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 after week of 14 Feb 2025, binary and grayscale image)

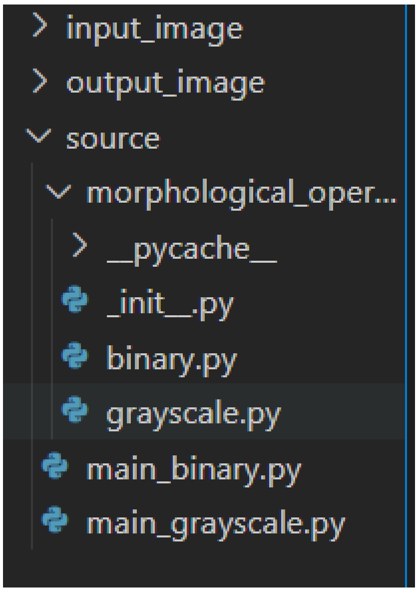
I. Evaluation summary:

No	Types	Task	Implemetation (Without OpenCV)	Implementation (With OpenCV)	Completion (%)
1	**Binary image**	**Binary Dilation**	Implemented using manual convolution and max filter.	Uses cv2.dilate() with a structuring element.	100%
2		**Binary Erosion**	Implemented using manual convolution and min filter.	Uses cv2.morphologyEx(..., cv2.MORPH_OPEN).	100%
3		**Binary Opening**	Combines manual erosion and dilation sequentially	Uses cv2.morphologyEx(..., cv2.MORPH_OPEN).	100%
4		**Binary Closing**	Combines manual dilation and erosion sequentially.	Uses cv2.morphologyEx(..., cv2.MORPH_CLOSE).	100%
5		**Hit-or-Miss**	Implemented using two structuring elements and logical operations.	Uses cv2.morphologyEx(..., cv2.MORPH_HITMISS).	100%
6		**Boundary Extraction**	Subtracts eroded image from the original manually.	Uses cv2.subtract(image, cv2.erode(image, kernel)).	100%
7		**Region Filling**	Uses iterative processing with a seed-based algorithm.	Uses OpenCV flood-fill (cv2.floodFill()).	100%
8		**Extraction of Connected Components**	Implemented using labeling algorithms (BFS)	Uses cv2.connectedComponents()	100%
9		**Convex Hull**	Implemented using Jarvis's March algorithm	Uses cv2.convexHull()	100%
10		**Thinning**	Implemented using iterative Zhang-Suen thinning algorithm	Uses cv2.ximgproc.thinning()	100%

11		**Thickening**	Implemented using dilation followed by intersection with original image		100%
12		**Skeleton**	Implemented using iterative morphological thinning until convergence		100%
13		**Reconstruction**	Implemented using marker-based morphological dilation		100%
14		**Pruning**	Implemented using skeletonization and branch-point removal		100%
1	**Gray scale**	**Grayscale Dilation**	Implemented using max filter over neighborhood	Uses cv2.dilate()	100%
2		**Grayscale Erosion**	Implemented using min filter over neighborhood	Uses cv2.erode()	100%
3		**Grayscale Opening**	Implemented using grayscale erosion followed by dilation	Uses cv2.morphologyEx(…, cv2.MORPH_OPEN)	100%
4		**Grayscale Closing**	Implemented using grayscale dilation followed by erosion	Uses cv2.morphologyEx(…, cv2.MORPH_CLOSE)	100%
5		**Grayscale smoothing**	Implemented using Gaussian or median filtering		100%
6		**Grayscale Morphology Gradient**	Implemented using difference between dilation and erosion	Uses cv2.morphologyEx(…, cv2.MORPH_GRADIENT)	100%
7		**Top-hat transformation**	Implemented using difference between original image and opened image	Uses cv2.morphologyEx(…, cv2.MORPH_TOPHAT)	100%
8		**Textural segmentation**	Implemented using morphological filtering and thresholding		100%
9		**Granulometry**	Implemented using morphological opening with increasing kernel sizes		100%
10		**Reconstruction**	Implemented using marker-controlled dilation until convergence		100%

II. List of features and file structure:

File structure:



input_image/ – Stores input images.

output_image/ – Contains processed output images.

source/ – Main source code directory.

- **morphological_operations/** – Contains morphological processing modules.
    - **binary.py** – Handles binary image operations.
    - **grayscale.py** – Handles grayscale image operations.
- **main_binary.py –** Executes binary image processing.
- **main_grayscale.py** – Executes grayscale image processing.

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.
- **connected_components(img)** – Identifies and labels connected components in a binary image.

- **convex_hull(img)** – Computes the convex hull of bright regions using the Jarvis March algorithm.
- **thinning(img)** – Thins objects in the image using the Zhang-Suen thinning algorithm.
- **thickening(img, kernel)** – Thickens objects by applying conditional dilation.
- **skeletonization(img)** – Extracts the skeleton of objects via iterative erosion.
- **reconstruction(marker, mask, kernel)** – Reconstructs an image using dilation with constraints.
- **pruning(skeleton, iterations)** – Refines a skeleton by removing endpoints iteratively.

2. grayscale.py

Main functions:
• **pad_image()** – Adds padding to avoid processing errors.
• **erode()** – Applies grayscale erosion to darken bright regions.
• **dilate()** – Applies grayscale dilation to brighten dark regions.
• **opening()** – Performs grayscale opening (Erosion → Dilation) to remove small bright noise.
• **closing()** – Performs grayscale closing (Dilation → Erosion) to fill small dark gaps.
• **gradient()** – Computes the morphological gradient (Dilation - Erosion) to highlight edges.
• **top_hat()** – Extracts small bright details using the Top-Hat transformation (Original - Opening).
• **black_hat()** – Extracts small dark details using the Black-Hat transformation (Closing - Original).
• **reconstruction(marker, mask, kernel)** – Restores an image using morphological reconstruction, applying both dilation (for bright regions) and erosion (for dark regions).
• **watershed_segmentation(img, markers)** – Segments an image using the watershed algorithm.

3. main_binary.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, Hit-or-Miss, Boundary Extraction, Region Filling, Connected Components, Convex Hull, Thinning, Thickening, Skeletonization, Reconstruction, Pruning.
- 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.

4. main_grayscale.py

Main functions:

- Read the input image and process it as a grayscale image.
- Provide two processing modes:
  - **Manual:** Use custom algorithms from **grayscale.py.**
  - **OpenCV:** Use the OpenCV library for processing.
- Support operations: Dilate, Erode, Open, Close, Gradient, TopHat, BlackHat, Reconstruction, Thinning, Thickening
- Display and save the processed image.
- Measure and display the execution time of each method.

Details:

- **apply_manual(img, kernel)** – Applies custom morphological operations for grayscale images.
- **apply_opencv(img, kernel)** – Applies grayscale morphological operations using OpenCV.
- **operator(in_file, out_file, mor_op, mode, wait_key_time=0)** – Processes the image with the specified operation and mode, displays and saves results, and 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
```

```python
# 7 Thuật toán bổ sung

from collections import deque

def connected_components(img):
    """Tách các thành phần liên thông trong ảnh nhị phân."""
    h, w = img.shape
    labels = np.zeros(img.shape, dtype=np.int32)  # Định dạng số nguyên để chứa nhiều nhãn
    label = 1
    directions = [(-1,0), (1,0), (0,-1), (0,1), (-1,-1), (-1,1), (1,-1), (1,1)]

    for i in range(h):
        for j in range(w):
            if img[i, j] == 1 and labels[i, j] == 0:
                queue = deque([(i, j)])
                while queue:
                    x, y = queue.popleft()
                    if labels[x, y] == 0:
                        labels[x, y] = label
                        for dx, dy in directions:
                            nx, ny = x + dx, y + dy
                            if 0 <= nx < h and 0 <= ny < w and img[nx, ny] == 1 and labels[nx, ny] == 0:
                                queue.append((nx, ny))
                label += 1
    return labels
```

source > morphological_operator > binary.py > connected_components

```python
def convex_hull(img):
    """Tính toán bao lồi của một vùng sáng trong ảnh bằng thuật toán Jarvis March (Gift Wrapping)."""
    points = np.argwhere(img == 1)  # Lấy tất cả điểm có giá trị 1
    if len(points) == 0:
        return img  # Nếu không có điểm nào, trả về ảnh ban đầu

    def cross_product(o, a, b):
        """Tích có hướng để kiểm tra điểm nằm bên trái hay phải đường oa."""
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Tìm điểm có tọa độ (y, x) nhỏ nhất -> điểm dưới cùng bên trái
    start = points[np.argmin(points[:, 1])]
    hull = [start]

    while True:
        candidate = None
        for p in points:
            if np.array_equal(p, hull[-1]):  # Bỏ qua chính điểm hiện tại
                continue
            if candidate is None or cross_product(hull[-1], candidate, p) < 0:
                candidate = p
            elif cross_product(hull[-1], candidate, p) == 0:
                # Nếu ba điểm thẳng hàng, chọn điểm xa hơn
                if np.linalg.norm(p - hull[-1]) > np.linalg.norm(candidate - hull[-1]):
                    candidate = p
        if np.array_equal(candidate, start):  # Nếu quay lại điểm đầu -> kết thúc
            break
        hull.append(candidate)

    # Tạo ảnh chứa bao lồi
    hull_img = np.zeros_like(img)
    for p in hull:
        hull_img[p[0], p[1]] = 1

    return hull_img
```

Grayscale.py

source > morphological_operator > grayscale.py > ...

```python
import numpy as np

def pad_image_grayscale(img, kernel, pad_value):
    """Thêm padding với giá trị tùy theo phép toán (0 cho dilation, 255 cho erosion)."""
    pad_h, pad_w = kernel.shape[0] // 2, kernel.shape[1] // 2
    return np.pad(img, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant', constant_values=pad_value)

def grayscale_dilation(img, kernel):
    """Grayscale Dilation: (f ⊕ b)(s,t) = max{f(s-x, t-y) + b(x,y)}"""
    padded_img = pad_image_grayscale(img, kernel, 0)  # Pad với 0
    result = np.zeros_like(img, dtype=np.uint8)
    kh, kw = kernel.shape

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            region = padded_img[i:i+kh, j:j+kw]
            result[i, j] = np.clip(np.max(region + kernel), 0, 255)  # Giữ giá trị trong [0, 255]

    return result
```

source > morphological_operator > grayscale.py > grayscale_dilation

```python
def grayscale_erosion(img, kernel):
    """Grayscale Erosion: (f ⊖ b)(s,t) = min{f(s+x, t+y) - b(x,y)}"""
    padded_img = pad_image_grayscale(img, kernel, 255)  # Pad với 255 cho erosion
    result = np.zeros_like(img, dtype=np.uint8)
    kh, kw = kernel.shape

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            region = padded_img[i:i+kh, j:j+kw]
            result[i, j] = np.clip(np.min(region - kernel), 0, 255)  # Giữ giá trị trong

    return result



def grayscale_opening(img, kernel):
    """Grayscale Opening: f ∘ b = (f ⊖ b) ⊕ b"""
    return grayscale_dilation(grayscale_erosion(img, kernel), kernel)

def grayscale_closing(img, kernel):
    """Grayscale Closing: f • b = (f ⊕ b) ⊖ b"""
    return grayscale_erosion(grayscale_dilation(img, kernel), kernel)
```

source > morphological_operator > grayscale.py > ...

