HCMUS - VNUHCM / FIT / Computer Vision & Cognitive Cybernetics Department

Digital Image and Video Processing Application

Student ID: 21127690

Student name: Ngo Nguyen Thanh Thanh

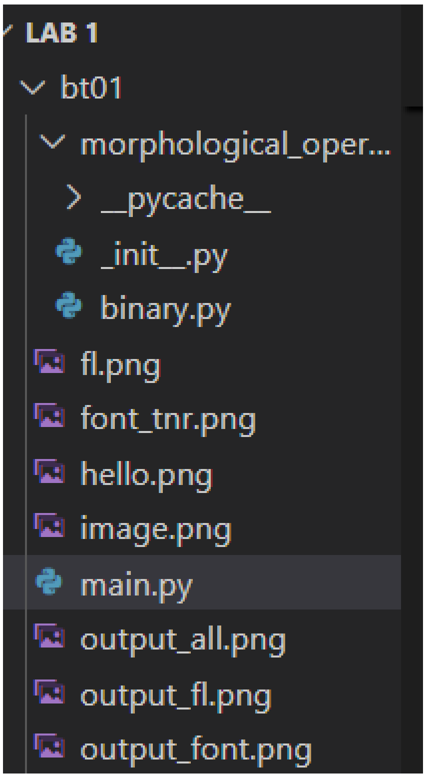
Report: Morphological Operations (operators up to week of 14 Feb 2025)

I. Evaluation summary:

No	Task	Implementation (Without OpenCV)	Time Complexity (Without OpenCV)	Implementation (With OpenCV)	Time Complexity (With OpenCV)	Completion (%)
1	**Binary Dilation**	Implemented using manual convolution and max filter.	O(n2·k2) (for kernel size k)	Uses cv2.dilate() with a structuring element.	O(n2) (optimized)	100%
2	**Binary Erosion**	Implemented using manual convolution and min filter.	O(n2·k2) (for kernel size k)	Uses cv2.morphologyEx(..., cv2.MORPH_OPEN).	O(n2)	100%
3	**Binary Opening**	Combines manual erosion and dilation sequentially	O(n2·k2) (for kernel size k)	Uses cv2.morphologyEx(..., cv2.MORPH_OPEN).	O(n2) (optimized)	100%
4	**Binary Closing**	Combines manual dilation and erosion sequentially.	O(n2·k2) (for kernel size k)	Uses cv2.morphologyEx(..., cv2.MORPH_CLOSE).	O(n2)	100%
5	**Hit-or-Miss**	Implemented using two structuring elements and logical operations.	O(n2·k2) (for kernel size k)	Uses cv2.morphologyEx(..., cv2.MORPH_HITMISS).	O(n2)	100%
6	**Boundary Extraction**	Subtracts eroded image from the original manually.	O(n2·k2) (for kernel size k)	Uses cv2.subtract(image, cv2.erode(image, kernel)).	O(n2)	100%
7	**Region Filling**	Uses iterative processing with a seed-based algorithm.	O(n2) to (n3) (depending on number of iterations)	Uses OpenCV flood-fill (cv2.floodFill()).	O(n2) (optimized)	100%

II. List of features and file structure:

File structure:



- **morphological_operations/**: A package containing functions for morphological operations.
- **__init__.py**: Defines morphological_operations as a package, allowing imports.
- **binary.py**: Implements image processing functions.
- **main.py** (outside the package): The main script that likely imports binary.py and runs the program.
- **Image files (.png)**: Used as input or for testing.

This structure keeps functions modular (binary.py) and execution separate (main.py).

Functions and methods used:

1. binary.py (Custom Binary Image Processing Library)

Main functions:

- **pad_image()**: Adds padding to the image to avoid errors during processing.
- **erode()**: Performs the erosion operation to shrink the bright areas.
- **dilate()**: Performs the dilation operation to expand the bright areas.
- **opening()**: Performs the opening operation (Erosion → Dilation) to remove small noise.
- **closing()**: Performs the closing operation (Dilation → Erosion) to fill small gaps.
- **hit_or_miss()**: Applies the Hit-or-Miss operation to detect specific shapes or patterns.

- **boundary_extraction()**: Extracts the boundary of the bright regions.
- **region_filling()**: Fills the bright regions based on dilation.

2. main.py (Main Program)

Main functions:

- Read the input image and convert it to a binary image.
- Provide two processing modes:
  - **Manual**: Use custom algorithms from **binary.py**.
  - **OpenCV**: Use the OpenCV library for processing.
- Support operations: Dilate, Erode, Open, Close, HitMiss, Boundary, Fill.
- Display and save the processed image.
- Measure and display the execution time of each method.

Details:

- **apply_manual(img, kernel):** Applies custom morphological operations.
- **apply_opencv(img, kernel):** Applies morphological operations using OpenCV.
- **operator(in_file, out_file, mor_op, mode, wait_key_time=0):** Processes image with specified operation and mode, displays and saves results, measures execution time.
- **main(argv):** Parses command-line arguments and calls operator() to process the image.

How to run code:

1. Install Required Libraries

First, make sure you have the necessary libraries installed, such as opencv, numpy, and morphological_operator. You can install them using pip:

pip install opencv-python numpy

Note: The morphological_operator library appears to be a custom-written library, so make sure it exists in the same directory or has been correctly installed.

2. Command Line Structure

Here's the command to run the program from the command line:

python main.py -i <input_file> -o <output_file> [-p <morph_operator>] -m <mode> -t <wait_key_time>

3. Explanation of Parameters:

- -i <input_file>: Path to the input image file.
- -o <output_file>: Path to save the result.
- -p <morph_operator> (Optional): The specific morphological operation you want to apply (e.g., "Dilate", "Erode", etc.).
- -m <mode>: The execution mode ("manual" for custom-written algorithms or "opencv" for OpenCV).

- -t <wait_key_time> (Optional): Time to wait (in milliseconds) before closing the window displaying the image.

4. Example:

Suppose you have an image file input.jpg and you want to apply the "Dilate" operation using the custom algorithm (manual mode) and save the result to output.jpg. You would run:

python main.py -i input.jpg -o output.jpg -p Dilate -m manual

5. Options:

- If you don't specify a morphological operation, the program will apply all available operations.
- If you don't specify a wait time, the program will not wait and will automatically close the window displaying the image.

6. Error Information:

If there's an error, such as a missing image file or incorrect parameter, the program will show an appropriate error message.

