

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**  
**KHOA CÔNG NGHỆ THÔNG TIN**



## **BÁO CÁO THỰC HÀNH**

**Ứng dụng xử lý ảnh số và video số 20\_23**

*Giảng viên hướng dẫn – TS. Lý Quốc Ngọc*

Thành phố Hồ Chí Minh – 2023

## MỤC LỤC

<b>THÔNG TIN SINH VIÊN .....</b>	<b>4</b>
<b>Bảng mức độ hoàn thành công việc bài tập một.....</b>	<b>4</b>
<b>Bảng mức độ hoàn thành công việc bài tập hai.....</b>	<b>4</b>
<b>I. Các mặt nạ con – kernel.....</b>	<b>5</b>
<b>II. Binary image &amp; Gray image .....</b>	<b>5</b>
1. Binary image .....	5
2. Gray image .....	7
3. So sánh .....	8
<b>BÀI TẬP MỘT THỰC HÀNH – BINARY IMAGE .....</b>	<b>9</b>
<b>III. Các toán tử Morphology được sử dụng trong bài thực hành một .....</b>	<b>9</b>
1. Toán tử Dilation .....	9
2. Toán tử Erosion .....	15
3. Toán tử Opening.....	21
4. Toán tử Closing .....	25
5. Toán tử Hit or Miss .....	30
6. Toán tử Boundary Extraction.....	36
7. Toán tử Convex Hull .....	41
8. Toán tử rút trích thành phần liên thông.....	46
9. Toán tử Region Filling.....	49
10. Toán tử Thinning .....	54
<b>BÀI TẬP HAI THỰC HÀNH – GRAY IMAGE.....</b>	<b>61</b>
<b>IV. Các toán tử Morphology được sử dụng trong bài thực hành hai .....</b>	<b>61</b>
1. Toán tử Dilation .....	61
2. Toán tử Erosion .....	66
3. Toán tử Opening.....	71

4. Toán tử Closing .....	74
5. Toán tử Gradient.....	78
6. Toán tử Granulometry .....	84
7. Toán tử Reconstruction.....	87
8. Toán tử Smoothing (làm tròn) .....	92
9. Toán tử Textual Segmentation.....	96
10. Toán tử Top Hat .....	100
<b>V. Thủ tạo nhiễu và lọc nhiễu (tăng cường) qua Opening và Closing .....</b>	<b>104</b>
<b>VI. Hướng dẫn chương trình.....</b>	<b>105</b>
1. Sơ đồ tổ chức thư mục.....	105
2. Cách khởi chạy chương trình.....	105
<b>TÀI LIỆU THAM KHẢO.....</b>	<b>108</b>

## THÔNG TIN SINH VIÊN

MSSV	Họ Tên	Email	Ghi chú
20120201	Phạm Gia Thông	20120201@student.hcmus.edu.vn	

### Bảng mức độ hoàn thành công việc bài tập một

STT	Toán tử	OpenCV	Non-OpenCV
1	Dilation	100%	100%
2	Erosion	100%	100%
3	Opening	100%	100%
4	Closing	100%	100%
5	Hit or Miss	100%	100%
6	Region Filling	100%	80%
7	Connected Components	100%	100%
8	Boundary Extraction	100%	100%
9	Convex Hull	100%	70%
10	Thinning	100%	60%

### Bảng mức độ hoàn thành công việc bài tập hai

STT	Toán tử	OpenCV	Non-OpenCV
1	Dilation	100%	100%
2	Erosion	100%	100%
3	Opening	100%	100%
4	Closing	100%	100%
5	Gradient	100%	100%
6	Granulometry	80%	80%
7	Reconstruction	100%	70%
8	Smoothing	100%	100%
9	Text segmentation	100%	100%
10	Top hat	100%	100%

## I. Các mặt nạ con – kernel

Mặt nạ con kernel với bài tập có sử dụng thư viện OpenCV

```
# defining the kernel i.e. Structuring element, tạo ra nhiều lớp mặt nạ kernel khác nhau
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))      #sử dụng để tạo ra một kernel hình elip (ellipse) với kích thước 5x5, và các số 1 tạo thành khung trống trung tâm
kernel_one = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))  #mặt nạ kernel hình chữ nhật 3x3
kernel_two = cv2.getStructuringElement(cv2.MORPH_RECT, (6, 3))  #mặt nạ kernel hình chữ nhật 6x3
kernel_thr = cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))  #mặt nạ kernel hình chữ thập 5x5
ker = np.array([[0, 1, 0],
               [1, 1, 1],
               [0, 1, 0]], dtype=np.uint8)    #mặt nạ kernel hình chữ thập 3x3
kerl = np.ones((5,5),np.uint8)      #mặt nạ kernel hình chữ nhật 5x5
```

Mặt nạ con kernel với bài tập không sử dụng thư viện OpenCV

```
# Định nghĩa kernel
kernel_1 = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
kernel_2 = np.array([[1, 1, 1],
                    [1, 1, 1],
                    [1, 1, 1]])
kernel_3 = np.array([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
kernel_4 = np.ones((5,5),np.uint8)      #mặt nạ kernel hình chữ nhật 5x5
mask = [[1, 1, 1],
        [1, 0, 1],
        [1, 1, 1]]
```

- Mặt nạ kernel hình ellipse với kích thước 5x5
- Mặt nạ kernel hình chữ nhật với kích thước 3x3
- Mặt nạ kernel hình chữ nhật với kích thước 5x5
- Mặt nạ kernel hình chữ nhật với kích thước 6x3

## II. Binary image & Gray image

### 1. Binary image

Ảnh nhị phân là một loại ảnh kỹ thuật số chỉ chứa hai giá trị pixel có thể có, thường là màu trắng và đen hoặc 0 và 1. Trong đó, mỗi điểm ảnh trên ảnh được biểu diễn bằng một bit, với giá trị 0 thường được sử dụng để biểu thị màu đen và giá trị 1 được sử dụng để biểu thị màu trắng.