```python
def grayscale_smoothing(img, kernel):
    """Grayscale Smoothing: Áp dụng Opening rồi Closing để làm mịn ảnh."""
    return grayscale_closing(grayscale_opening(img, kernel), kernel)

def grayscale_morphology_gradient(img, kernel):
    """Grayscale Morphology Gradient: (f ⊕ b) - (f ⊖ b)"""
    dilated = grayscale_dilation(img, kernel)
    eroded = grayscale_erosion(img, kernel)
    return np.clip(dilated.astype(np.int16) - eroded.astype(np.int16), 0, 255).astype(np.uint8)

def top_hat(img, kernel):
    """Top-hat Transformation: f - (f ∘ b)"""
    opened = grayscale_opening(img, kernel)
    return np.clip(img.astype(np.int16) - opened.astype(np.int16), 0, 255).astype(np.uint8)

def textural_segmentation(img, kernel):
    """Textural Segmentation: Dùng Top-hat để phân đoạn kết cấu"""
    return top_hat(img, kernel)
```

source > morphological_operator > grayscale.py > top_hat

```python
def granulometry(img, sizes):
    """Granulometry: Tính tổng giá trị pixel sau Opening với các kích thước kernel khác nhau"""
    granulometry_result = []
    for size in sizes:
        kernel = np.ones((size, size), dtype=np.uint8)
        opened = grayscale_opening(img, kernel)
        granulometry_result.append(np.sum(opened))
    return granulometry_result

def reconstruction(marker, mask, kernel, max_iter=100):
    """Morphological Reconstruction by Dilation: R_g^D(f)"""
    prev_marker = np.zeros_like(marker)
    iter_count = 0

    while not np.array_equal(marker, prev_marker):
        prev_marker = marker.copy()
        marker = np.minimum(grayscale_dilation(marker, kernel), mask)
        iter_count += 1
        if iter_count >= max_iter:
            print("Warning: Reconstruction reached max iterations!")
            break

    return np.clip(marker, 0, 255).astype(np.uint8)
```

source > morphological_operator > grayscale.py > reconstruction

```python

def reconstruction(marker, mask, kernel, max_iter=100):
    """Morphological Reconstruction by Dilation: R_g^D(f)"""
    prev_marker = np.zeros_like(marker)
    iter_count = 0

    while not np.array_equal(marker, prev_marker):
        prev_marker = marker.copy()
        marker = np.minimum(grayscale_dilation(marker, kernel), mask)
        iter_count += 1
        if iter_count >= max_iter:
            print("Warning: Reconstruction reached max iterations!")
            break

    return np.clip(marker, 0, 255).astype(np.uint8)

def create_structuring_element(size):
    """Tạo phần tử cấu trúc hình vuông"""
    return np.ones((size, size), dtype=np.uint8)

```

source > morphological_operator > binary.py > ...

```python
def convex_hull(img):
    """
    Tính toán bao lồi của một vùng sáng trong ảnh bằng thuật toán Jarvis March (Gift Wrapping).

    Parameters:
        img (numpy.ndarray): Ảnh nhị phân đầu vào (0: nền, 1: vùng sáng).

    Returns:
        numpy.ndarray: Ảnh nhị phân có đường bao lồi.
    """

    # Lấy danh sách tọa độ các điểm có giá trị 1 (điểm thuộc vùng sáng)
    points = np.argwhere(img == 1)

    # Nếu ảnh không có điểm sáng nào, trả về ảnh ban đầu
    if len(points) == 0:
        return img

    def cross_product(o, a, b):
        """
        Tính tích có hướng giữa hai vector oa và ob.
        Giá trị âm -> b nằm bên phải oa (ngược chiều kim đồng hồ).
        Giá trị dương -> b nằm bên trái oa (thuận chiều kim đồng hồ).
        """
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Tìm điểm có tọa độ (y, x) nhỏ nhất -> đây là điểm xuất phát của convex hull
    start = points[np.argmin(points[:, 1])]  # Chọn điểm có x nhỏ nhất (nếu trùng thì chọn y nhỏ nhất)
    hull = [start]  # Danh sách chứa các điểm của convex hull

    while True:
        candidate = None  # Điểm kế tiếp trong convex hull
```

source > morphological_operator > binary.py > convex_hull > cross_product

```python
def convex_hull(img):

    while True:
        candidate = None  # Điểm kế tiếp trong convex hull
        for p in points:
            if np.array_equal(p, hull[-1]):  # Bỏ qua chính điểm hiện tại
                continue

            if candidate is None or cross_product(hull[-1], candidate, p) < 0:
                candidate = p  # Chọn điểm xa nhất theo chiều ngược kim đồng hồ

            elif cross_product(hull[-1], candidate, p) == 0:
                # Nếu ba điểm thẳng hàng, chọn điểm xa hơn
                if np.linalg.norm(p - hull[-1]) > np.linalg.norm(candidate - hull[-1]):
                    candidate = p

        if np.array_equal(candidate, start):  # Nếu quay lại điểm đầu tiên, thuật toán kết thúc
            break

        hull.append(candidate)  # Thêm điểm mới vào convex hull

    # Tạo ảnh nhị phân mới chứa bao lồi
    hull_img = np.zeros_like(img)
    for p in hull:
        hull_img[p[0], p[1]] = 1  # Đánh dấu các điểm thuộc đường bao lồi

    return hull_img
```

source > morphological_operator > binary.py > convex_hull

```python
def thinning(img):
    """Làm mỏng ảnh bằng thuật toán Zhang-Suen."""
    def count_transitions(P):
        """Đếm số lần chuyển từ 0 -> 1 theo thứ tự vòng."""
        P = [P[1,2], P[2,2], P[2,1], P[2,0], P[1,0], P[0,0], P[0,1], P[0,2], P[1,2]]
        return sum((P[i] == 0 and P[i+1] == 1) for i in range(8))

    def step(img, pass_num):
        markers = np.zeros_like(img)
        for i in range(1, img.shape[0] - 1):
            for j in range(1, img.shape[1] - 1):
                P = img[i-1:i+2, j-1:j+2]
                if img[i, j] == 1:
                    neighbors = np.sum(P) - 1
                    transitions = count_transitions(P)
                    cond1 = 2 <= neighbors <= 6
                    cond2 = transitions == 1
                    cond3 = P[0,1] * P[1,2] * P[2,1] == 0 if pass_num == 0 else P[1,2] * P[2,1] * P[1,0] == 0
                    if cond1 and cond2 and cond3:
                        markers[i, j] = 1
        img[markers == 1] = 0

    prev = np.zeros_like(img)
    while not np.array_equal(img, prev):
        prev = img.copy()
        step(img, 0)
        step(img, 1)

    return img
```

source > morphological_operator > binary.py > thinning

```python
def thickening(img, kernel):
    """Làm dày ảnh bằng dilation có điều kiện."""
    complement = np.logical_not(img)
    new_img = dilate(img, kernel)
    return np.logical_and(new_img, complement).astype(np.uint8)


def skeletonization(img):
    """Tạo bộ xương của đối tượng bằng phương pháp lặp erosion."""
    skeleton = np.zeros_like(img)
    temp = img.copy()
    while np.any(temp):
        eroded = erode(temp, np.ones((3, 3), np.uint8))
        skeleton = np.logical_or(skeleton, temp - eroded).astype(np.uint8)
        temp = eroded
    return skeleton


def reconstruction(marker, mask, kernel):
    """Tái tạo ảnh từ một marker bằng phương pháp giãn nở có giới hạn."""
    while True:
        next_marker = np.minimum(dilate(marker, kernel), mask)
        if np.array_equal(next_marker, marker):
            break
        marker = next_marker
    return marker
```

```python
def reconstruction(marker, mask, kernel):
    """Tái tạo ảnh từ một marker bằng phương pháp giãn nở có giới hạn."""
    while True:
        next_marker = np.minimum(dilate(marker, kernel), mask)
        if np.array_equal(next_marker, marker):
            break
        marker = next_marker
    return marker


def pruning(skeleton, iterations=1):
    """Cắt tỉa ảnh bộ xương bằng cách loại bỏ điểm cuối."""
    kernel = np.ones((3, 3), np.uint8)
    for _ in range(iterations):
        endpoints = np.logical_and(skeleton, np.sum(erode(skeleton, kernel), axis=(0, 1)) == 1)
        skeleton = np.logical_and(skeleton, np.logical_not(endpoints)).astype(np.uint8)
    return skeleton

```

Main_binary.py

main_binary.py ✕ | main_grayscale.py | binary.py | image_landscape.png | output_star_manual_all_grayscale.png | grays

source > main_binary.py > operator

```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
# Import trực tiếp các hàm từ binary.py

def apply_manual(img, kernel, seed=(10, 10)):
    print("apply_manual() function called")  # Kiểm tra xem hàm được gọi chưa

    print("Creating operations dictionary...")

    operations = {}
    try:
        print("Adding Original...")
        operations["Original"] = img

        print("Adding Dilate...")
        operations["Dilate"] = binary.dilate(img, kernel)

        print("Adding Erode...")
        operations["Erode"] = binary.erode(img, kernel)

        print("Adding Open...")
        operations["Open"] = binary.opening(img, kernel)

        print("Adding Close...")
        operations["Close"] = binary.closing(img, kernel)

```

source > main_binary.py > apply_manual

```python
def apply_manual(img, kernel, seed=(10, 10)):

        print("Adding HitMiss...")
        operations["HitMiss"] = binary.hit_or_miss(img, kernel)

        print("Adding Boundary...")
        operations["Boundary"] = binary.boundary_extraction(img, kernel)

        print("Adding Fill...")
        operations["Fill"] = binary.region_filling(img, kernel, seed)

        print("Adding ConnectedComponents...")
        operations["ConnectedComponents"] = binary.connected_components(img)

        print("Adding ConvexHull...")
        operations["ConvexHull"] = binary.convex_hull(img)

        print("Adding Thinning...")
        operations["Thinning"] = binary.thinning(img)

        print("Adding Thickening...")
        operations["Thickening"] = binary.thickening(img, kernel)

        print("Adding Skeletonization...")
        operations["Skeletonization"] = binary.skeletonization(img)

        print("Adding Reconstruction...")
        operations["Reconstruction"] = binary.reconstruction(img, img, kernel)

        print("Adding Pruning...")
        operations["Pruning"] = binary.pruning(img)
```

