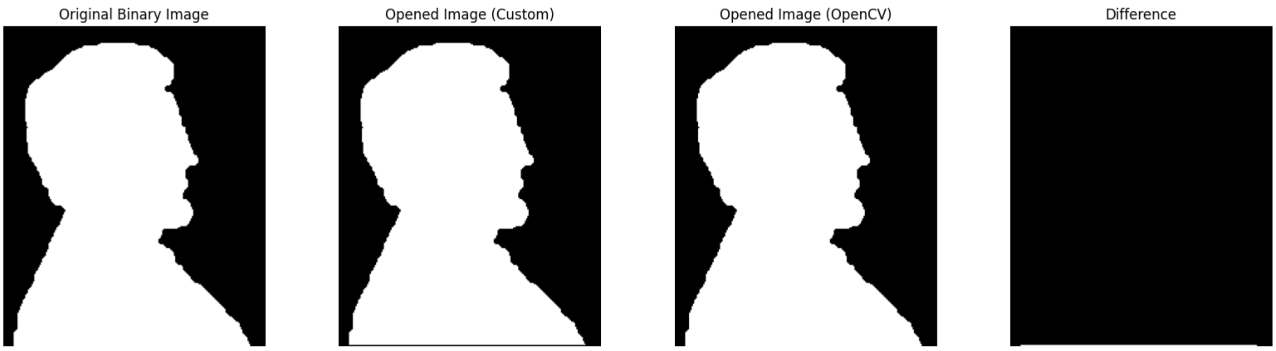

Figure 5: Binary Closing comparing with openCV

3.5 Grayscale Dilation

Figure 6: Grayscale Dilation

3.6 Grayscale Erosion

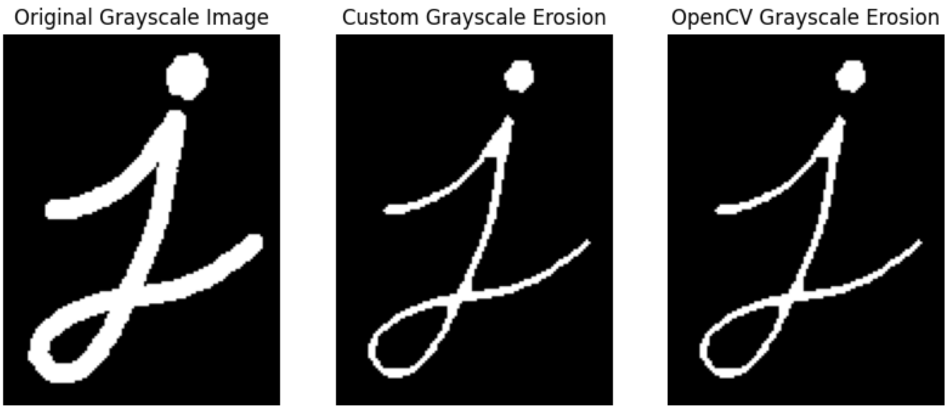

Figure 7: Grayscale Erosion

3.7 Grayscale Opening

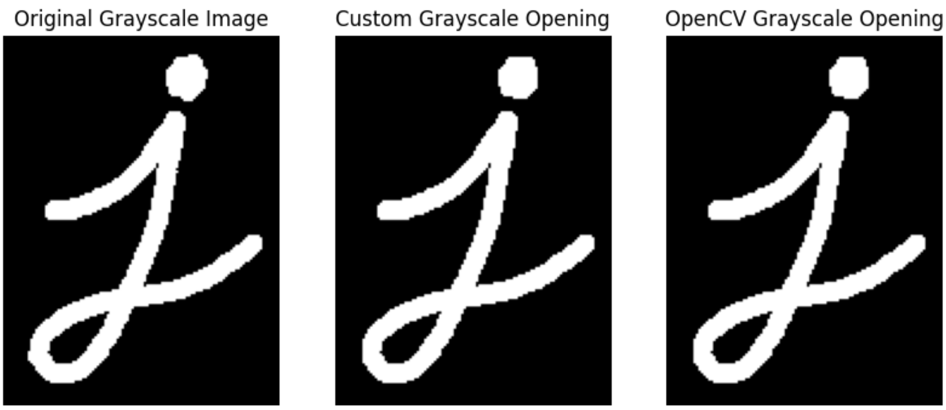

Figure 8: Grayscale Opening

3.8 Grayscale Closing

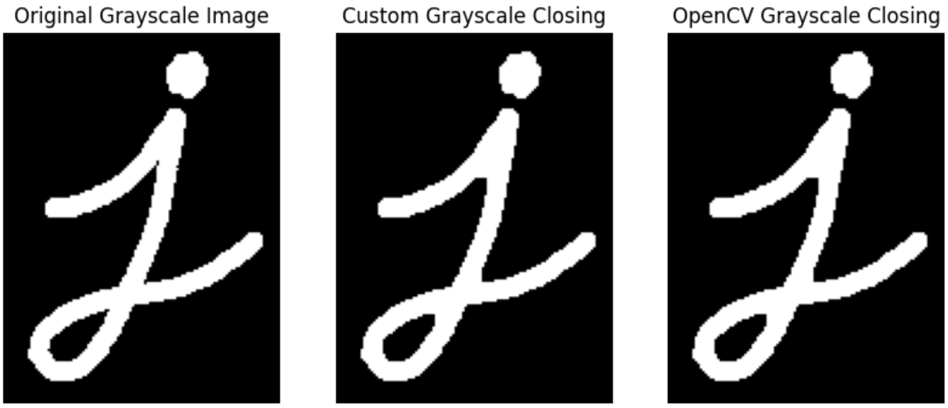