Các đặc điểm chính của ảnh nhị phân bao gồm:

- Chỉ chứa hai giá trị pixel: 0 và 1, tương ứng với màu đen và màu trắng.
- Kích thước tập tin nhỏ hơn so với các định dạng ảnh khác, vì chỉ cần lưu trữ một bit cho mỗi điểm ảnh.
- Dễ dàng để xử lý và tính toán vì chỉ có hai giá trị pixel.
- Thường được sử dụng trong các ứng dụng như phát hiện cạnh, phân đoạn ảnh, trích xuất đặc trưng và nhận dạng khuôn mặt và chữ viết tay.
- Có thể bị mất thông tin, do đó cần phải cân nhắc sử dụng chúng trong các ứng dụng đòi hỏi độ chính xác cao.
- Dễ dàng chuyển đổi sang các định dạng ảnh khác và ngược lại, ví dụ như ảnh grayscale.

Các đặc điểm này khiến cho ảnh nhị phân trở thành một định dạng ảnh phổ biến và quan trọng trong xử lý ảnh và các ứng dụng liên quan đến nhận dạng và phân tích hình ảnh. Ảnh nhị phân với một vai trò quan trọng trong các phép toán hình thái học (morphology) trên ảnh. Toán tử hình thái học là một kỹ thuật xử lý ảnh dựa trên các phép biến đổi hình học trên các đối tượng trong ảnh như phóng to, co lại, tìm kiếm cạnh, loại bỏ nhiễu, v.v.

Với ảnh nhị phân, các phép toán hình thái học trở nên dễ dàng hơn và hiệu quả hơn so với các loại ảnh khác vì chỉ có hai giá trị pixel. Những phép toán này giúp cải thiện chất lượng ảnh, làm giảm nhiễu, tách đối tượng và nâng cao khả năng phân tích đối tượng.

Mặc dù các phép toán hình thái học có thể có tính phức tạp, nhưng các công cụ xử lý ảnh hiện đại cung cấp cho người dùng nhiều tùy chọn và thuật toán khác nhau để tối ưu hóa các kết quả. Tuy nhiên, để thực hiện các phép toán hình thái học hiệu quả, người dùng cần có kiến thức chuyên môn về xử lý ảnh và thuật toán hình thái học.

```
# Đọc ảnh đầu vào đen trắng nhị phân
img = cv2.imread(str(file_dir), cv2.IMREAD_GRAYSCALE)

# Chuyển đổi ảnh sang dạng nhị phân với ngưỡng 127
_, binary_img = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)

return binary_img, path

img = Image.open(str(file_dir)).convert('L') # Chuyển sang mức xám
# Chuyển đổi ảnh sang dạng numpy array
img_array = np.array(img)

# Tạo ảnh nhị phân với ngưỡng 127
binary_img = np.zeros_like(img_array) # Tạo ảnh đen
binary_img[img_array > 127] = 255 # Đặt các pixel lớn hơn 127 thành trắng
#binary_img = gray2binary(img_array)

# Chuyển đổi ảnh nhị phân sang định dạng PIL Image
binary_img_pil = Image.fromarray(binary_img)

return binary_img_pil, path
```

## 2. Gray image

Ảnh độ xám (grayscale image) là một loại ảnh kỹ thuật số chỉ chứa các giá trị mức xám từ 0 đến 255, thường được sử dụng để biểu diễn các ảnh với độ sáng và tối khác nhau mà không cần sử dụng nhiều bộ nhớ như ảnh màu.

Trong ảnh độ xám, mỗi điểm ảnh được biểu diễn bằng một giá trị duy nhất, thường là một số nguyên từ 0 đến 255, với giá trị 0 tương ứng với màu đen và giá trị 255 tương ứng với màu trắng. Các giá trị mức xám nằm giữa 0 và 255 biểu thị các mức xám khác nhau của điểm ảnh, trong đó giá trị càng cao thì mức xám càng sáng.

Các đặc điểm chính của ảnh độ xám bao gồm:

- Chỉ sử dụng một kênh màu: Ảnh độ xám chỉ sử dụng một kênh màu duy nhất để biểu diễn các giá trị mức xám từ 0 đến 255.
- Kích thước ảnh nhỏ hơn ảnh màu: Vì chỉ sử dụng một kênh màu nên kích thước của ảnh độ xám thường nhỏ hơn so với ảnh màu.
- Dễ dàng để tính toán và xử lý: Do có kích thước nhỏ hơn và chỉ sử dụng một kênh màu nên ảnh độ xám dễ dàng để tính toán và xử lý trong các ứng dụng xử lý ảnh.
- Biểu diễn độ sáng và tối của ảnh: Ảnh độ xám được sử dụng để biểu diễn các ảnh với độ sáng và tối khác nhau mà không cần sử dụng nhiều bộ nhớ như ảnh màu.
- Thường được sử dụng trong các ứng dụng nhận dạng và xử lý ảnh: Các phương pháp xử lý ảnh độ xám thường được sử dụng trong các ứng dụng nhận dạng, phân loại và xử lý ảnh, bao gồm cả xử lý ảnh y tế, phát hiện biển số xe và chụp ảnh vi mô

Ảnh độ xám đóng vai trò quan trọng trong toán tử hình thái học morphology trong xử lý ảnh. Toán tử hình thái học là một phương pháp xử lý ảnh dựa trên các phép biến đổi hình thái của đối tượng, như co giãn, giãn, mở, đóng, để phát hiện và loại bỏ các đối tượng nhiễu, cấu trúc không mong muốn và phát hiện các đặc trưng hình thái.

Các toán tử hình thái học thường được áp dụng trên ảnh độ xám để tạo ra các biến đổi hình thái như giãn, co giãn, đóng và mở. Ảnh độ xám cung cấp thông tin về độ sáng và tối của các điểm ảnh, giúp xác định các đối tượng có kích thước khác nhau trong ảnh.