```python
def apply_manual(img, kernel, seed=(10, 10)):
    except Exception as e:
        print(f"Error when adding to operations: {e}")
        sys.exit(1)

    print("Operations dictionary created:", list(operations.keys()))
    return operations

def apply_opencv(img, kernel, seed=(10, 10)):
    """Áp dụng các phép toán hình thái bằng OpenCV."""
    # OpenCV có hỗ trợ một số phép toán như Dilate, Erode, Open, Close, HitMiss
    # ConnectedComponents và Skeletonization cũng được hỗ trợ trong 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, seed, 255)[1],  # Lấp vùng bằng floodFill
        "ConnectedComponents (OpenCV)": cv2.connectedComponents(img)[1],  # Tách thành phần liên thông
        # OpenCV không có hàm trực tiếp cho Skeletonization, nhưng có thể dùng thinning của OpenCV
        "Skeletonization (OpenCV)": cv2.ximgproc.thinning(img * 255) // 255  # Làm mỏng để gần giống bộ xương
    }

def operator(in_file, out_file, mor_op, mode, wait_key_time=0):
    print(f"operator() called with: in_file={in_file}, out_file={out_file}, mor_op={mor_op}, mode={mode}")

    """Thực hiện phép toán hình thái trên ảnh."""
```

source > main_binary.py > apply_opencv

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

    # 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)

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

    # Điểm seed mặc định cho Fill và Reconstruction
    seed = (10, 10)

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


    # Chọn chế độ thực hiện
    print(f"Mode received: {mode}")
    if mode == "manual":
        print("Calling apply_manual()...")
```

source > main_binary.py > operator

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=0):
        print("Calling apply_manual()...")
        operations = apply_manual(img, kernel, seed)
        if operations is None:
            print("Error: apply_manual() returned None")
            sys.exit(1)
        print("Operations dictionary:", list(operations.keys()))
        print("apply_manual() executed successfully.")
        method = "Manual (Custom)"
    elif mode == "opencv":
        operations = apply_opencv(img, kernel, seed)  # 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:
            print(f"Executing {mor_op} using {method}")
            img_out = operations[mor_op]
            # Chuẩn hóa ảnh đầu ra để hiển thị (nếu cần)
            if img_out is None or img_out.size == 0:
                print(f"Error: '{mor_op}' returned an empty result.")
```

source > main_binary.py > operator

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=0):
                sys.exit(1)
            if img_out.dtype != np.uint8 or img_out.max() > 1:
                img_out_display = (img_out / img_out.max() * 255).astype(np.uint8)
            else:
                img_out_display = img_out * 255
            cv2.imshow(f"Result: {mor_op} - {method}", img_out_display)  # Hiển thị kết quả
            cv2.imwrite(out_file, img_out_display)  # 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 = 4, 4  # Tăng lưới để chứa nhiều phép toán hơn (16 slot)
        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

        # Ghép ảnh vào lưới
        for idx, (label, img) in enumerate(zip(labels, images)):
            if idx >= rows * cols:  # Giới hạn số lượng ảnh hiển thị
                break
            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
```

ource > main_binary.py > operator

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=0):
            y_end = y_start + h
            x_start = col * w
            x_end = x_start + w

            # Chuẩn hóa ảnh để hiển thị (nếu cần)
            if img.dtype != np.uint8 or img.max() > 1:
                img_display = (img / img.max() * 255).astype(np.uint8)
            else:
                img_display = img * 255

            grid_img[y_start:y_end, x_start:x_end] = img_display  # 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"Execution Time ({method}): {exec_time:.6f} seconds")

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_t
```

source > main_binary.py > main

```python
def main(argv):
    try:
        opts, args = getopt.getopt(argv, "hi:o:p:m:t:", ["in_file=", "out_file=", "mor_operator=", "mode=", "wait_key_t
    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)

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

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

main_grayscale.py

source > main_grayscale.py > apply_manual

```python
import sys
import getopt
import cv2
import numpy as np
import os
import time
import matplotlib.pyplot as plt
from morphological_operator.grayscale import  pad_image_grayscale, grayscale_dilation, grayscale_erosion, grayscale_oper


def apply_manual(img, kernel, seed=(10, 10)):
    """Áp dụng các phép toán hình thái grayscale bằng thuật toán tự viết."""
    print("apply_manual() function called")  # Kiểm tra xem hàm được gọi chưa

    print("Creating operations dictionary...")
    operations = {}
    sizes = [3, 5, 7]  # Các kích thước kernel cho Granulometry

    try:
        print("Adding Original...")
        operations["Original"] = img

        print("Adding Dilate...")
        operations["Dilate"] = grayscale_dilation(img, kernel)

        print("Adding Erode...")
        operations["Erode"] = grayscale_erosion(img, kernel)


        print("Adding Open...")
        operations["Open"] = grayscale_opening(img, kernel)
```

main_grayscale.py × | image_landscape.png | output_star_manual_all_grayscale.png | grayscale.py | output_landscape_manual_all_g

urce > main_grayscale.py > apply_manual

```python
def apply_manual(img, kernel, seed=(10, 10)):

        print("Adding Erode...")
        operations["Erode"] = grayscale_erosion(img, kernel)


        print("Adding Open...")
        operations["Open"] = grayscale_opening(img, kernel)

        print("Adding Close...")
        operations["Close"] = grayscale_closing(img, kernel)

        print("Adding Smoothing...")
        operations["Smoothing"] = grayscale_smoothing(img, kernel)

        print("Adding Gradient...")
        operations["Gradient"] = grayscale_morphology_gradient(img, kernel)

        print("Adding TopHat...")
        operations["TopHat"] = top_hat(img, kernel)

        print("Adding TexturalSegmentation...")
        operations["TexturalSegmentation"] = textural_segmentation(img, kernel)

        print("Adding Granulometry...")
        operations["Granulometry"] = granulometry(img, sizes)  # Trả về list, cần xử lý riêng khi hiển thị

        print("Adding Reconstruction...")
        operations["Reconstruction"] = reconstruction(img, img, kernel)  # Dùng img làm marker và mask

```

source > main_grayscale.py > apply_manual

```python
def apply_manual(img, kernel, seed=(10, 10)):
    except Exception as e:
        print(f"Error when adding to operations: {e}")
        sys.exit(1)

    print("Operations dictionary created:", list(operations.keys()))
    return operations

def apply_opencv(img, kernel, seed=(10, 10)):
    """Áp dụng các phép toán hình thái grayscale bằng OpenCV."""
    return {
        "Original": img,
        "Dilate": cv2.dilate(img, kernel, iterations=1),
        "Erode": cv2.erode(img, kernel, iterations=1),
        "Open": cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel),
        "Close": cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel),
        "Gradient": cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel),
        "TopHat": cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel),
    }

def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):
    """Thực hiện phép toán hình thái trên ảnh grayscale."""
    print(f"Đang xử lý file đầu vào: {in_file}")

    if not os.path.exists(in_file):
        print(f"Error: Input file '{in_file}' not found.")
        sys.exit(1)
```

source > main_grayscale.py > operator

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):

    img = cv2.imread(in_file, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print(f"Error: Unable to read image file '{in_file}'. Check file format and path.")
        sys.exit(1)
    print(f"Đã đọc ảnh: {img.shape}")

    # Chuẩn hóa ảnh về [0, 1] để phù hợp với tính toán
    img = img.astype(np.uint8)

    kernel = np.ones((3, 3), dtype=np.uint8)
    seed = (10, 10)

    start_time = time.time()

    if mode == "manual":
        operations = apply_manual(img, kernel, seed)
        method = "Manual (Custom)"
    elif mode == "opencv":
        operations = apply_opencv(img, kernel, seed)
        method = "OpenCV"
    else:
        print("Error: Invalid mode. Choose 'manual' or 'opencv'.")
        sys.exit(1)

    exec_time = time.time() - start_time
    mor_op = mor_op.capitalize() if mor_op else ""
    print(f"Danh sách phép toán khả dụng: {list(operations.keys())}")

```

source > main_grayscale.py > operator

```python
def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):
    if mor_op:
        if mor_op in operations:
            result = operations[mor_op]
            if isinstance(result, list):  # Đặc biệt cho Granulometry
                print(f"Granulometry results for sizes [3, 5, 7]: {result}")
                # Không hiển thị ảnh cho Granulometry, chỉ lưu hoặc in kết quả
                np.savetxt(out_file.replace('.png', '.txt'), result, fmt='%.6f')
                print(f"Granulometry results saved to {out_file.replace('.png', '.txt')}")
            else:
                # Chuẩn hóa kết quả để hiển thị
                result_display = (result - result.min()) / (result.max() - result.min()) * 255
                result_display = result_display.astype(np.uint8)
                cv2.imshow(f"Result: {mor_op} - {method}", result_display)
                cv2.imwrite(out_file, result_display)
                cv2.waitKey(wait_key_time)
                cv2.destroyAllWindows()
                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}'")
            print("Available operations:", list(operations.keys()))
            sys.exit(1)
```

```python
source > main_grayscale.py > operator
def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):
    else:
        rows, cols = 2, 5  # Điều chỉnh lưới cho số lượng phép toán phù hợp
        images = [op for key, op in operations.items() if key != "Granulometry"]
        labels = [key for key in operations.keys() if key != "Granulometry"]

        h, w = images[0].shape
        label_height = 30
        grid_img = np.ones((h * rows + label_height * rows, w * cols), dtype=np.uint8) * 255

        for idx, (label, img) in enumerate(zip(labels, images)):
            if idx >= rows * cols:
                break
            row, col = divmod(idx, cols)
            y_start = row * (h + label_height)
            y_end = y_start + h
            x_start = col * w
            x_end = x_start + w

            if img.max() - img.min() == 0:
                img_display = np.zeros_like(img, dtype=np.uint8)  # Gán ảnh về 0 nếu không có sự thay đổi
            else:
                img_display = (img - img.min()) / (img.max() - img.min()) * 255
                img_display = img_display.astype(np.uint8)

            img_display = img_display.astype(np.uint8)
            grid_img[y_start:y_end, x_start:x_end] = img_display
            cv2.putText(grid_img, label, (x_start + 5, y_end + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.6, 0, 2)
```

```python
source > main_grayscale.py > operator
def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):
        cv2.imshow(f"All Morphological Operations - {method}", grid_img)
        cv2.imwrite(out_file, grid_img)
        cv2.waitKey(wait_key_time)
        cv2.destroyAllWindows()
        print(f"All operations saved to {out_file}")
        import matplotlib.pyplot as plt

        if "Granulometry" in operations:
            granulometry_result = operations["Granulometry"]
            sizes = list(range(1, len(granulometry_result) + 1))  # Tạo danh sách kích thước SE

            # Vẽ biểu đồ Granulometry
            plt.figure(figsize=(8, 5))
            plt.plot(sizes, granulometry_result, marker='o', linestyle='-', color='b', label="Granulometry Profile")

            # Thêm tiêu đề và nhãn
            plt.xlabel("Structuring Element Size")
            plt.ylabel("Sum of Pixels After Opening")
            plt.title("Granulometry Analysis")
            plt.legend()
            plt.grid(True)

            # Lưu file hình ảnh
            img_filename = out_file.replace('.png', '_granulometry.png')
            plt.savefig(img_filename, dpi=300)  # Lưu với độ phân giải 300 dpi
            print(f"Granulometry plot saved to {img_filename}")

            # Hiển thị biểu đồ
            plt.show()
```

```python
source > main_grayscale.py > operator
def operator(in_file, out_file, mor_op, mode, wait_key_time=5000):
        print(f"Execution Time ({method}): {exec_time:.6f} seconds")


def main(argv):
    """Xử lý đầu vào từ dòng lệnh."""
    input_file = ''
    output_file = ''
    mor_op = ''
    mode = 'manual'
    wait_key_time = 5000

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

    try:
        opts, args = getopt.getopt(argv, "hi:o:p:m:t:", ["in_file=", "out_file=", "mor_operator=", "mode=", "wait_key_t
    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()
```

```python
source > main_grayscale.py > main
def main(argv):
        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()
        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)

    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:

<u>Binary image.</u>

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**

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

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

- **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

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

- **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.