Image proof:

Binary.py

bt01 > morphological_operator > binary.py > erode

```python
import numpy as np

def pad_image(img, kernel):
    """Thêm padding vào ảnh để tránh lỗi tràn khi thực hiện phép toán hình thái."""
    pad_h, pad_w = kernel.shape[0] // 2, kernel.shape[1] // 2  # Tính toán số pixel cần pad
    return np.pad(img, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant', constant_values=0)

def erode(img, kernel):
    """Thực hiện phép co (Erosion) để thu nhỏ vùng sáng."""
    padded_img = pad_image(img, kernel)  # Thêm padding vào ảnh
    result = np.zeros_like(img)  # Khởi tạo ảnh kết quả với giá trị 0
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            region = padded_img[i:i+kernel.shape[0], j:j+kernel.shape[1]]  # Lấy vùng con
            if np.array_equal(region * kernel, kernel):  # Kiểm tra nếu trùng khớp với kernel
                result[i, j] = 1  # Đặt giá trị pixel là 1
    return result

def dilate(img, kernel):
    """Thực hiện phép giãn (Dilation) để mở rộng vùng sáng."""
    padded_img = pad_image(img, kernel)  # Thêm padding vào ảnh
    result = np.zeros_like(img)  # Khởi tạo ảnh kết quả với giá trị 0
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            region = padded_img[i:i+kernel.shape[0], j:j+kernel.shape[1]]  # Lấy vùng con
            if np.any(region * kernel):  # Nếu có ít nhất một phần tử là 1
                result[i, j] = 1  # Đặt giá trị pixel là 1
    return result
```

```python
def opening(img, kernel):
    """Phép mở: Erosion trước, sau đó Dilation (giúp loại bỏ nhiễu nhỏ)."""
    return dilate(erode(img, kernel), kernel)

def closing(img, kernel):
    """Phép đóng: Dilation trước, sau đó Erosion (giúp lấp đầy các lỗ hổng nhỏ)."""
    return erode(dilate(img, kernel), kernel)

def hit_or_miss(img, kernel):
    """Phép toán Hit-or-Miss để tìm các mẫu hình dạng cụ thể trong ảnh."""
    complement = 1 - img  # Lấy ảnh nền (background)

    # Xác định hai phần của kernel
    kernel_fg = (kernel == 1).astype(np.uint8)  # B1: foreground (các giá trị 1 trong kernel)
    kernel_bg = (kernel == -1).astype(np.uint8)  # B2: background (các giá trị -1 trong kernel)

    # Thực hiện phép co trên cả hai phần
    eroded_fg = erode(img, kernel_fg)  # Co ảnh với foreground
    eroded_bg = erode(complement, kernel_bg)  # Co ảnh với background

    # Lấy giao của hai ảnh co để tìm vùng khớp hoàn toàn
    return np.logical_and(eroded_fg, eroded_bg).astype(np.uint8)
```

```python
def boundary_extraction(img, kernel):
    """Tách đường biên của vùng sáng trong ảnh."""
    return img - erode(img, kernel)  # Lấy phần ảnh ban đầu trừ đi ảnh bị co

def region_filling(img, kernel, seed):
    """Thuật toán lấp đầy vùng sáng dựa trên phép toán giãn (Dilation)."""
    result = np.zeros_like(img, dtype=np.uint8)  # Ảnh kết quả ban đầu (tất cả là 0)
    result[seed] = 1  # Đặt pixel seed ban đầu thành 1

    # Xác định vùng nền (background)
    background = 1 - img  # Đảm bảo chỉ mở rộng vào vùng nền

    while True:
        new_result = dilate(result, kernel) & background  # Giãn vùng seed nhưng giữ trong nền
        if np.array_equal(new_result, result):  # Nếu không có thay đổi, dừng lặp
            break
        result = new_result  # Cập nhật ảnh kết quả

    return result
```

Main.py

bt01 > main.py > apply_manual

```python
import sys
import getopt
import cv2
import numpy as np
import os
import time
from morphological_operator import binary  # Import thư viện xử lý hình thái tự viết

def apply_manual(img, kernel):
    """Áp dụng các phép toán hình thái bằng thuật toán tự viết."""
    return {
        "Original": img,  # Ảnh gốc
        "Dilate": binary.dilate(img, kernel),  # Giãn nở (Dilation)
        "Erode": binary.erode(img, kernel),  # Co lại (Erosion)
        "Open": binary.opening(img, kernel),  # Mở (Opening)
        "Close": binary.closing(img, kernel),  # Đóng (Closing)
        "HitMiss": binary.hit_or_miss(img, kernel),  # Hit-or-Miss
        "Boundary": binary.boundary_extraction(img, kernel),  # Trích xuất biên
        "Fill": binary.region_filling(img, kernel, (10, 10))  # Lấp vùng
    }

def apply_opencv(img, kernel):
    """Áp dụng các phép toán hình thái bằng OpenCV."""
    return {
        "Original": img,
        "Dilate (OpenCV)": cv2.dilate(img, kernel),  # Giãn nở dùng OpenCV
        "Erode (OpenCV)": cv2.erode(img, kernel),  # Co lại dùng OpenCV
        "Open (OpenCV)": cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel),  # Mở
        "Close (OpenCV)": cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel),  # Đóng
        "HitMiss (OpenCV)": cv2.morphologyEx(img, cv2.MORPH_HITMISS, kernel),  # Hit-or-Miss
        "Boundary (OpenCV)": cv2.dilate(img, kernel) - img,  # Biên = (Giãn nở - Ảnh gốc)
        "Fill (OpenCV)": cv2.floodFill(img.copy(), None, (10, 10), 255)[1]  # Lấp vùng bằng floodFill
    }

```

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=0):
    """Thực hiện phép toán hình thái trên ảnh."""

    # Kiểm tra xem tệp ảnh có tồn tại không
    if not os.path.exists(in_file):
        print(f"Error: Input file '{in_file}' not found.")
        sys.exit(1)

    # Đọc ảnh đầu vào ở chế độ grayscale
    img_origin = cv2.imread(in_file, 0)
    if img_origin is None:
        print(f"Error: Unable to read image file '{in_file}'. Check file format and path.")
        sys.exit(1)

    # Chuyển ảnh về nhị phân bằng threshold
    _, img = cv2.threshold(img_origin, 128, 1, cv2.THRESH_BINARY_INV if np.mean(img_origin) > 128 else cv2.THRESH_BINARY)

    # Kernel 3x3 dùng cho phép toán hình thái
    kernel = np.array([[0, 1, 0],
                       [1, 1, 1],
                       [0, 1, 0]], dtype=np.uint8)