Toán tử hình thái học có thể được sử dụng để loại bỏ các nhiễu và đối tượng không mong muốn khỏi ảnh độ xám, tìm kiếm đường viền của các đối tượng trong ảnh, phân tách các đối tượng gần nhau và tạo ra các đặc trưng hình thái của các đối tượng. Toán tử hình thái học trên ảnh độ xám có thể đòi hỏi một số kiến thức về xử lý ảnh, nhưng cũng có thể được áp dụng với các công cụ và thư viện xử lý ảnh có sẵn để giúp việc xử lý trở nên đơn giản hơn.

```
# Đọc ảnh đầu vào đen trắng độ xám
img = cv2.imread(str(file_dir), cv2.IMREAD_GRAYSCALE)

return img
```

```
# Đọc ảnh đầu vào đen trắng độ xám
img = Image.open(str(file_dir)).convert('L')

return img
```

### 3. So sánh

Ảnh nhị phân và ảnh độ xám đều được sử dụng trong xử lý các phép toán hình thái học morphology, tuy nhiên, chúng có những đặc điểm khác nhau.

- Ảnh nhị phân chỉ chứa 2 giá trị màu là trắng và đen, đại diện cho các vùng đối tượng và nền. Ảnh độ xám chứa các giá trị mức xám từ 0 đến 255, biểu diễn độ sáng tại mỗi điểm ảnh trong ảnh.
- Vì chỉ chứa 2 giá trị màu, ảnh nhị phân thường có kích thước nhỏ hơn so với ảnh độ xám. Điều này làm cho việc xử lý ảnh nhị phân trở nên nhanh hơn và ít tốn tài nguyên hơn.
- Trong khi đó, ảnh độ xám chứa nhiều thông tin hơn về độ sáng của các điểm ảnh, do đó nó phù hợp hơn để sử dụng trong các ứng dụng xử lý ảnh y tế, phát hiện biển số xe và chụp ảnh vi mô.
- Trong các phép toán hình thái học morphology, ảnh nhị phân thường được sử dụng để thực hiện các phép biến đổi như giãn, co giãn, đóng và mở để loại bỏ các đối tượng nhiễu và tạo ra các đặc trưng hình thái của các đối tượng.
- Trong khi đó, ảnh độ xám thường được sử dụng để phát hiện các cạnh và đường viền của các đối tượng trong ảnh, để xác định các đối tượng có kích thước khác nhau trong ảnh và tạo ra các biến đổi hình thái như giãn, co giãn, đóng và mở.

Xử lý hình ảnh: Trong quá trình xử lý hình ảnh, ảnh nhị phân được sử dụng để phát hiện và loại bỏ các đối tượng nhiễu, cấu trúc không mong muốn, trong khi ảnh độ xám được sử dụng để phát hiện và loại bỏ các đối tượng nhiễu, cấu trúc không mong muốn và phát hiện các đặc trưng hình thái.

Độ khó khi thao tác với ảnh: Ảnh nhị phân thường dễ dàng để xử lý và thao tác hơn so với ảnh độ xám. Tuy nhiên, với các phép toán hình thái học, ảnh độ xám cũng có thể được sử dụng và có thể mang lại kết quả tốt hơn trong một số trường hợp.

Các phép toán hình thái học: Trong các phép toán hình thái học, ảnh nhị phân thường được sử dụng để thực hiện các phép toán như giãn, co giãn, đóng và mở trên các đối tượng trong ảnh. Các phép toán này được thực hiện trên các pixel trắng hoặc đen, đại diện cho các đối tượng hoặc nền. Trong khi đó, các phép toán trên ảnh độ xám thường được sử dụng để tạo ra

các biến đổi hình thái như giãn, co giãn, đóng và mở trong không gian mức xám để phát hiện các đối tượng có kích thước khác nhau trong ảnh.

Tóm lại, ảnh nhị phân và ảnh độ xám đều có vai trò quan trọng trong các phép toán hình thái học morphology. Ảnh nhị phân dễ dàng để xử lý và thao tác, trong khi ảnh độ xám cung cấp thông tin về độ sáng và tối của các điểm ảnh, giúp xác định các đối tượng có kích thước khác nhau trong ảnh. Các phép toán trên ảnh nhị phân thường được sử dụng để loại bỏ các đối tượng nhiễu và cấu trúc không mong muốn, trong khi các phép toán trên ảnh độ xám thường được sử dụng để phân tích các đặc trưng hình thái của các đối tượng trong không gian mức xám.

## BÀI TẬP MỘT THỰC HÀNH – BINARY IMAGE

### III. Các toán tử Morphology được sử dụng trong bài thực hành một

Morphology là một phương pháp xử lý hình ảnh được sử dụng để phân tích và xử lý các cấu trúc hình học trong hình ảnh như đường viền, tăng cường cấu trúc đối tượng, hình dạng và kích thước của các đối tượng. Toán tử morphology được sử dụng để thực hiện các thao tác như xóa bỏ nhiễu, tinh giản hình ảnh, phóng to hoặc thu nhỏ các đối tượng, tìm kiếm đường biên, phát hiện vật thể, ...

Các toán tử morphology được sử dụng rộng rãi trong xử lý hình ảnh để tạo ra các phép biến đổi và xử lý ảnh phức tạp như phát hiện khu vực, tách đối tượng, định lượng đối tượng, nâng cao chất lượng ảnh,... Dựa trên cơ sở phép toán đại số của các toán tử phi tuyến tác động trên hình dáng đối tượng (Algebra of non-linear operator), thay thế phép tích chập (Linear algebraic system of convolution) để thực hiện các phép biến đổi.

#### 1. Toán tử Dilation

Dilation (Dilate): Tăng kích thước của một vật thể bằng cách thêm các pixel vào vật thể để làm nó dày và rộng hơn. Mục đích chính là lấp kẽ hở, lỗ hỏng của đối tượng.