9. connected_components(img)

Algorithm

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

- **Usage**: Identifies and labels connected components in a binary image. Useful for separating distinct objects.
- **Algorithm Explanation**:
  - Initializes a label matrix with zeros and starts labeling from 1.
  - Iterates through the image, and when a foreground pixel (1) is found without a label, a flood-fill approach (using a queue) is used to assign the same label to all connected pixels.

- Uses 8-connectivity (including diagonals) to ensure all connected pixels are grouped under the same label.
- Returns an image where each connected component has a unique label.

10. convex_hull(img)

Algorithm

$$X_0^i = A,\ i = 1,2,3,4$$
$$X_k^i = (X_{k-1} \otimes B^i) \cup A,\quad i = 1,2,3,4\ \ and\ \ k = 1,2,3,.$$
$$D^i = X_{conv}^i\ (X_k^i = X_{k-1}^i)$$
$$C(A) = \bigcup_{i=1}^{4} D^i$$

Usage: Computes the convex hull of bright regions using the Jarvis March (Gift Wrapping) algorithm. This is useful for shape analysis and object simplification.

Algorithm Explanation:

- Extracts all foreground pixel coordinates.
- Finds the leftmost point to start the hull.
- Iteratively selects the next point that forms the largest counter-clockwise angle with the current hull.
- Continues until the hull forms a closed shape.
- Returns a binary image where the convex hull is marked.

11. thinning(img)

Algorithm

$$X \varnothing B = X - (X \otimes B)$$
$$\{B\} = \{B^1, B^2, B^3, \ldots, B^n\}, B^i = R(B^{i-1})$$
$$X \varnothing \{B\} = ((\ldots((X \varnothing B^1) \varnothing B^2)\ldots) \varnothing B^n)$$

Usage: Reduces the thickness of objects to a single-pixel-wide skeleton using the Zhang-Suen algorithm. Useful for shape representation and handwriting recognition.

Algorithm Explanation:

- Iteratively removes border pixels while preserving connectivity.
- Uses two sub-iterations:
  - First pass removes pixels that satisfy specific neighborhood conditions.
  - Second pass removes a different set of pixels.
- Stops when no further changes occur.
- Returns a skeletonized version of the image.

12. thickening(img, kernel)

Algorithm

$$X * B = X \bigcup (X \otimes B)$$
$$\{B\} = \{B^1, B^2, B^3, ..., B^n\}, B^i = R(B^{i-1})$$
$$X * \{B\} = ((...((X * B^1) * B^2)...) * B^n)$$

Usage: Expands objects in the binary image while preserving their general structure. Often used for strengthening weak edges.

Algorithm Explanation:

- Applies dilation to the image using the given structuring element.
- Combines the dilated image with the complement of the original image to selectively thicken object boundaries.
- Returns the thickened binary image.

13. skeletonization(img)

Algorithm

$$S(X) = \{p \in X : \exists r \geq 0, B(p,r) \text{ is a maximal ball of } X\}$$
$$S(X) = \bigcup_{k=0}^{K} S_k(X)$$
$$S_k(X) = (X \Theta kB) - (X \Theta kB) \circ B$$
$$(X \Theta kB) = (((X \Theta B) \Theta B) \Theta ....) \Theta B, k \text{ times}$$
$$K = \max\{k \mid (X \Theta kB) \neq \varnothing\}$$

Usage: Extracts the medial axis (skeleton) of objects in a binary image while preserving topology. Useful for shape representation.

Algorithm Explanation:

- Iteratively applies erosion to the image.
- At each step, the difference between the eroded image and its opened version is stored as part of the skeleton.
- The process stops when the image is completely eroded.
- Returns the extracted skeleton.

14. reconstruction(marker, mask, kernel)

$$S(X) = \bigcup_{k=0}^{K} (S_k(X) \oplus kB)$$
$$(S_k(X) \oplus kB) = ((...(S_k(X) \oplus B) \oplus B) \oplus ...) \oplus B$$

- **Usage**: Reconstructs an image by iteratively propagating bright pixels from a marker image while being constrained by a mask image. Useful for restoring objects after opening operations.
- **Algorithm Explanation**:
    - Initializes the marker image with a given seed region.

- Iteratively applies dilation to expand the marker image while ensuring it does not exceed the mask image.
- Stops when no further changes occur.
- Returns the reconstructed image.

15. pruning(skeleton, iterations)

Algorithm

$$X_1 = X \oslash \{B\} \, (thinning)$$

$$X_2 = \bigcup_{k=1}^{8} (X_1 \otimes B^k) \, (hit-or-miss)$$

$$X_3 = (X_2 \oplus H) \cap A$$

$$X_4 = X_1 \cup X_3$$

Usage: Removes extraneous endpoints from a skeletonized image to refine its structure. Often used to clean up noisy skeletons.

Algorithm Explanation:

- Identifies endpoints in the skeleton (pixels with only one neighbor).
- Iteratively removes these endpoints up to the specified number of iterations.
- Returns a pruned version of the skeleton.

GRAYSCALE IMAGE

1. Grayscale Dilation

- **Definition**

$$(f \oplus b)(s,t) = \max\{ f(s-x, t-y) + b(x,y) \mid (s-x), (t-y) \in D_f ; (x,y) \in D_b \}$$

- **Usage**: Expands bright regions in a grayscale image by replacing each pixel with the maximum value in its neighborhood. Enhances bright structures.
- **Algorithm Explanation**:
  - Applies padding to the image to avoid boundary issues.
  - Iterates through the image, replacing each pixel with the maximum value within the structuring element.
  - Returns the dilated grayscale image.

2. Grayscale Erosion

- **Definition**

$$(f \ominus b)(s,t) = \min\{ f(s+x, t+y) - b(x,y) \mid (s+x), (t+y) \in D_f ; (x,y) \in D_b \}$$

Usage: Shrinks bright regions by replacing each pixel with the minimum value in its neighborhood. Removes small bright details.

Algorithm Explanation:

- Applies padding to handle boundary effects.
- Iterates through the image, replacing each pixel with the minimum value within the structuring element.
- Returns the eroded grayscale image.

3. Grayscale Opening

• Definition

$$f \circ b = (f \Theta b) \oplus b$$

- **Usage**: Removes small bright spots by first performing erosion (shrinking objects) followed by dilation (restoring shape). Helps with noise reduction.
- **Algorithm Explanation**:
    - Applies grayscale erosion to shrink bright areas.
    - Applies grayscale dilation to restore larger structures.
    - Returns the processed grayscale image.

4. Grayscale Closing

• Definition

$$f \bullet b = (f \oplus b) \Theta b$$

Usage: Fills small dark gaps by first performing dilation (expanding objects) followed by erosion (shrinking back). Helps close small holes in bright areas.

Algorithm Explanation:

- Applies grayscale dilation to expand bright areas.
- Applies grayscale erosion to restore original object shapes.
- Returns the processed grayscale image.

5. Grayscale Smoothing

• Definition

$$h = (f \circ b) \bullet b$$

- **Usage**: Reduces noise and smooths object boundaries by applying both grayscale opening and closing sequentially.
- **Algorithm Explanation**:
    - First applies grayscale opening to remove small bright spots.
    - Then applies grayscale closing to fill small dark holes.
    - Returns the smoothed grayscale image.

6. Morphology Gradient

• Definition

$$h = (f \oplus b) - (f \Theta b)$$

- **Usage**: Highlights object edges by computing the difference between dilated and eroded versions of the image. Enhances edge features.

- **Algorithm Explanation**:
    - Computes the difference: **(Dilation - Erosion)**.
    - Bright areas indicate regions with

7. Top Hat

- Definition

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

- **Usage**: Extracts small bright details by subtracting the opened image from the original. Useful for enhancing fine structures.
- **Algorithm Explanation**:
    - Applies **grayscale opening** to remove small bright regions.
    - Subtracts the opened image from the original image, preserving only small bright details.
    - Returns the top-hat transformed image.

8. Textual segmentation

- Definition

$$h = (f \bullet b_1) \circ b_2$$

- **Usage**: Enhances texture features in an image by extracting fine structures using the top-hat transformation.
- **Algorithm Explanation**:
    - Uses **top-hat transformation** to highlight small bright details.
    - Enhances textural patterns by emphasizing variations in local brightness.
    - Returns the segmented image emphasizing textures.

9. Granulometry

- **Usage**: Analyzes the size distribution of bright structures by applying **grayscale opening** with different structuring element sizes.
- **Algorithm Explanation**:
    - Iterates through different kernel sizes.
    - Applies **grayscale opening** at each size to filter out smaller bright regions.
    - Computes the sum of pixels after each opening, indicating the presence of structures of different sizes.
    - Returns a list of summed pixel values for different kernel sizes, used to plot a granulometry curve.

10. Reconstruction

1. Opening by reconstruction of the original image using a horizontal line of size 1x71 pixels in the erosion operation

$$O_R^{(n)}(f) = R_f^D[f \ominus nb]$$

2. Subtract the opening by reconstruction from original image

$$f' = f - O_R^{(n)}(f)$$

3. Opening by reconstruction of the f' using a vertical line of size 11x1 pixels

$$f1 = O_R^{(n)}(f') = R_f^D[f' \ominus nb']$$

4. Dilate f1 with a line SE of size 1x21, get f2.

5. Calculate the minimum between the dilated image f2 and and f', get f3.

6. By using f3 as a marker and the dilated image f2 as the mask,

$$R_{f2}^D(f3) = D_{f2}^{(k)}(f3)$$

with k such that $D_{f2}^{(k)}(f3) = D_{f2}^{(k+1)}(f3)$

- **Usage**: Restores missing or occluded image regions by iteratively applying **dilation** constrained by a mask.
- **Algorithm Explanation**:
  - Initializes a **marker** image, typically a subset of the mask.
  - Iteratively dilates the marker while ensuring it does not exceed the mask.
  - Stops when the marker no longer changes between iterations or reaches **max_iter**.
  - Returns the reconstructed grayscale image.