    # Bắt đầu tính thời gian thực thi
    start_time = time.time()
```

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=0):

    # Chọn chế độ thực hiện
    if mode == "manual":
        operations = apply_manual(img, kernel)  # Dùng thuật toán tự viết
        method = "Manual (Custom)"
    elif mode == "opencv":
        operations = apply_opencv(img, kernel)  # Dùng OpenCV
        method = "OpenCV"
    else:
        print("Error: Invalid mode. Choose 'manual' or 'opencv'.")
        sys.exit(1)

    exec_time = time.time() - start_time  # Thời gian thực thi

    # Nếu chỉ thực hiện một phép toán cụ thể
    if mor_op:
        if mor_op in operations:
            img_out = operations[mor_op]
            cv2.imshow(f"Result: {mor_op} - {method}", img_out * 255)  # Hiển thị kết quả
            cv2.imwrite(out_file, img_out * 255)  # Lưu ảnh kết quả
            cv2.waitKey(wait_key_time)
            print(f"Output saved to {out_file}")
            print(f"Time Complexity ({method}): {exec_time:.6f} seconds")
        else:
            print(f"Error: Unknown morphological operation '{mor_op}'")
    else:
        # Nếu không chọn phép toán cụ thể, hiển thị tất cả kết quả trên một ảnh
        rows, cols = 2, 4  # Xếp lưới 2 hàng 4 cột
        images = list(operations.values())  # Lấy danh sách ảnh kết quả
        labels = list(operations.keys())  # Nhãn cho từng ảnh

        h, w = images[0].shape  # Kích thước ảnh
        label_height = 30  # Kích thước vùng hiển thị nhãn
        grid_img = np.ones((h * rows + label_height * rows, w * cols), dtype=np.uint8) * 255  # Tạo ảnh nền trắng
```

```python
        # Ghép ảnh vào lưới
        for idx, (label, img) in enumerate(zip(labels, images)):
            row, col = divmod(idx, cols)  # Xác định vị trí trong lưới
            y_start = row * (h + label_height)
            y_end = y_start + h
            x_start = col * w
            x_end = x_start + w

            grid_img[y_start:y_end, x_start:x_end] = img * 255  # Đưa ảnh vào grid
            cv2.putText(grid_img, label, (x_start + 5, y_end + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.6, 0, 2)  # Ghi nhãn

        cv2.imshow(f"All Morphological Operations - {method}", grid_img)  # Hiển thị ảnh tổng hợp
        cv2.imwrite(out_file, grid_img)  # Lưu ảnh tổng hợp
        cv2.waitKey(wait_key_time)
        print(f"All operations saved to {out_file}")
        print(f"Time Complexity ({method}): {exec_time:.6f} seconds")
```

```python
def main(argv):
    """Xử lý đầu vào từ dòng lệnh."""
    input_file = ''
    output_file = ''
    mor_op = ''
    mode = 'manual'  # Mặc định dùng thuật toán thủ công
    wait_key_time = 0

    description = 'Usage: main.py -i <input_file> -o <output_file> [-p <morph_operator>] -m <mode> -t <wait_key_time>'

    try:
        opts, args = getopt.getopt(argv, "hi:o:p:m:t:", ["in_file=", "out_file=", "mor_operator=", "mode=", "wait_key_time="])
    except getopt.GetoptError:
        print(description)
        sys.exit(2)

    for opt, arg in opts:
        if opt == '-h':
            print(description)
            sys.exit()
        elif opt in ("-i", "--in_file"):
            input_file = arg
        elif opt in ("-o", "--out_file"):
            output_file = arg
        elif opt in ("-p", "--mor_operator"):
            mor_op = arg
        elif opt in ("-m", "--mode"):
            mode = arg.lower()  # Chuyển về chữ thường
        elif opt in ("-t", "--wait_key_time"):
            wait_key_time = int(arg)

    if not input_file or not output_file:
        print("Error: Missing required arguments.")
        print(description)
        sys.exit(1)
```

```python
    operator(input_file, output_file, mor_op, mode, wait_key_time)

if __name__ == "__main__":
    main(sys.argv[1:])
```

III. Summarization of the usage

Detailed Usage and Algorithm Explanation:

1. pad_image(img, kernel)

- **Usage:** Adds padding to the image to prevent overflow errors when applying morphological operations. The padding size is determined by the kernel's shape.
- **Algorithm Explanation:**
  - The function calculates the padding required for height (pad_h) and width (pad_w) based on the kernel size.
  - It then uses np.pad() to add zero-padding to the image around the borders.
  - This ensures that the kernel fits within the image boundaries during operations.

2. erode(img, kernel)

- **Definition**

$$X \Theta B = \{ p \in \varepsilon^2 : p + b \in X, \forall b \in B \}$$
$$X \Theta B = \{ p \in \varepsilon^2 : (B)_p \subseteq X \}$$
$$X \Theta B = \bigcap_{b \in B} X_{-b}$$

- **Usage:** Performs erosion on a binary image. Erosion shrinks bright regions by removing pixels at the borders of bright regions.
- **Algorithm Explanation:**
  - The function first pads the image to prevent boundary errors during erosion.
  - It then iterates over each pixel of the image.
  - For each pixel, it extracts a region equal to the kernel's size.
  - The region is compared with the kernel, and if it matches, the pixel is set to 1 in the result.
  - Otherwise, the pixel remains 0, thus shrinking bright areas.

3. dilate(img, kernel)

- **Definition**

$$X \oplus B = \{ p \in \varepsilon^2 : p = x + b, x \in X \text{ and } b \in B \}$$
$$X \oplus B = \{ p \in \varepsilon^2 : (\hat{B})_p \cap X \neq \varnothing \}$$
$$X \oplus B = \bigcup_{b \in B} X_b$$

- **Usage:** Performs dilation on a binary image. Dilation expands bright regions by adding pixels to the borders of bright regions.
- **Algorithm Explanation:**
  - The image is padded to handle boundary issues.
  - The function iterates through each pixel, extracting a region the size of the kernel.
  - If any pixel in the region is 1 (according to the kernel), the center pixel in the result is set to 1, thereby expanding bright regions.

4. opening(img, kernel)

- **Definition**

$$X \circ B = (X \Theta B) \oplus B$$
$$(X \circ B = \bigcup \{ (B)_p \mid (B)_p \subseteq X \})$$

- **Usage:** Performs the opening operation, which is a sequence of erosion followed by dilation. It is used to remove small noise or isolate small bright regions.
- **Algorithm Explanation:**
  - First, erosion is applied to remove small bright regions.
  - Then, dilation is applied to restore the remaining regions while keeping small noise removed.

- The result is smoother and cleaner, especially for small objects.

5. closing(img, kernel)

- **Definition**

$$X \bullet B = (X \oplus B) \Theta B$$

$$X \bullet B = \{w \in \varepsilon^2 : (B)_p \bigcap X \neq \emptyset, w \in (B)_p\}$$

- **Usage:** Performs the closing operation, which is a sequence of dilation followed by erosion. It is used to close small holes or gaps in bright regions.
- **Algorithm Explanation:**
  - First, dilation is applied to expand bright regions and fill small holes.
  - Then, erosion is applied to restore the expanded regions, closing any gaps or small holes in the bright areas.