Trong xử lý ảnh, toán tử dilation được sử dụng để mở rộng các đối tượng trong hình ảnh bằng cách thêm vào các điểm còn lại trong khu vực lân cận của đối tượng. Các điểm này được thêm vào để làm tăng kích thước của đối tượng và giúp đối tượng trở nên mượt mà hơn.

$$X \oplus B = \{p \in \varepsilon^2 : p = x + b, x \in X \text{ and } b \in B\}$$

$$X \oplus B = \{p \in \varepsilon^2 : (\hat{B})_p \cap X \neq \emptyset\}$$

$$X \oplus B = \bigcup_{b \in B} X_b$$

### Tính chất

- Giao hoán:  $X \oplus B = B \oplus X$
- Kết hợp:  $X \oplus (B \oplus D) = (X \oplus B) \oplus D$
- **Hội tập tịnh tiến:**  $X \oplus B = \bigcup_{b \in B} X_b$
- Bất biến với phép tịnh tiến:  $(X_h \oplus B) = (X \oplus B)_h$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \oplus B \subseteq Y \oplus B$

Trong thư viện OpenCV, toán tử dilation được thực hiện bằng hàm cv2.dilate(). Hàm này có cú pháp như sau:

`cv2.dilate(src, dst, kernel, anchor, iterations, borderType, borderColor)`

Trong đó:

- src: là ảnh đầu vào (input image).
- dst: là ảnh đầu ra (output image).
- kernel: là ma trận (kernel) cỡ lớn được sử dụng cho toán tử dilation. Kernel này được tạo bằng hàm cv::getStructuringElement().
- anchor: là điểm trung tâm của kernel.
- iterations: là số lần lặp lại thực hiện toán tử dilation.
- borderType: là kiểu viền (border type) được sử dụng khi kernel trượt ra khỏi ảnh đầu vào. Có thể sử dụng các giá trị như cv::BORDER\_CONSTANT, cv::BORDER\_REPLICATE, cv::BORDER\_REFLECT,...

- `borderValue`: là giá trị được sử dụng khi kiểu viền là `cv::BORDER_CONSTANT`.