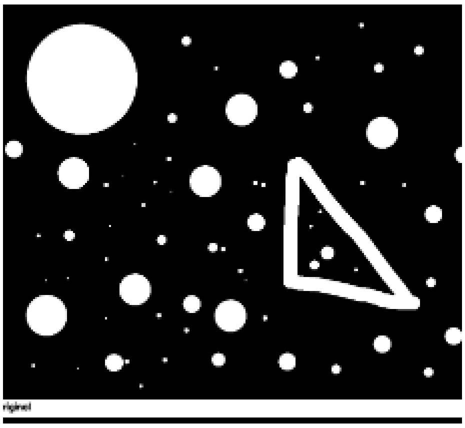
IV. EXPERIMENTS AND EVALUATION:

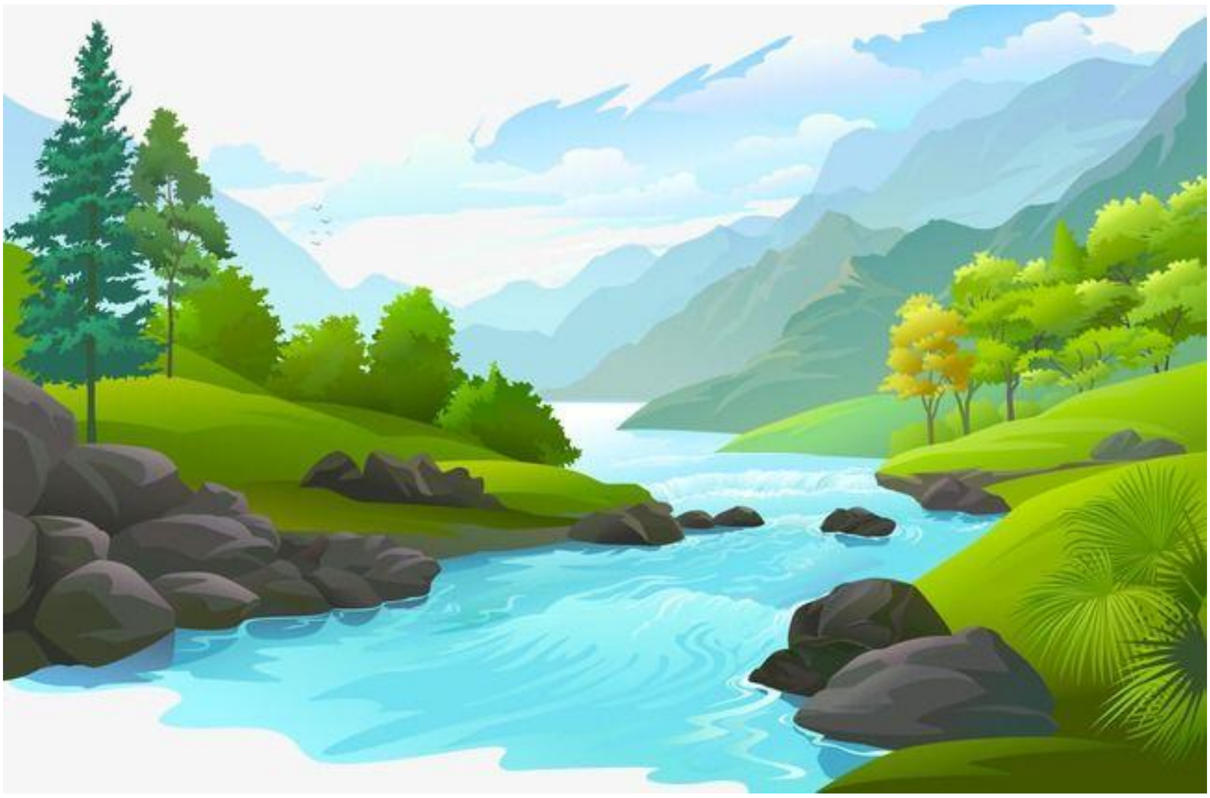
1st to 7th function in binary.py

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


love


Discover

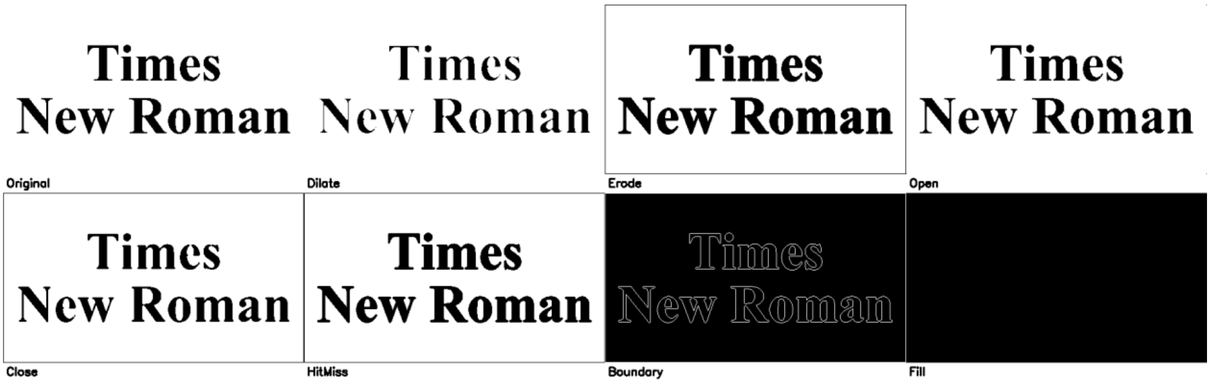




Results with opencv mode vs manual mode:

Text_tnr.png

Execution Time (Manual (Custom)): 15.685021 seconds



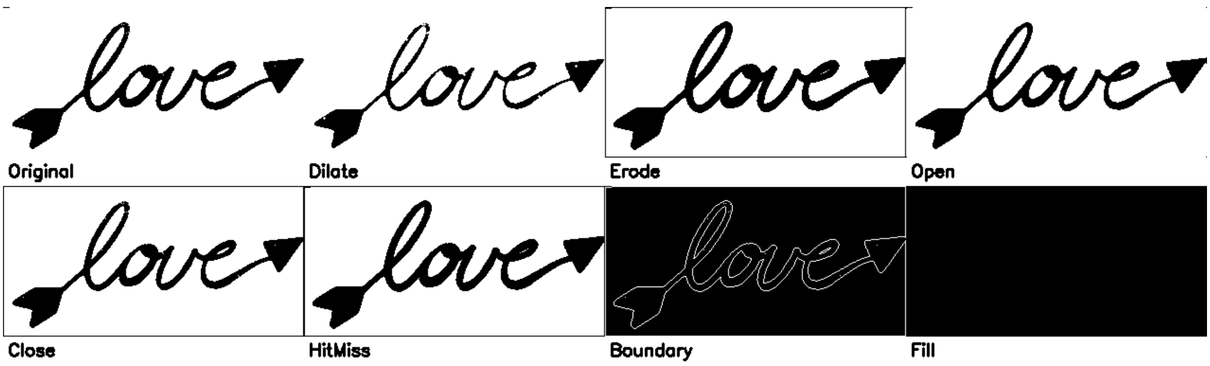
Execution Time (OpenCV): 0.002056 seconds

Comments:

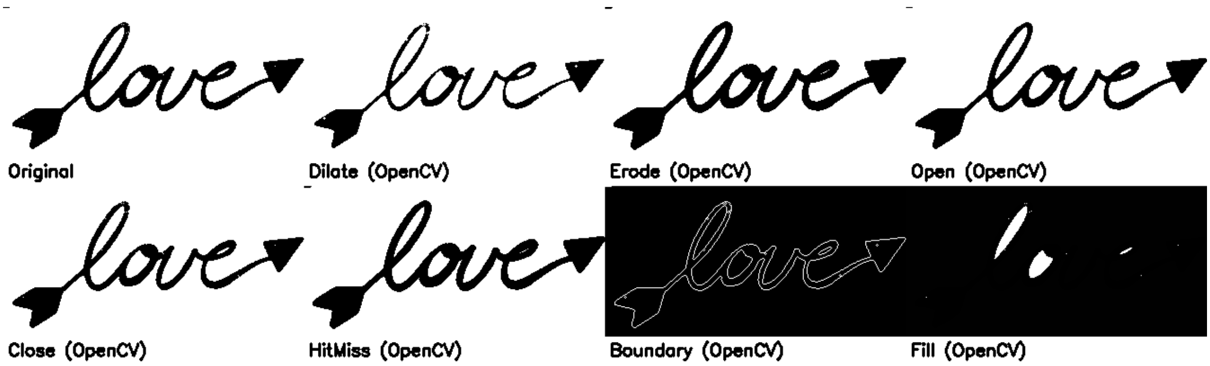
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Lover.png

Execution Time (Manual (Custom)): 6.485807 seconds



Execution Time (OpenCV): 0.000000 seconds

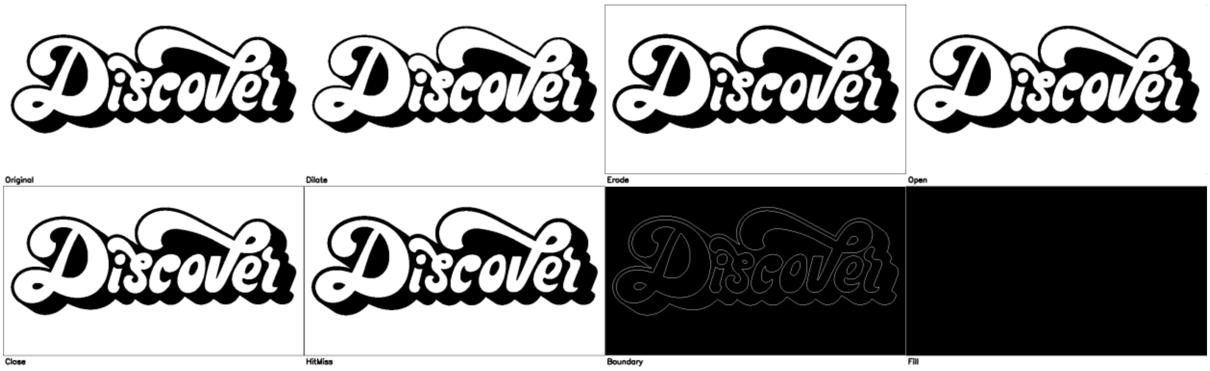


Comments:

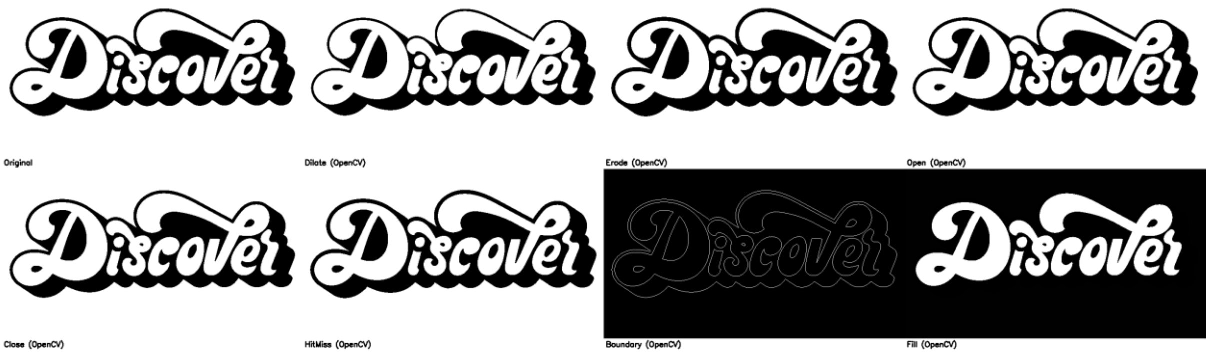
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Discover.jpg

Execution Time (Manual (Custom)): 30.812511 seconds



Execution Time (OpenCV): 0.004053 seconds

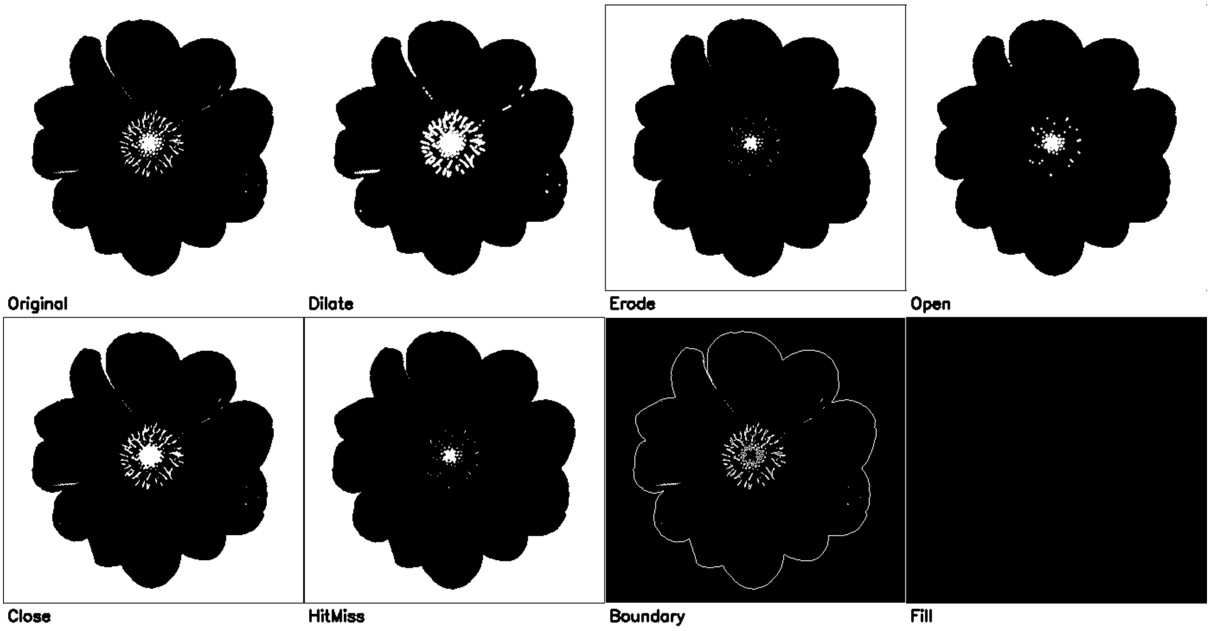


Comments:

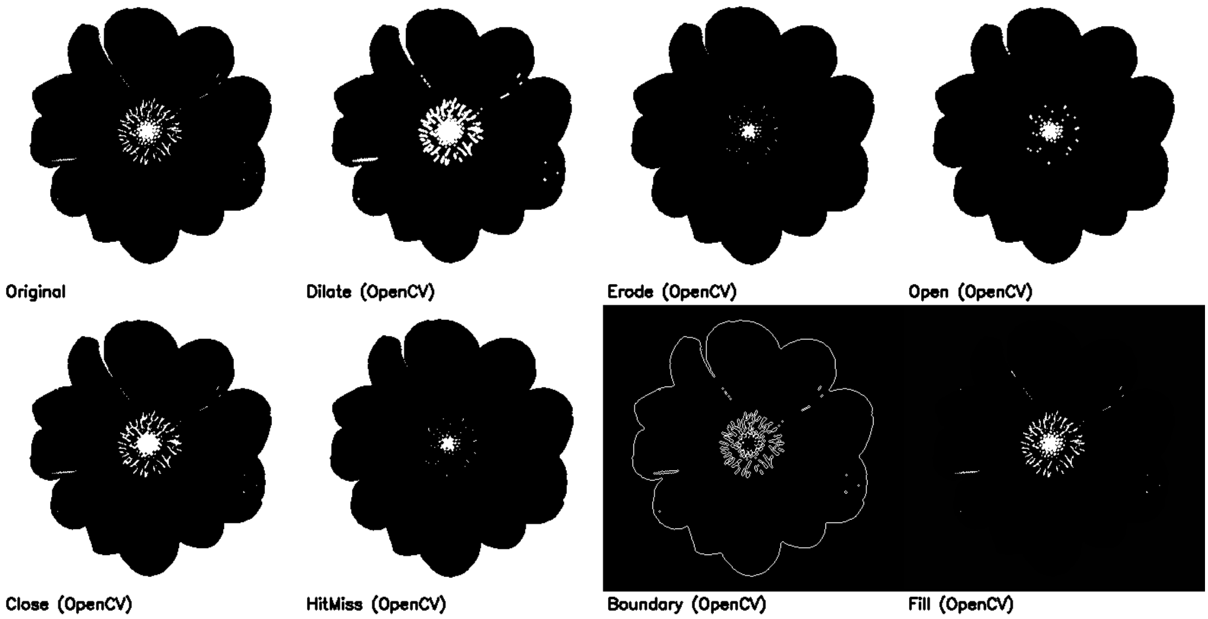
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Flower.png

Execution Time (Manual (Custom)): 9.767103 seconds



Execution Time (opencv): 0.016066 seconds

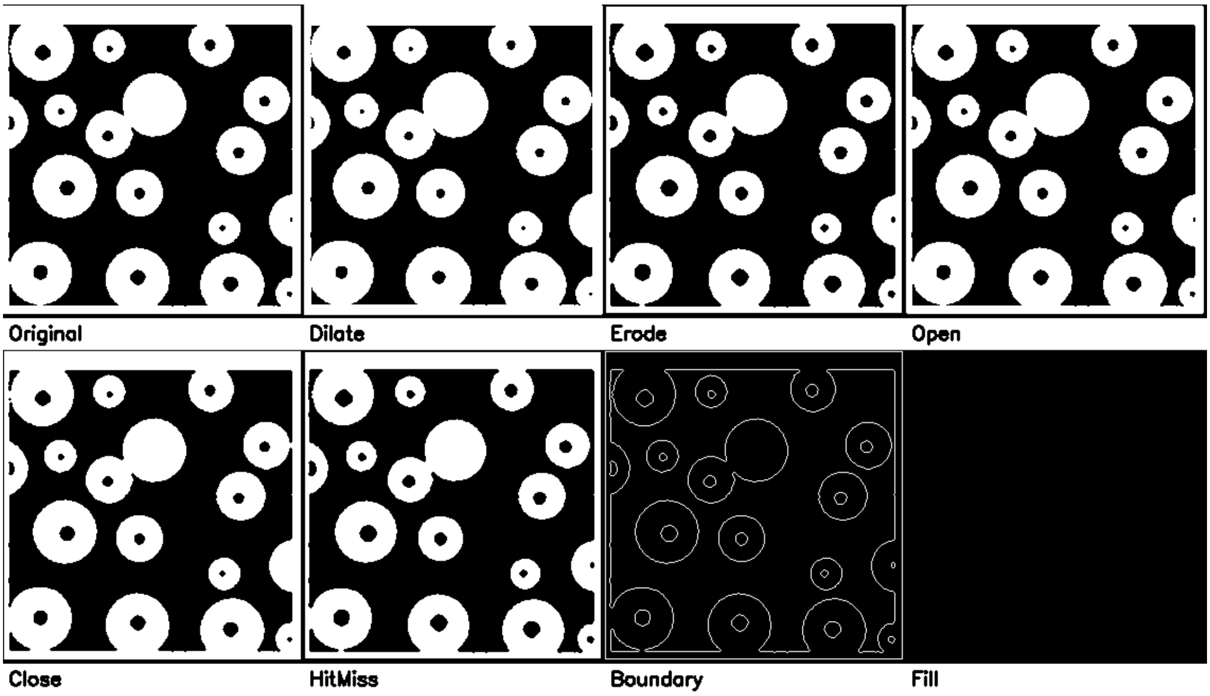


Comments:

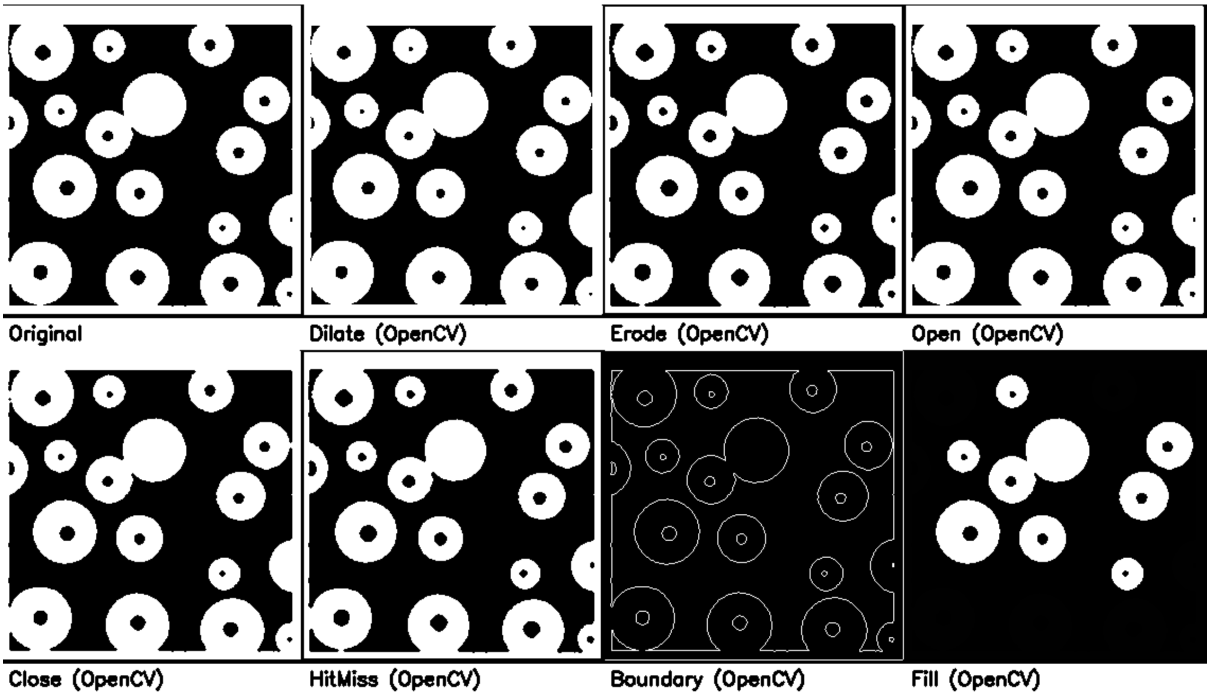
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation. Boundary operator (openCV) mode is more detailed than the custom version

<u>Circle.png</u>

Execution Time (manual): 7.076829 seconds



Execution Time (opencv): 0.016066 seconds

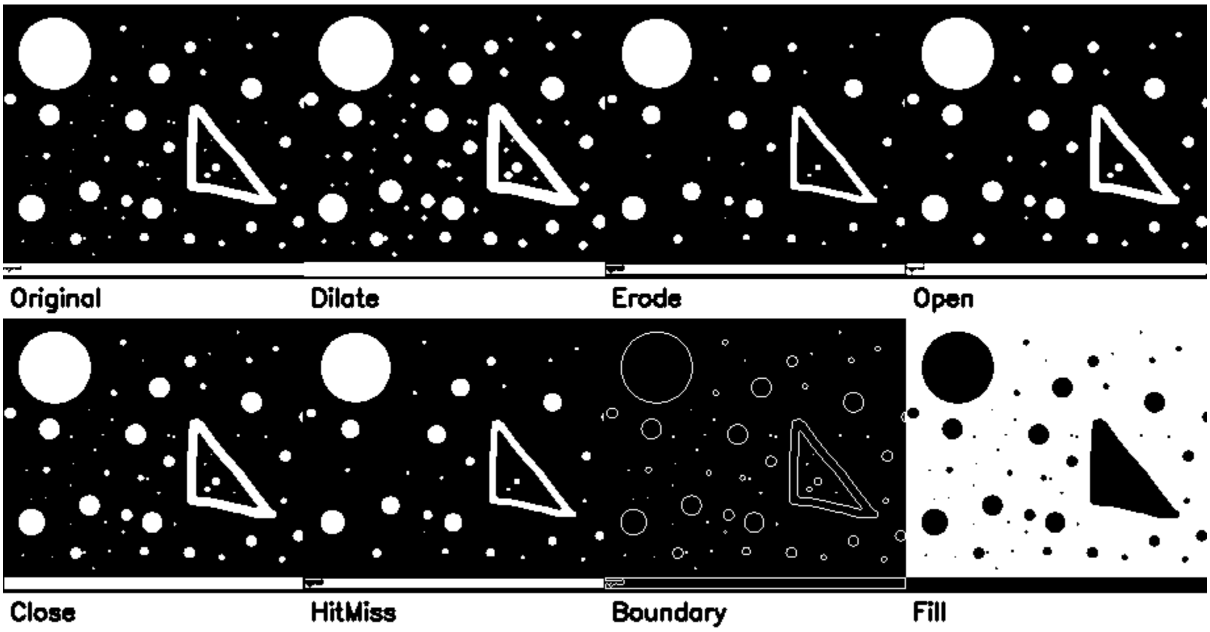


Comments:

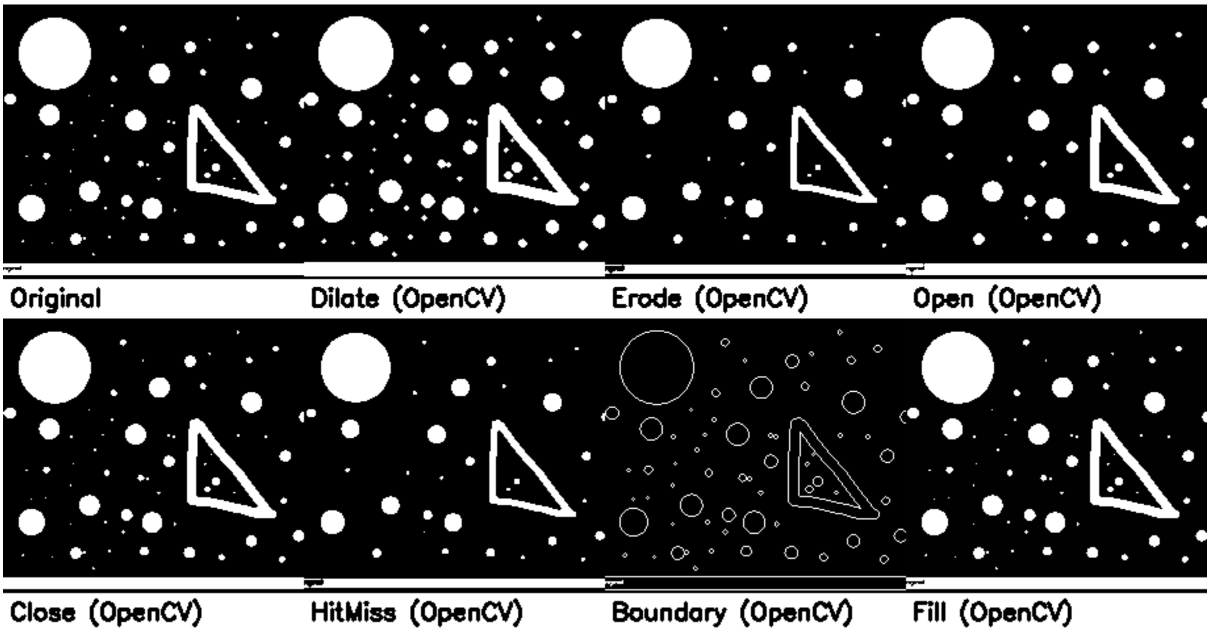
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.

Star.png

Execution Time (manual): 299.239042 seconds



Execution Time (opencv): 0.000000 seconds



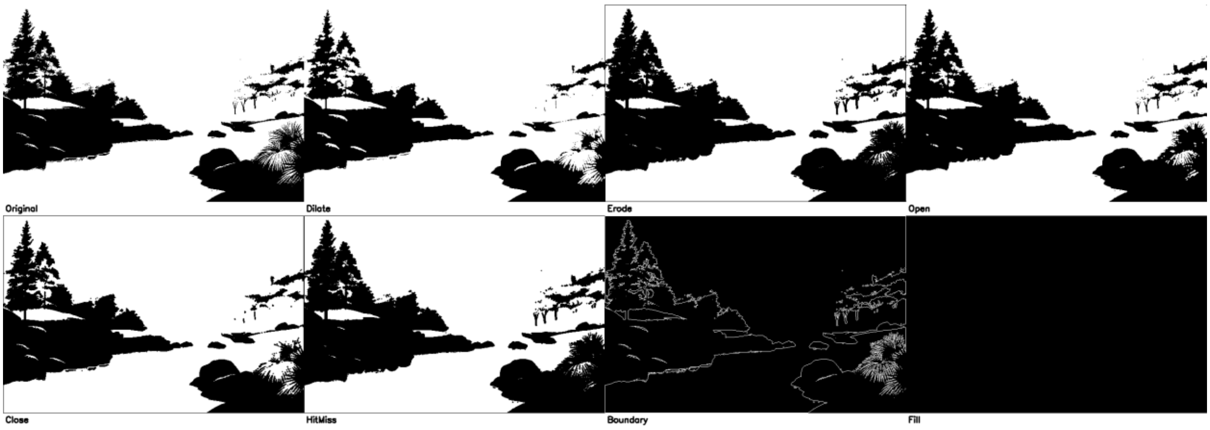
Comments:

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.

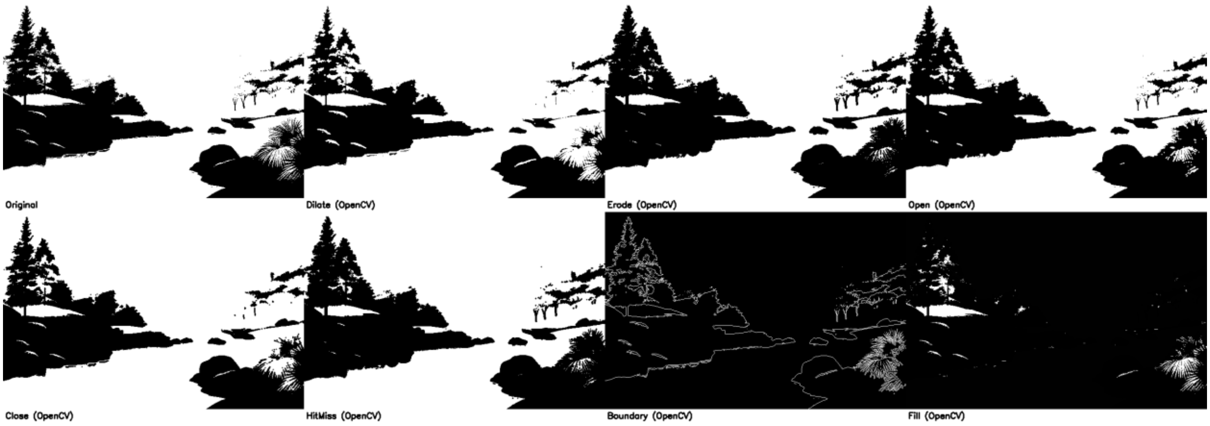
Region filling (manual) is not as expected, wrong region.