6. hit_or_miss(img, kernel)

Definition

$$B = (B_1, B_2)$$

$$B_1 = A \quad and \quad B_2 = W - A$$

$$X \otimes B = (X \Theta B_1) \cap (X^c \Theta B_2)$$

- **Usage:** Performs the Hit-or-Miss operation to detect specific shapes or patterns in the image.
- **Algorithm Explanation:**
  - The kernel is divided into two parts: the foreground (where the kernel has 1s) and the background (where the kernel has -1s).
  - The function erodes the original image with the foreground kernel and the complement of the image with the background kernel.
  - The result is a logical AND between the two eroded images, producing regions where both conditions are satisfied (i.e., the specific shape is found).

7. boundary_extraction(img, kernel)

Definition

$$\beta(A) = A - (A \Theta B)$$

- **Usage:** Extracts the boundaries of bright regions in the image.
- **Algorithm Explanation:**
  - The function performs erosion on the image and subtracts the eroded image from the original image.
  - The result is the boundary, or the outer edges, of the bright regions in the image.

8. region_filling(img, kernel, seed)

Algorithm

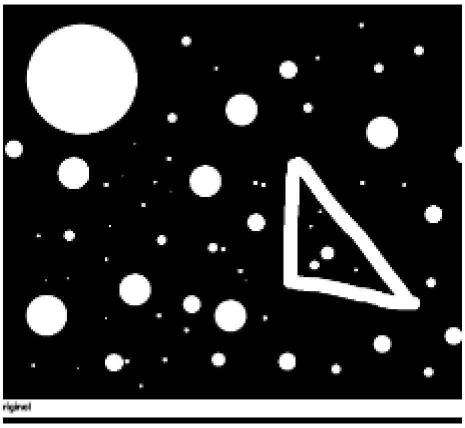
$$X_0 = p \quad (inside\ boundary)$$
$$X_k = (X_{k-1} \oplus B) \cap A^c, k = 1,2,3,\ldots$$
$$Stop\ if\ X_k = X_{k-1}$$

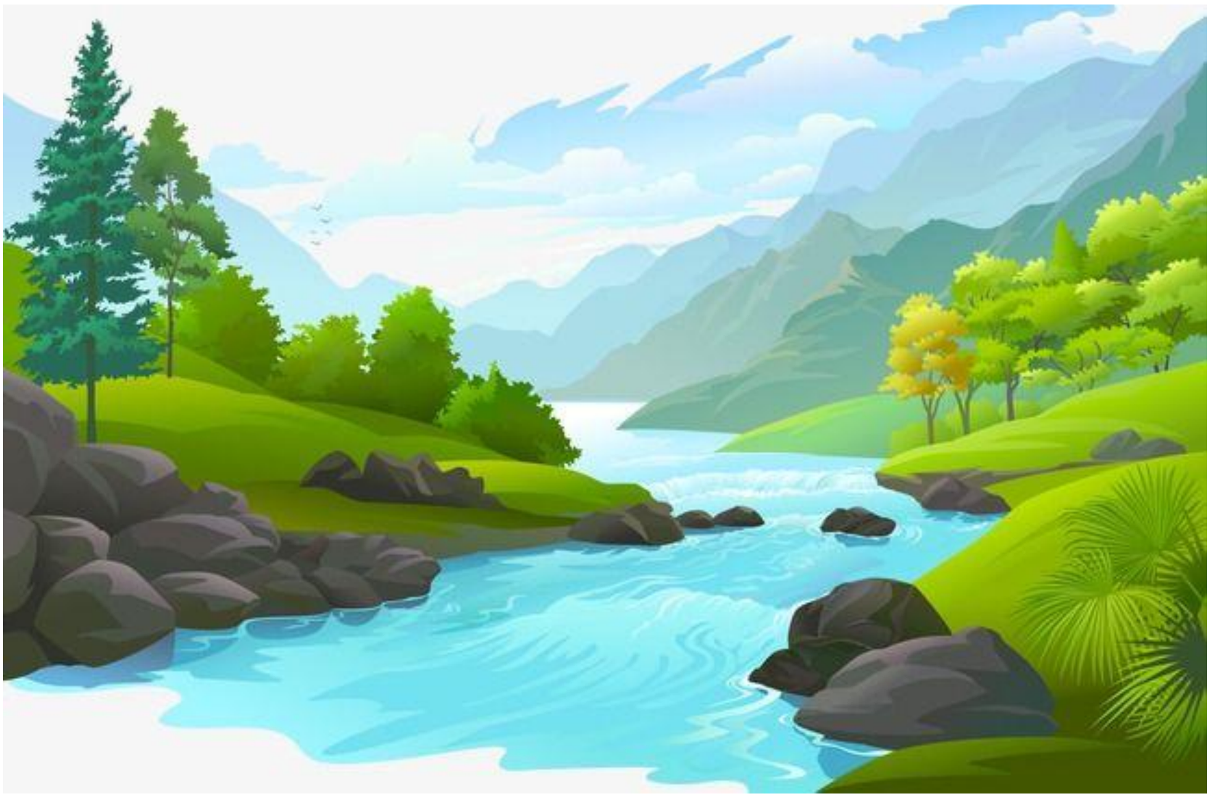
- **Usage:** Fills regions of bright areas starting from a seed pixel. This operation is useful for filling small holes or gaps within a bright region.
- **Algorithm Explanation:**
    - The function initializes a result image with all pixels set to 0 except for the seed pixel, which is set to 1.
    - It performs dilation iteratively on the result image, expanding the region starting from the seed, but only filling into areas that are part of the background.
    - The process continues until no changes occur between iterations, indicating that the entire region has been filled.

IV. EXPERIMENTS AND EVALUATION:

Test images: The algorithm on a test set consisting of images with different sizes, brightness levels, colors, structure and complexities as below:


Discover

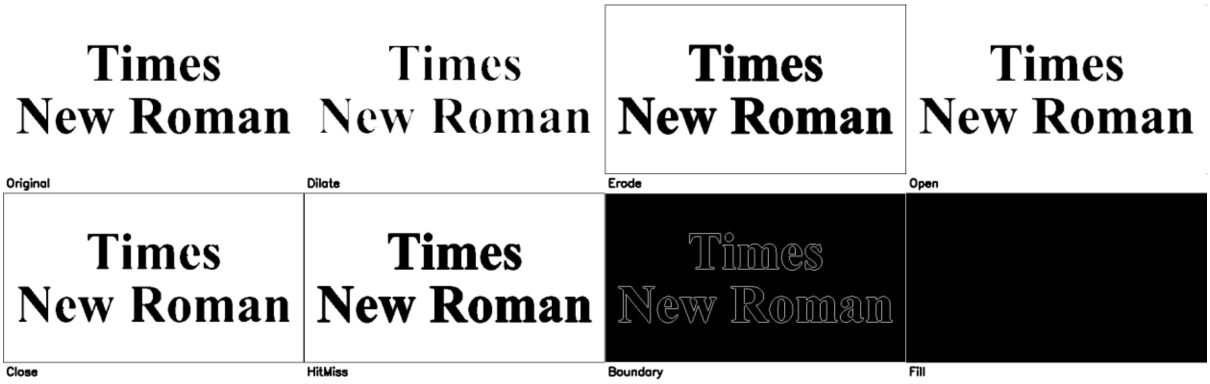




Results with opencv mode vs manual mode:

Text_tnr.png

Execution Time (Manual (Custom)): 15.685021 seconds



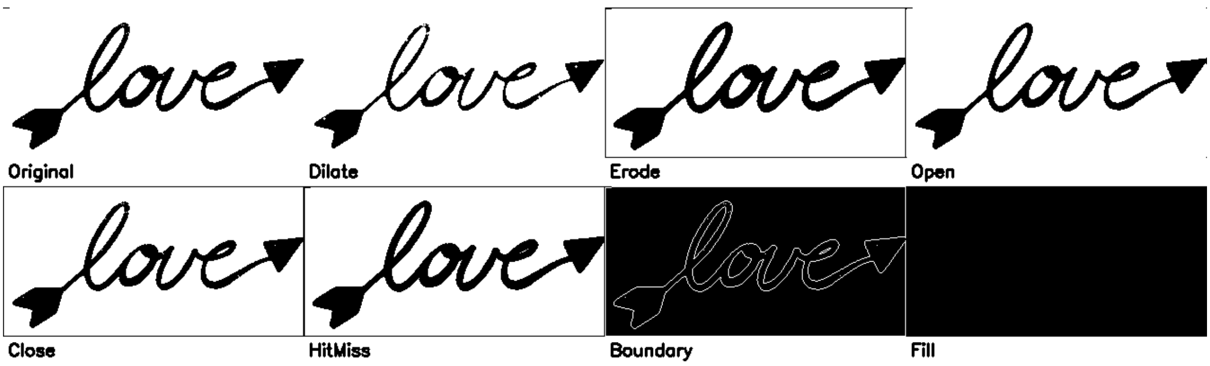
Execution Time (OpenCV): 0.002056 seconds

Comments:

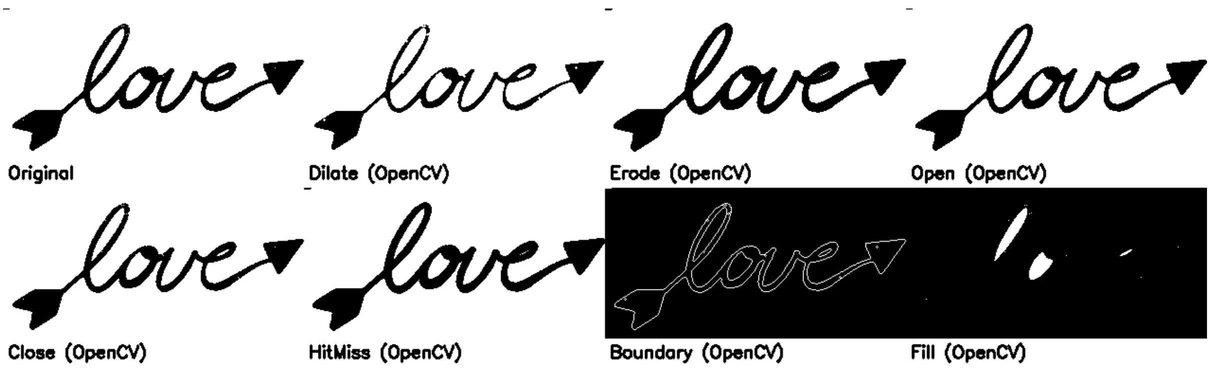
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Lover.png

Execution Time (Manual (Custom)): 6.485807 seconds



Execution Time (OpenCV): 0.000000 seconds

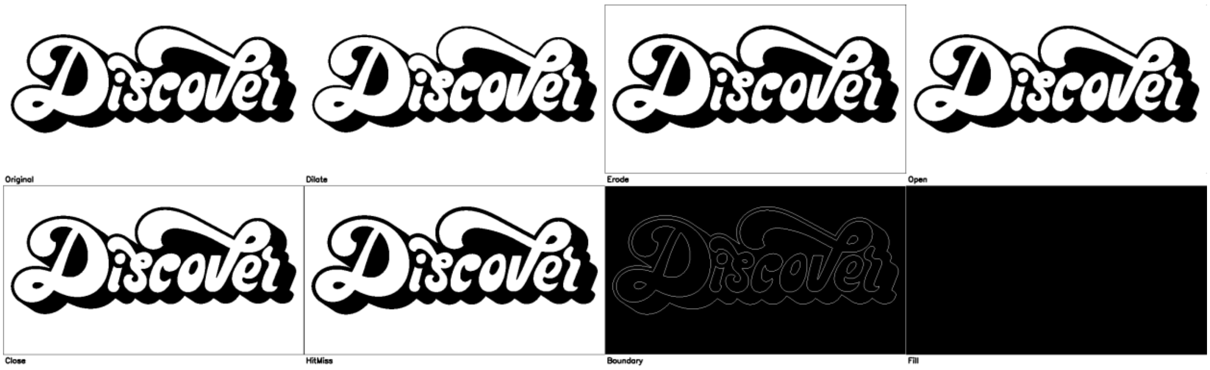


Comments:

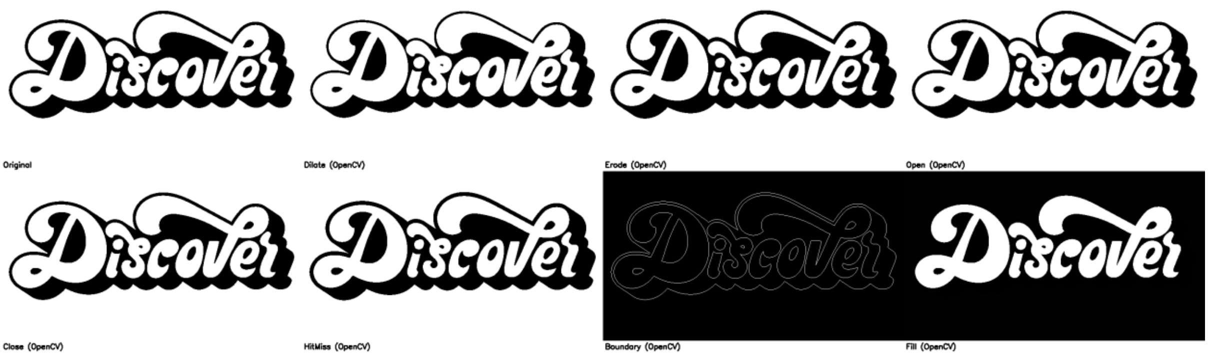
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Discover.jpg