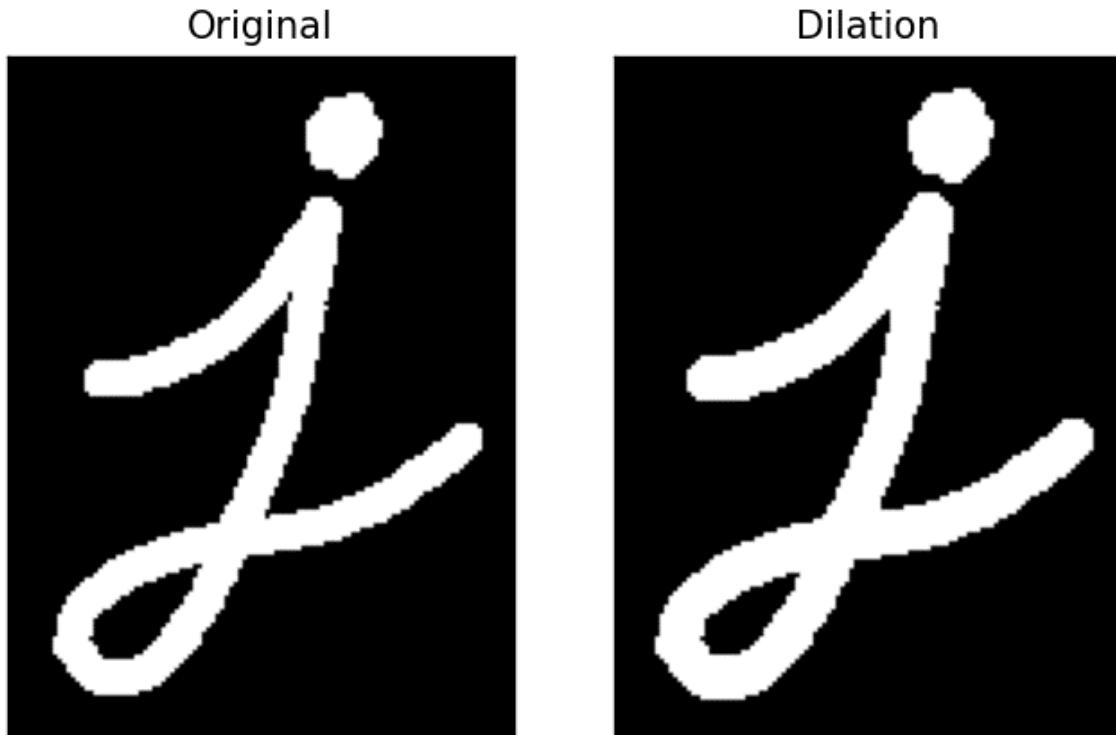
```
binary_img, _ = Read_Img()
exe = Choose_Kernel()

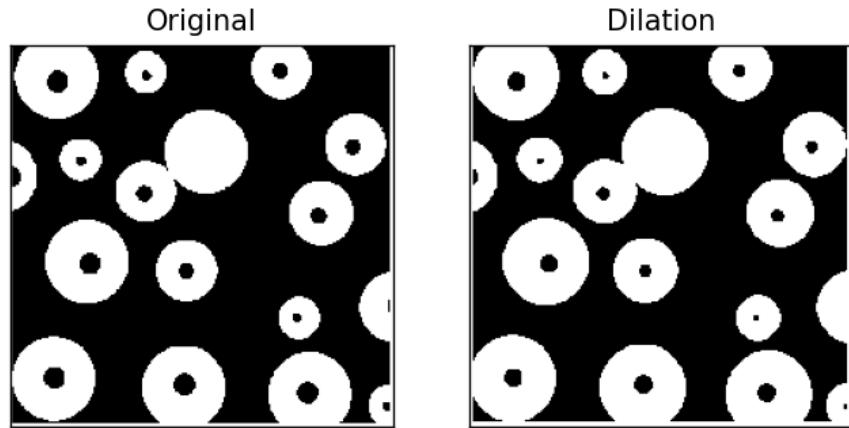
# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử dilation
dilation = cv2.dilate(binary_img, Kernel(exe), iterations = 1) #iterations = 1 nghĩa là bào mòn 1 lần

# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Dilation"), plt.imshow(dilation, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

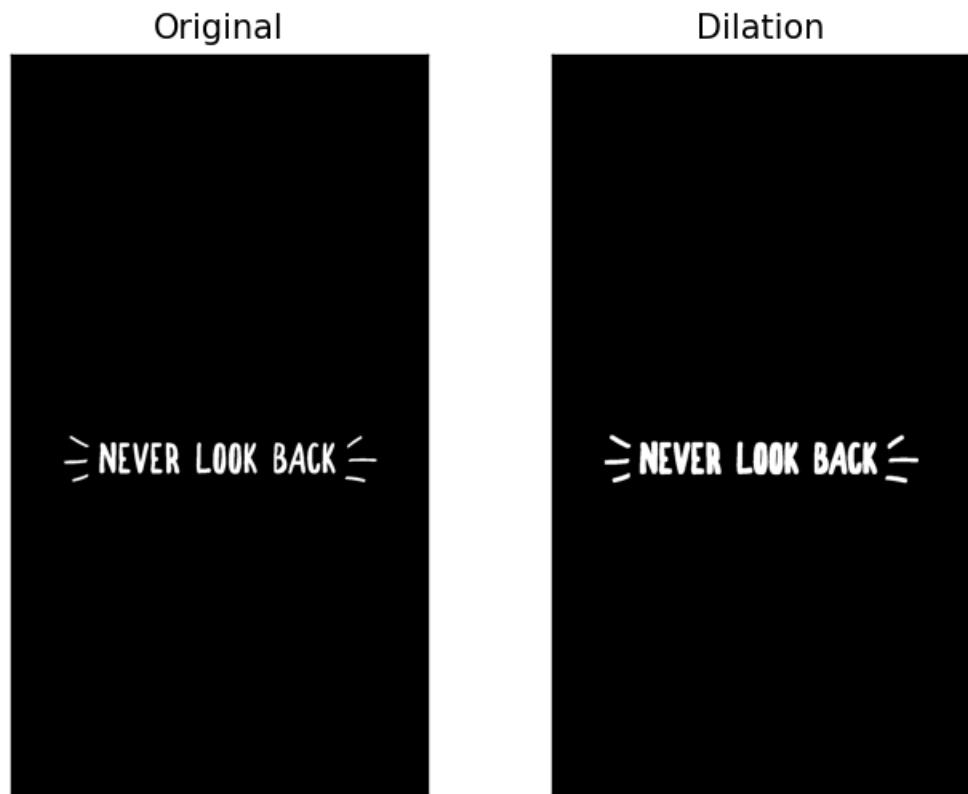
Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh `input.png`, `input1.png`, `quote.png`)

Dùng mặt nạ ker





Dùng mặt nạ kernel



Để thực hiện toán tử dilation trên một ảnh mà không sử dụng thư viện OpenCV, chúng ta cần thực hiện các bước sau:

- Định nghĩa kernel: Ta cần định nghĩa kernel (ma trận) có kích thước và hình dáng tùy ý. Kernel này có thể có các giá trị nhị phân, với giá trị bằng 1 tại các vị trí cần được thực hiện dilation, và giá trị bằng 0 tại các vị trí khác.
- Duyệt ảnh đầu vào: Ta duyệt qua từng điểm ảnh của ảnh đầu vào. Với mỗi điểm ảnh, ta lấy phần của ảnh tương ứng với kernel và thực hiện phép tính AND giữa kernel và phần ảnh này.
- Tính toán giá trị điểm ảnh đầu ra: Giá trị điểm ảnh đầu ra được tính bằng giá trị lớn nhất trong số các giá trị được tính toán ở bước 2.
- Lưu kết quả: Giá trị điểm ảnh tính được ở bước 3 được lưu vào vị trí tương ứng trong ảnh đầu ra.

Sau khi thực hiện các bước trên cho toàn bộ điểm ảnh của ảnh đầu vào, ta sẽ thu được ảnh đầu ra chứa đối tượng đã được mở rộng kích thước theo kernel khi thực hiện toán tử dilation.