Landscape.png

Execution Time (manual): 24.075548 seconds



Execution Time (opencv): 0.005046 seconds



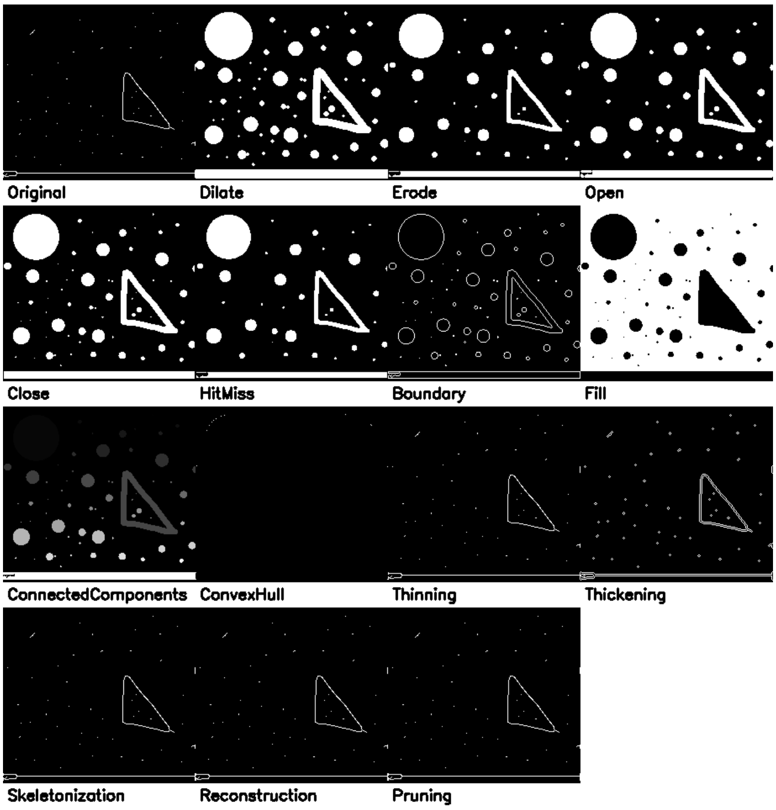
Comments:

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

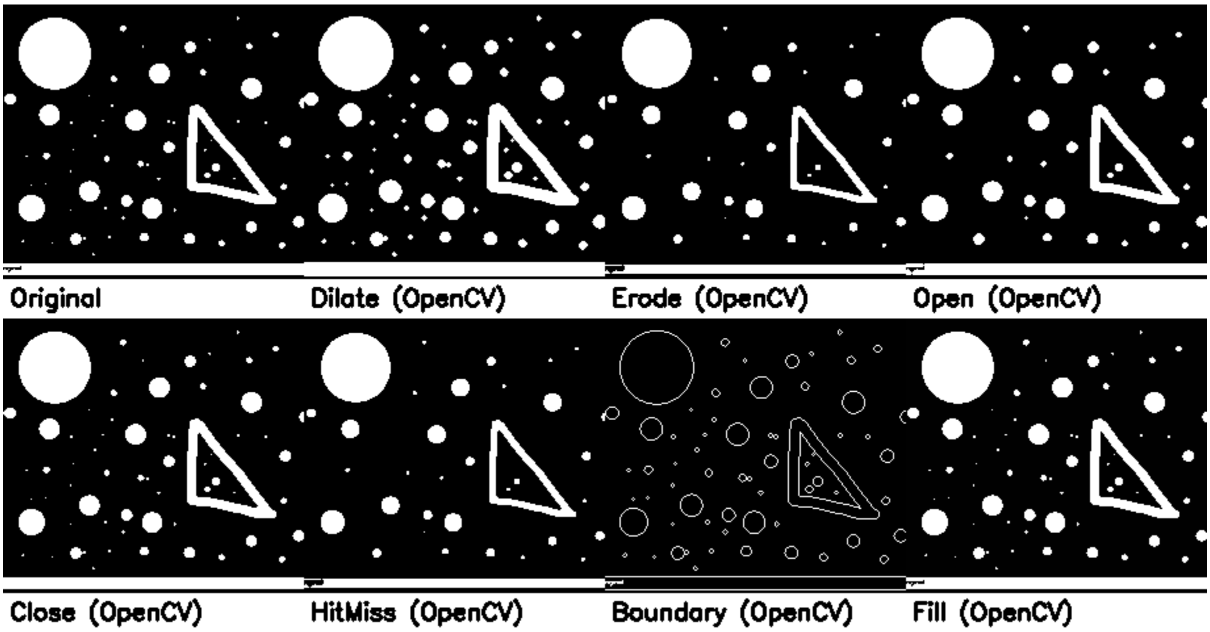
7^{st} to 14^{th} function in binary.py and 10 operators in grayscale image

I only test with star.png (the left function) và landscape.png (grayscale image)

Manual results:



Opencv results:



Comments:

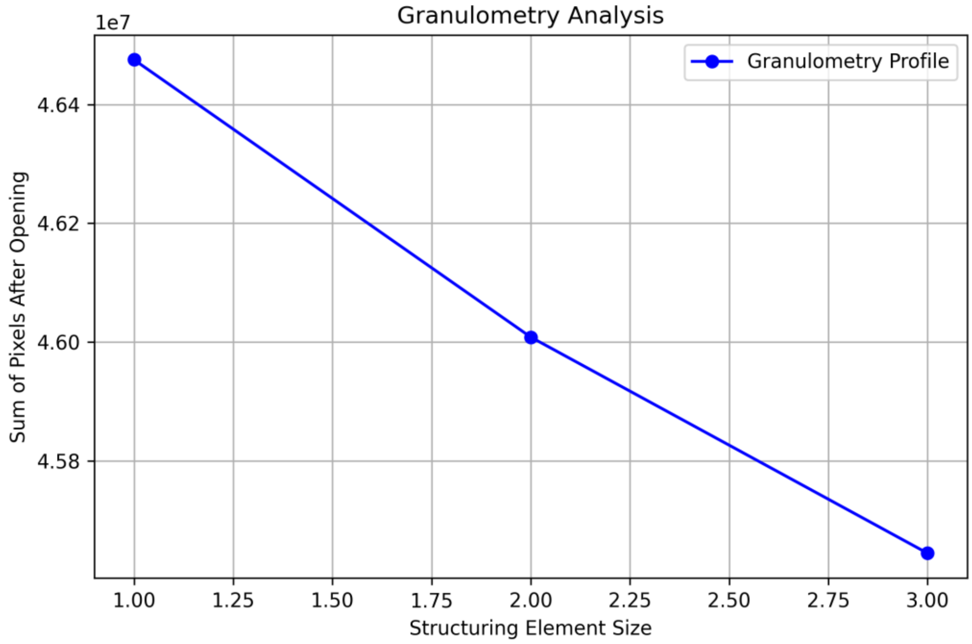
Similar results in morphological operations, but the OpenCV built-in method using flood filling missed some regions. OpenCV is significantly faster than the custom implementation.
Boundary operator (openCV) mode is more detailed than the custom version.

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

<u>Grayscale image.</u>

The results of granulometry function on landscape grayscale image

```
output_image > output_landscape_manual_all_grayscale.txt
1    46475241.000000
2    46008343.000000
3    45644549.000000
4
```



The Granulometry output file now contain different values

granulometry function is now producing **three distinct values:**

- **46475241.000000**
- **46008343.000000**
- **45644549.000000**

What do these values represent?

Each value corresponds to the **sum of pixel intensities** after applying **Grayscale Opening** with different structuring element (SE) sizes.

- The first value (**46475241.000000**) corresponds to **SE size 3x3**.
- The second value (**46008343.000000**) corresponds to **SE size 5x5**.
- The third value (**45644549.000000**) corresponds to **SE size 7x7**.

Why do the values decrease?

As the structuring element **size increases**, **more small bright regions are removed**, leading to a **reduction in total pixel intensity**.

- **Smaller SEs (3x3)** only remove very small noise.
- **Larger SEs (7x7)** remove larger details and thin structures, reducing the total pixel sum.

What does this tell us about the image?

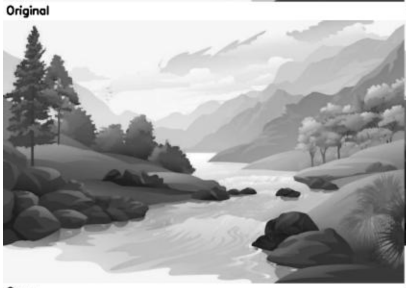
- The image **contains small bright structures** that are gradually removed as SE size increases.
- The decrease in values suggests **progressive removal of finer details**, which aligns with granulometry analysis.
- If values stayed **constant**, it would mean **no small structures were removed**, or SE sizes were **too small** to affect the image.

  Opencv results


Original

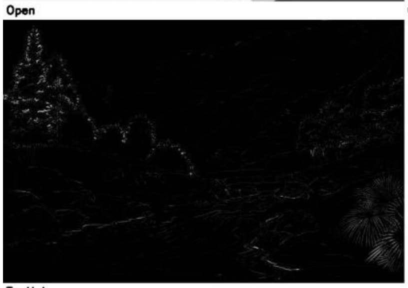

Dilate


Erode

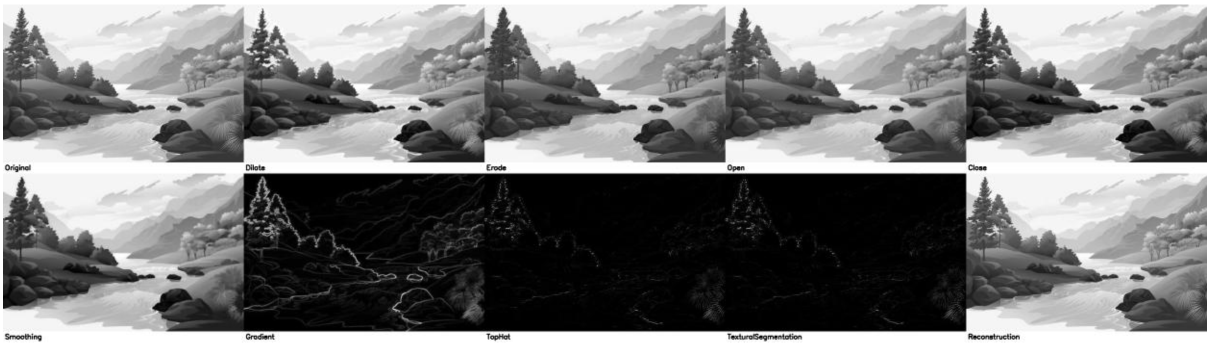

Open


Close


Gradient


TopHat

Manual results:



Comments:

The custom grayscale morphological operations produce results that often differ from OpenCV's built-in functions, with some outputs appearing darker or less refined. OpenCV generally provides smoother and more accurate transformations due to optimized padding, normalization, and efficient kernel application. The main issues in the custom implementation likely stem from **inefficient pixel scaling, boundary handling, and loop-based computations**. Refining these aspects could improve accuracy and bring results closer to OpenCV's optimized output.

Comparison in general

Binary image and grayscale image operator custom vs opencv built in

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/ Gradient	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
Thinning	Works but may leave artifacts if conditions aren't checked properly	Optimized and standardizes Zhang-Suen or Guo-Hall algorithms