Figure 9: Grayscale Closing

3.9 Grayscale Gradient

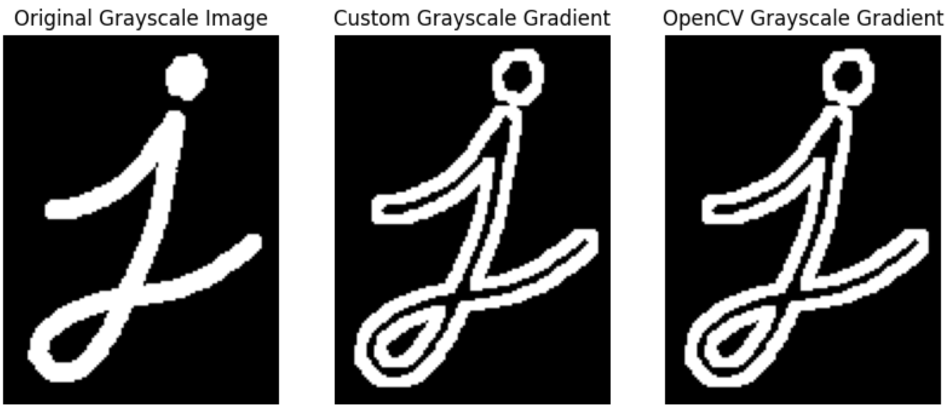

Figure 10: Grayscale Gradient

3.10 Grayscale Top-hat

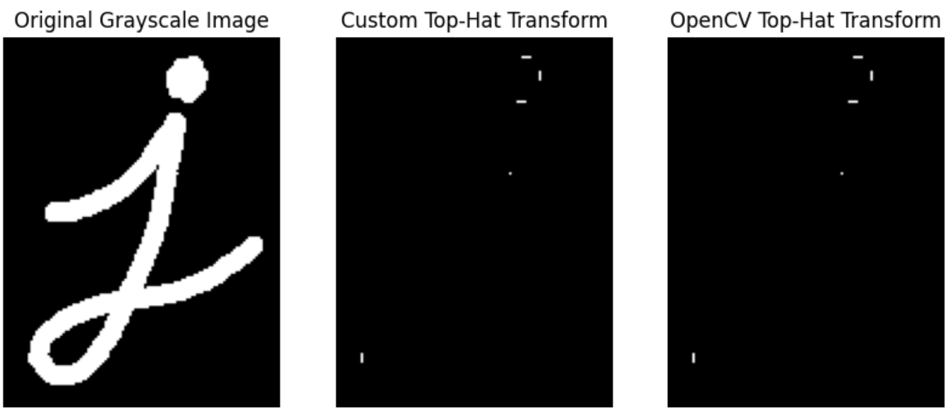

Figure 11: Grayscale Top-hat comparing with openCV

3.11 Grayscale Black-hat

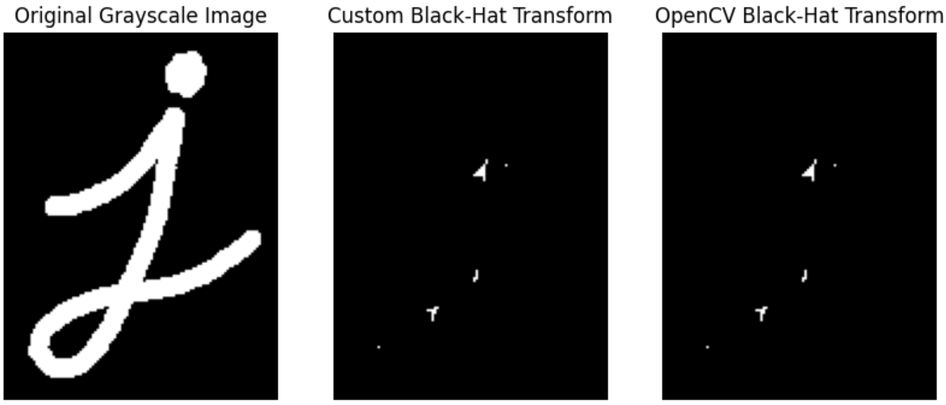

Figure 12: Grayscale Black-hat comparing with openCV