```
# thực hiện phép tích chập 2D với ma trận kernel và ảnh đầu vào
def convolution_dilation(ker_mat,img):
    #lấy kích thước của ma trận kernel ker_mat
    k_rows, k_cols = ker_mat.shape
    #lấy kích thước của ảnh đầu vào img
    rows, cols = img.shape

    # tạo ma trận padded image với giá trị 0
    img_pad=np.zeros((rows + k_rows - 1, cols + k_cols - 1))
    img_pad[k_rows-int(k_rows / 2) - 1:k_rows - 1 - int(k_rows / 2) + rows, k_cols - int(k_cols / 2) - 1:k_cols - int(k_cols / 2) - 1 + cols] = img

    # tạo ma trận output với giá trị 0
    op_img=np.zeros_like(img)

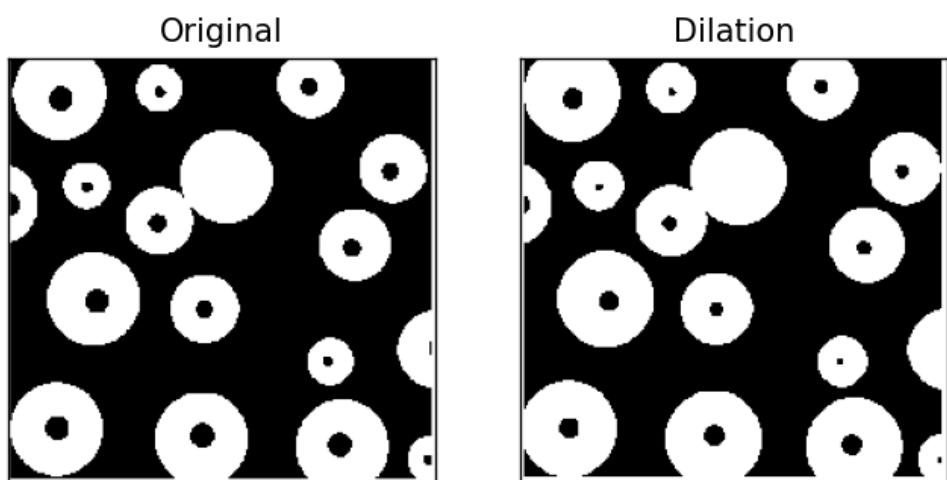
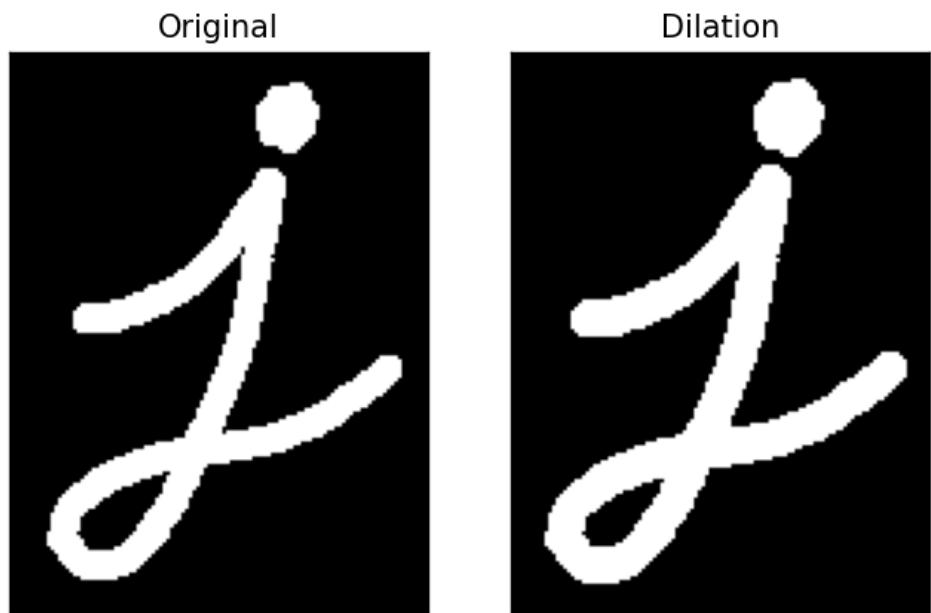
    # thực hiện dilation bằng cách tính tổng giá trị của các phần tử kernel và pirowsel ảnh tại vị trí tương ứng
    for i in range(int(k_rows / 2), int(k_rows / 2) + rows):
        for j in range(int(k_cols / 2), int(k_cols / 2) + cols):
            window = (ker_mat * img_pad[i - int(k_rows / 2): i - int(k_rows / 2) + k_rows, j - int(k_cols / 2): j - int(k_cols / 2) + k_cols]).sum()

            # Check if the kernel is fully contained within the image
            if(window > 0):
                op_img[i-int(k_rows/2),j-int(k_cols/2)] = 1
            else:
                op_img[i-int(k_rows/2),j-int(k_cols/2)] = 0

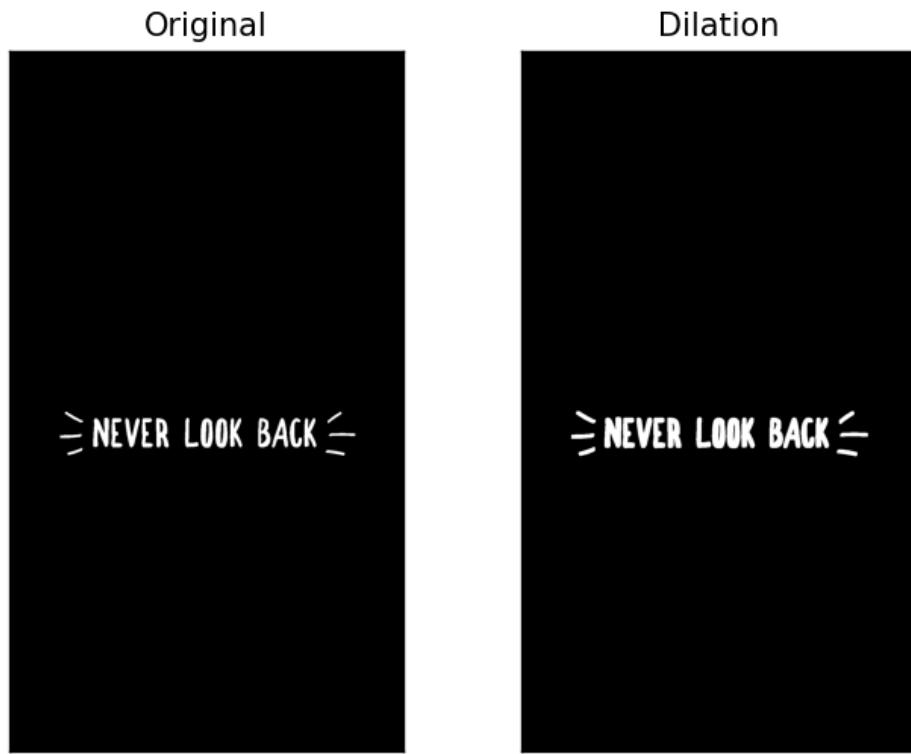
    return op_img
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1



Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử dilation ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

## 2. Toán tử Erosion

Erosion (Erode): Loại bỏ các pixel ngoại vi của một vật thể để thu nhỏ nó lại. Mục đích chính là loại bỏ chi tiết không thích hợp (theo nghĩa về kích thước).

Trong xử lý ảnh, toán tử erosion được sử dụng để co lại kích thước của các đối tượng trong hình ảnh bằng cách loại bỏ các điểm nằm ở lân cận của đối tượng. Các điểm này được loại bỏ để giảm kích thước của đối tượng và giúp đối tượng trở nên sắc nét hơn.