Execution Time (Manual (Custom)): 30.812511 seconds



Execution Time (OpenCV): 0.004053 seconds

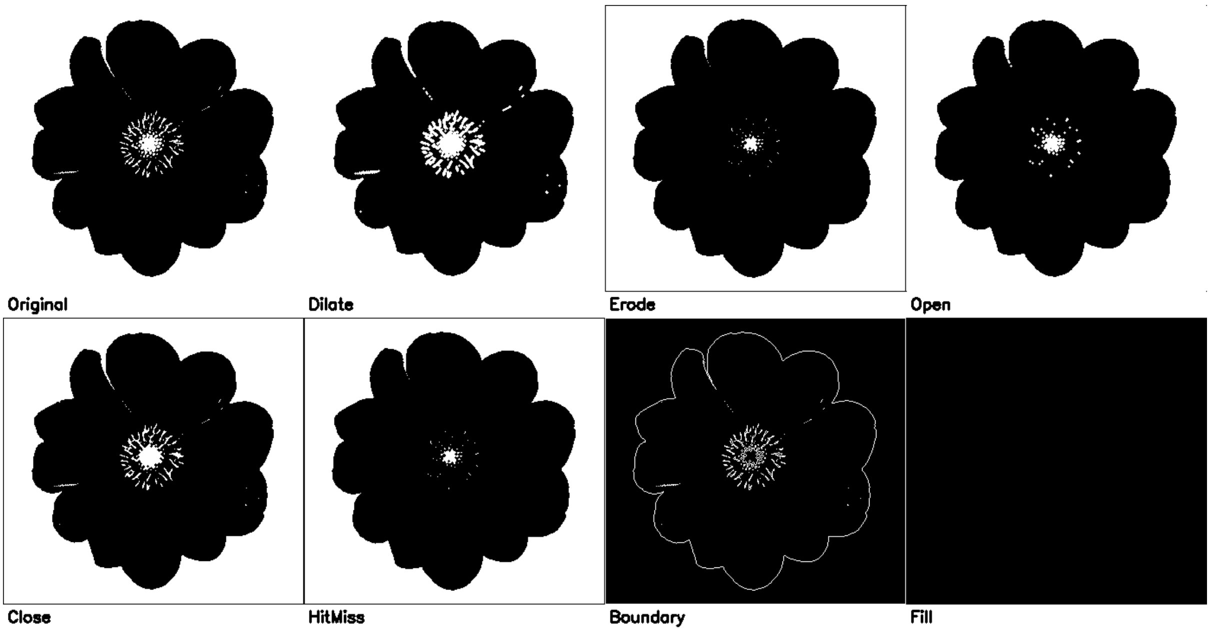


Comments:

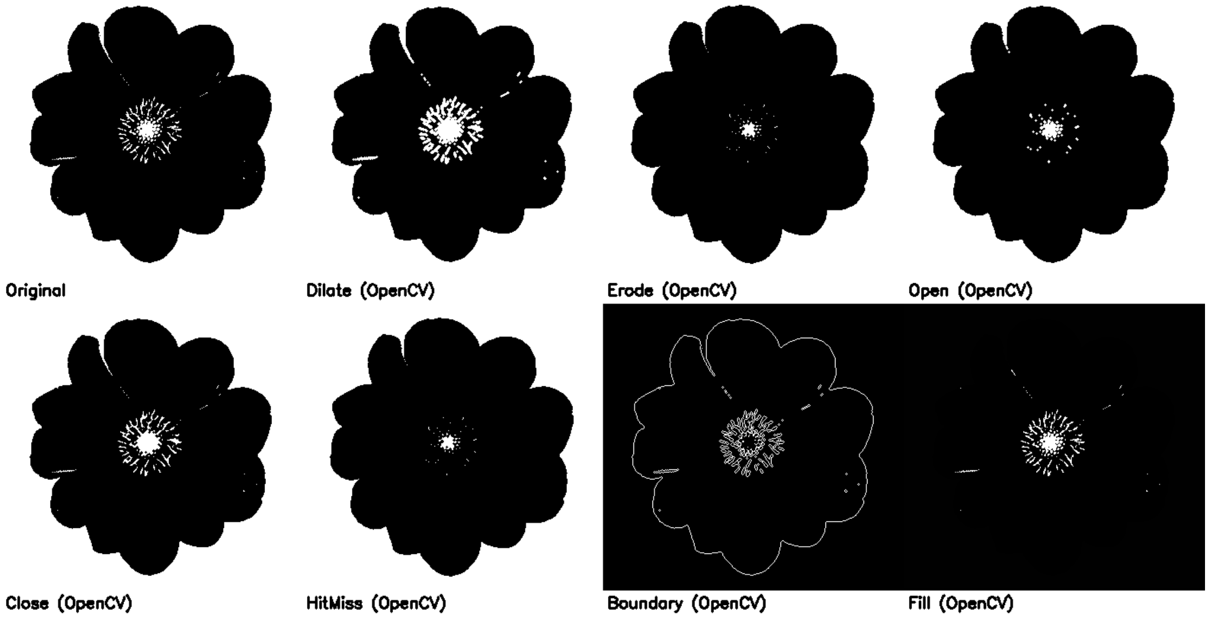
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

<u>Flower.png</u>

Execution Time (Manual (Custom)): 9.767103 seconds



Execution Time (opencv): 0.016066 seconds

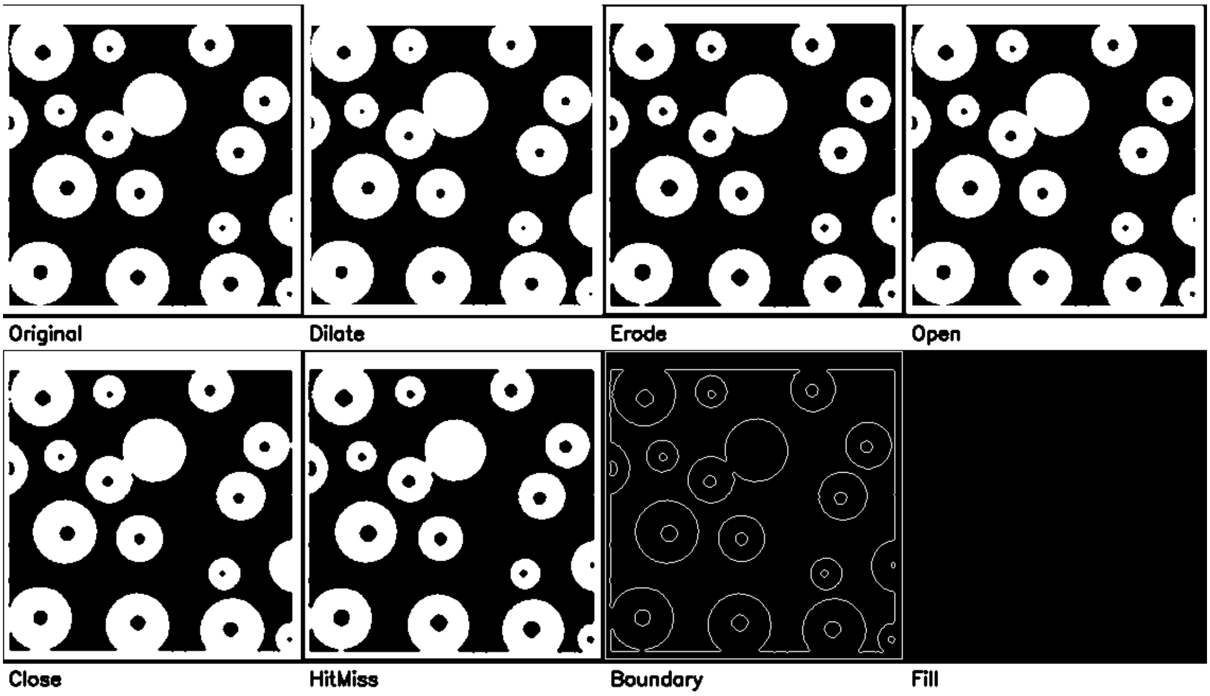


Comments:

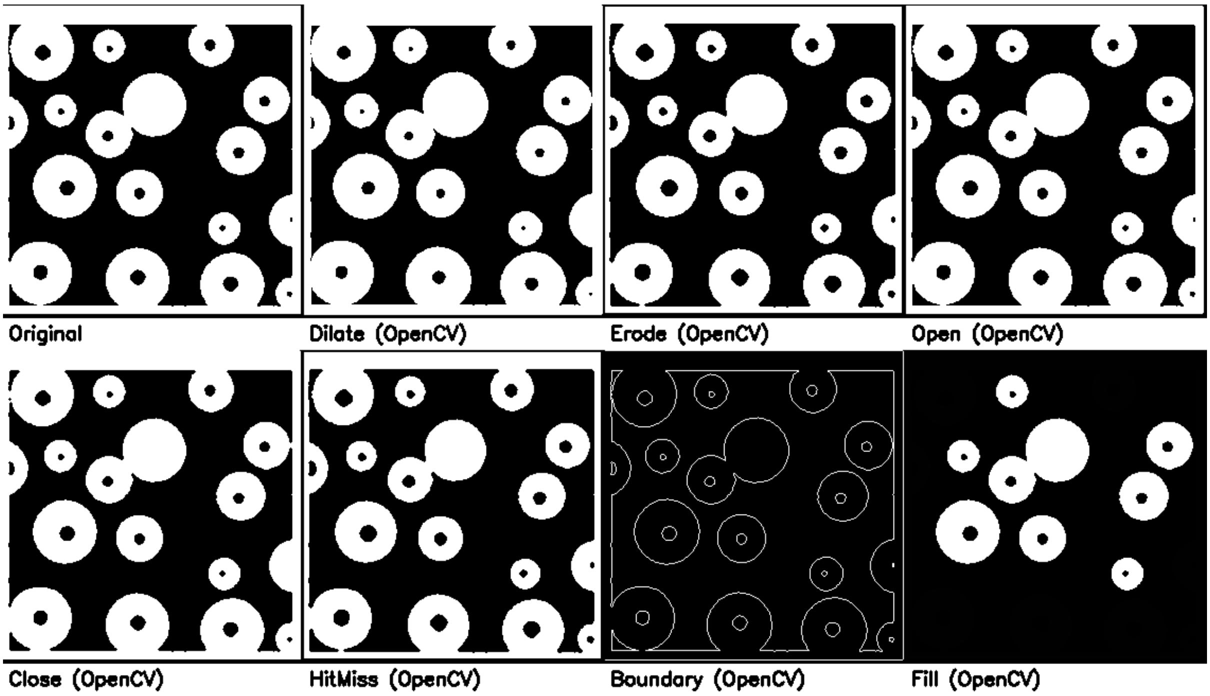
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation. Boundary operator (openCV) mode is more detailed than the custom version

<u>Circle.png</u>

Execution Time (manual): 7.076829 seconds



Execution Time (opencv): 0.016066 seconds

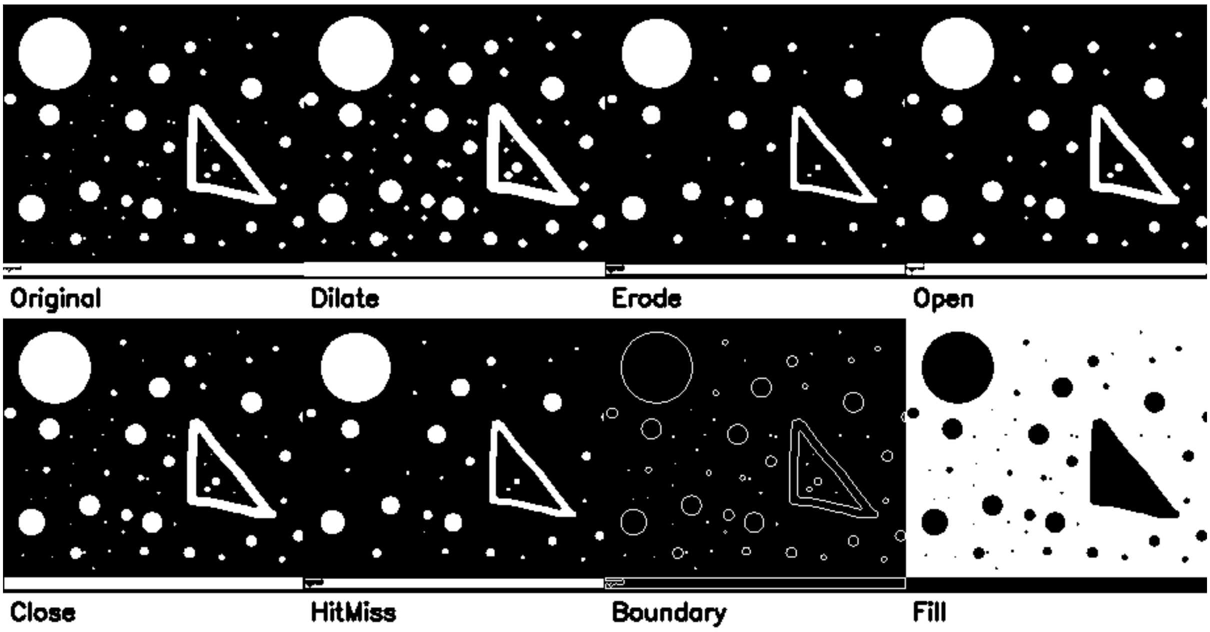


Comments:

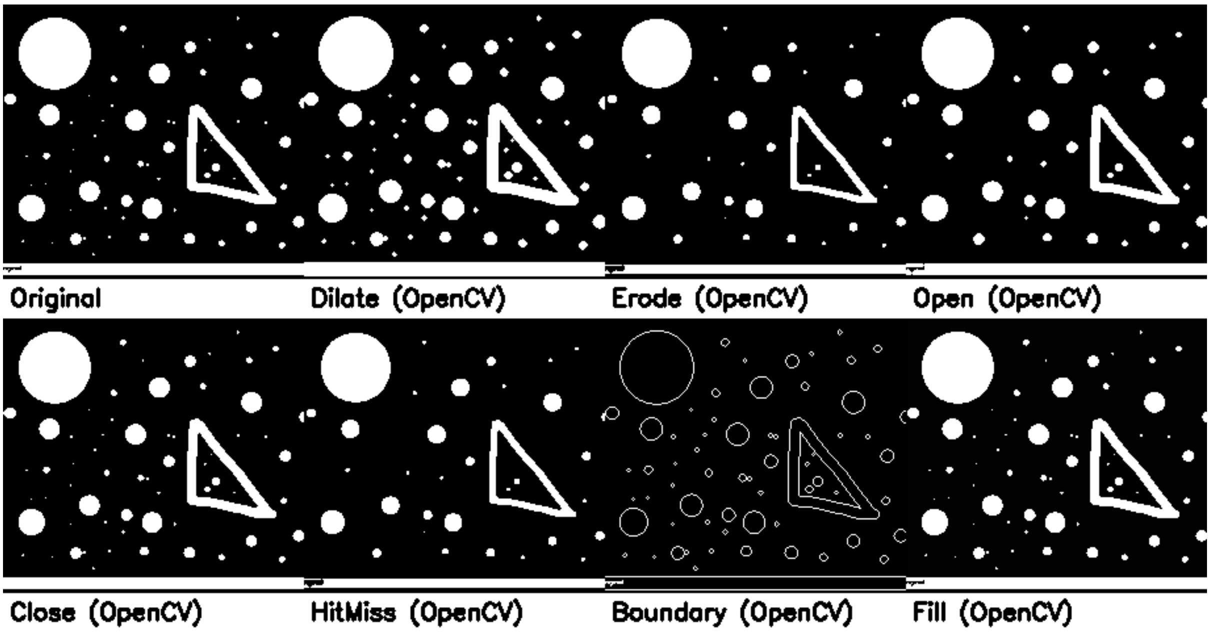
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Star.png

Execution Time (manual): 299.239042 seconds



Execution Time (opencv): 0.000000 seconds



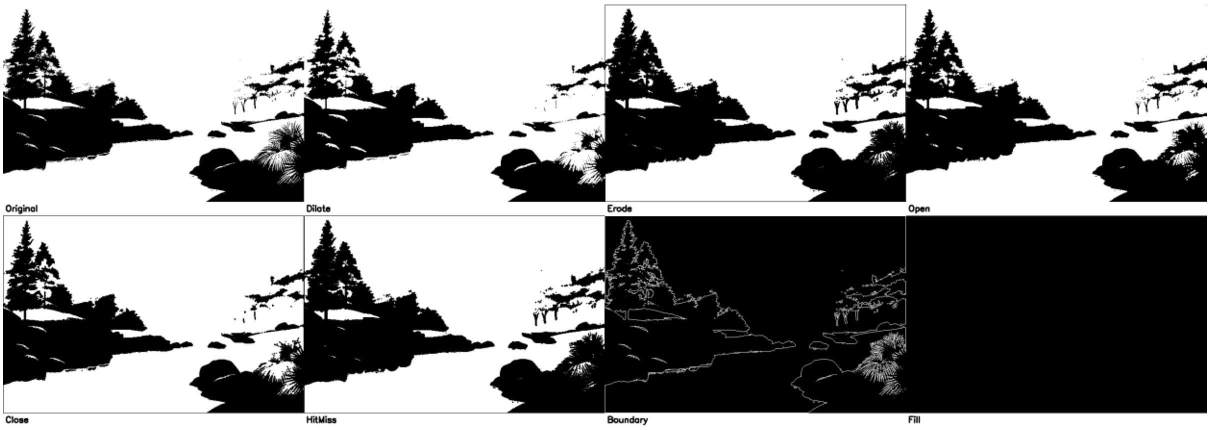
Comments:

Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation. Boundary operator (openCV) mode is more detailed than the custom version.

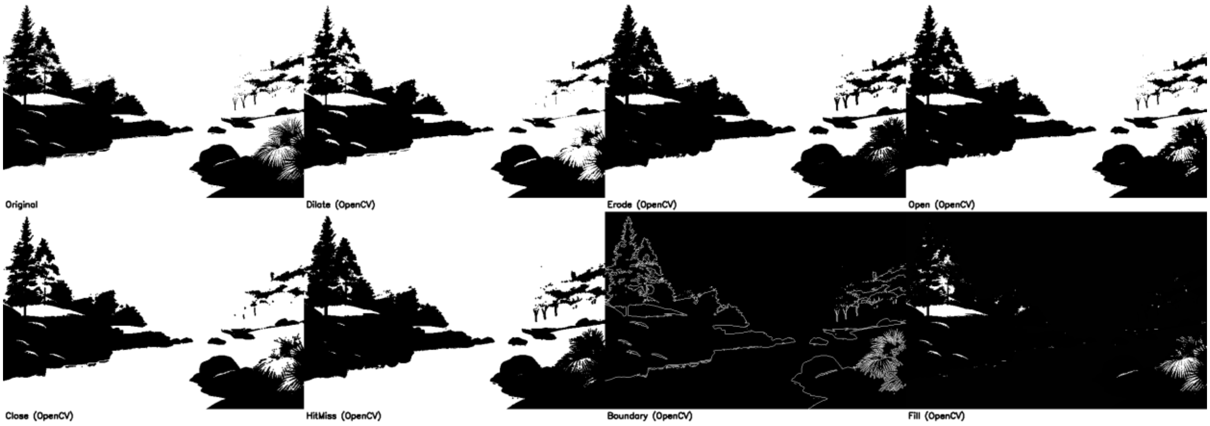
Region filling (manual) is not as expected, wrong region.

Landscape.png

Execution Time (manual): 24.075548 seconds



Execution Time (opencv): 0.005046 seconds



Comments:

Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.
Boundary operator (openCV) mode is more detailed than the custom version

Comparison in general

Aspect	Custom Implementation	OpenCV Implementation
Performance (Speed)	Slow (nested loops, sequential processing)	Fast (optimized C++ backend, SIMD, parallel processing)
Accuracy	Can be prone to errors (padding issues, structuring element misalignment)	Highly reliable and precise
Dilation & Erosion	Works if implemented correctly but slower	Optimized and fast
Opening & Closing	Affected by dilation/erosion accuracy	Standardized and efficient
Hit-or-Miss	May introduce errors if background processing is incorrect	Handles structuring elements correctly
Boundary Extraction	Accurate if erosion is properly done	Faster with correct results
Region Filling	Uses iterative dilation, slow for complex shapes	Uses flood fill, faster and more memory-efficient