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

## Tính chất

- Chống mở rộng:  $(0,0) \in B \Rightarrow X\Theta B \subseteq X$
- Không giao hoán:  $X\Theta B \neq B\Theta X$
- **Giao tập tịnh tiến ngược:**  $X\Theta B = \bigcap_{b \in B} X_{-b}$
- Bất biến với phép tịnh tiến:  $X_h\Theta B = (X\Theta B)_h$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X\Theta B \subseteq Y\Theta B$

Trong thư viện OpenCV, chúng ta có thể sử dụng hàm erode() để thực hiện toán tử erosion trên ảnh. Hàm này có các tham số đầu vào như sau:

`cv2.erode(src, kernel, dst=None, anchor=None, iterations=None, borderType=None, borderColor=None)`

Trong đó:

- src: ảnh đầu vào cần được thực hiện toán tử erosion.
- kernel: ma trận kernel có kích thước và hình dáng tùy ý, được sử dụng để thực hiện toán tử erosion.
- dst: ảnh đầu ra sau khi thực hiện toán tử erosion. Nếu không được cung cấp, hàm sẽ tạo ra ảnh đầu ra mới với cùng kích thước và kiểu dữ liệu với ảnh đầu vào.
- anchor: điểm neo, được sử dụng để xác định vị trí của kernel trong quá trình tính toán. Mặc định là (-1, -1), nghĩa là điểm neo ở trung tâm của kernel.
- iterations: số lần lặp lại toán tử erosion. Mặc định là 1.
- borderType: cách xử lý viền ảnh, có các giá trị như cv2.BORDER\_CONSTANT, cv2.BORDER\_REPLICATE, cv2.BORDER\_REFLECT, cv2.BORDER\_WRAP, v.v.
- borderColor: giá trị được sử dụng khi borderType được chọn là cv2.BORDER\_CONSTANT.

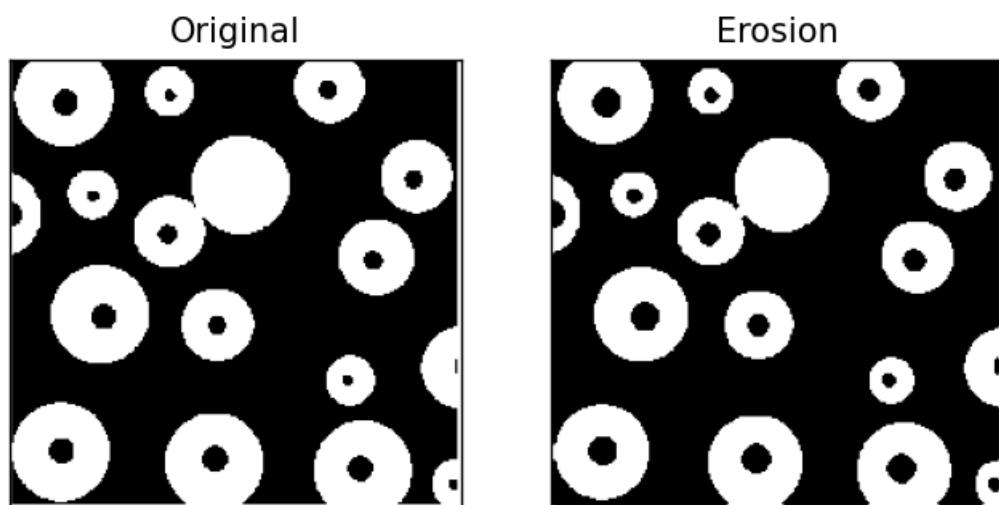
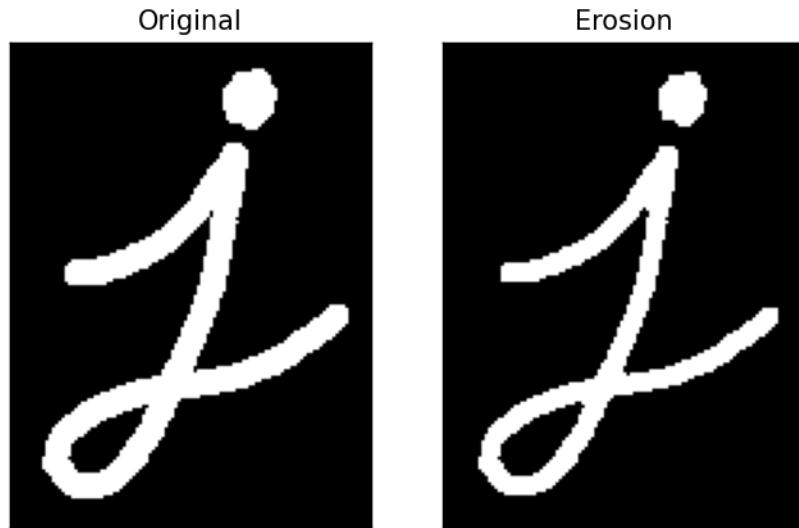
```
binary_img, _ = Read_Img()
exe = Choose_Kernel()

# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử erosion
erosion = cv2.erode(binary_img, Kernel(exe), iterations = 1)

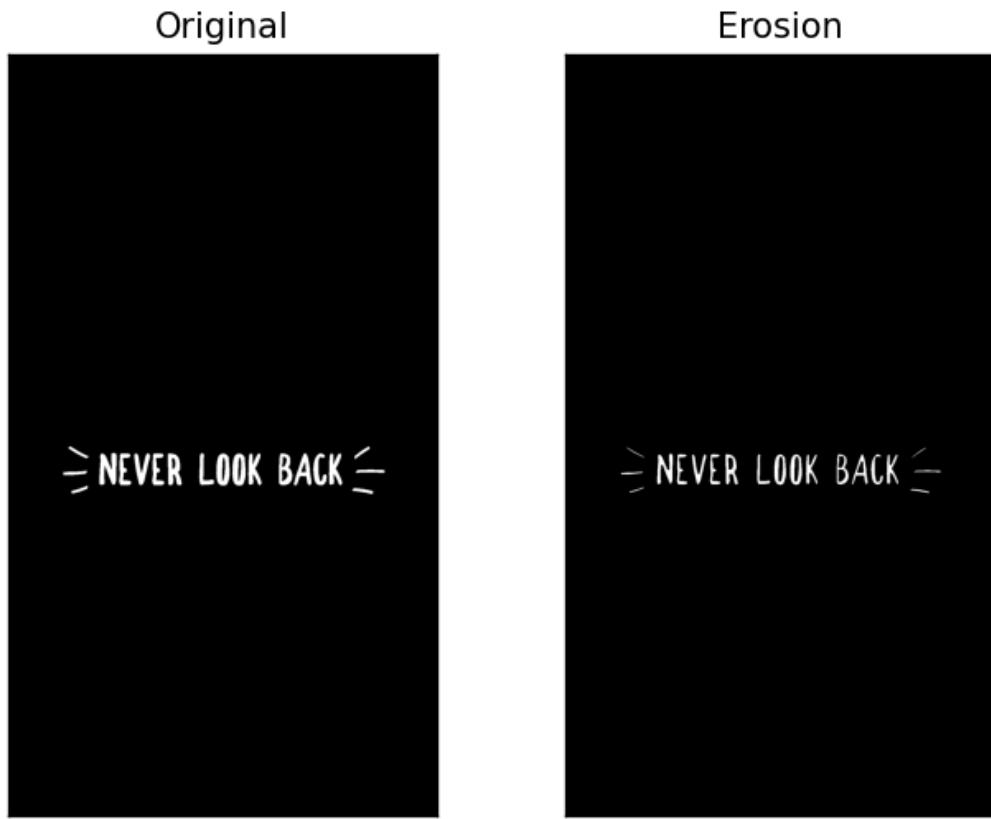
# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Erosion"), plt.imshow(erosion, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Khi không sử dụng thư viện OpenCV, chúng ta có thể thực hiện toán tử erosion bằng cách sử dụng các phép toán ma trận đơn giản như tích chập hoặc xử lý ma trận. Cụ thể, để thực hiện toán tử erosion trên ảnh, chúng ta có thể làm như sau:

- Đọc ảnh đầu vào và chuyển đổi thành một ma trận NumPy.
- Định nghĩa ma trận kernel, có kích thước và hình dáng tùy ý.
- Duyệt qua từng điểm ảnh của ảnh đầu vào.
- Tại mỗi điểm ảnh, tạo ra một vùng lân cận bằng cách lấy ma trận con trong ảnh đầu vào, có kích thước bằng kích thước của kernel và tâm là điểm đó.
- Tính toán giá trị tối thiểu trong vùng lân cận đó.
- Gán giá trị tối thiểu này vào điểm ảnh tương ứng trong ảnh đầu ra.
- Lặp lại quá trình từ bước 3 đến bước 6 cho tất cả các điểm ảnh trong ảnh đầu vào.
- Hiển thị ảnh đầu vào và ảnh đầu ra sau khi thực hiện toán tử erosion.

```

def erosion(img, kernel):
    # Lấy kích thước ảnh và kernel
    rows, cols = img.shape
    k_rows, k_cols = kernel.shape

    # Khởi tạo ảnh kết quả
    result = np.zeros((rows, cols))

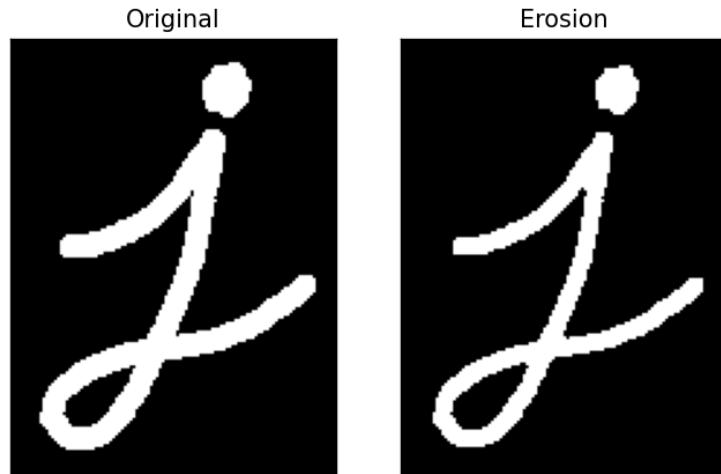
    # Duyệt qua từng pixel của ảnh đầu vào
    for i in range(rows):
        for j in range(cols):
            # Tìm giá trị nhỏ nhất trong vùng lân cận
            min_val = 255
            for m in range(k_rows):
                for n in range(k_cols):
                    if kernel[m, n] != 0:
                        x = i + m - k_rows // 2
                        y = j + n - k_cols // 2
                        if x >= 0 and x < rows and y >= 0 and y < cols:
                            if img[x, y] < min_val:
                                min_val = img[x, y]
            result[i, j] = min_val

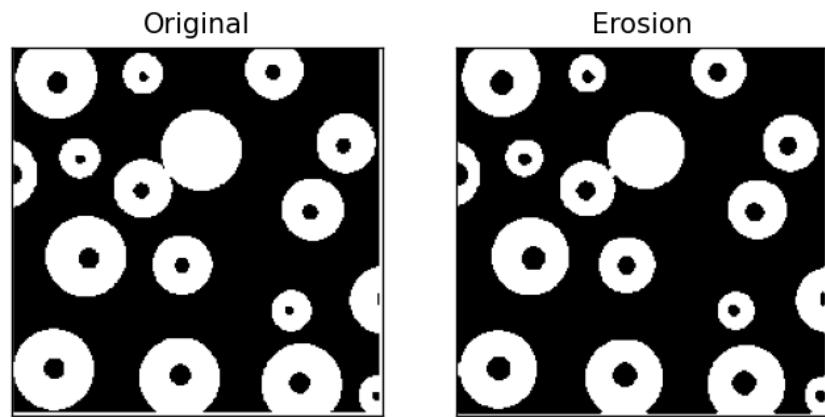
    return result.astype(np.uint8)

```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử erosion ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

### 3. Toán tử Opening

**Opening:** Thực hiện erosion trước rồi dilation để loại bỏ các chi tiết không mong muốn như những điểm nhiễu và kẻ đường nhỏ trong vật thể. Mục đích chính là làm trơn biên đối tượng, loại eo hẹp và chỗ lồi mỏng.

Toán tử opening là kết hợp giữa toán tử erosion và toán tử dilation trên ảnh. Kỹ thuật opening được sử dụng để loại bỏ các chi tiết nhỏ và giảm thiểu nhiễu trên ảnh.

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

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

#### Tính chất

- Chống mở rộng:  $(0,0) \in B \Rightarrow X \circ B \subseteq X$
- Lũy đẳng:  $X \circ B = (X \circ B) \circ B$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \circ B \subseteq Y \circ B$

Trong thư viện OpenCV, chúng ta có thể sử dụng hàm `cv2.morphologyEx(src, op, kernel)` để thực hiện toán tử opening trên ảnh. Hàm này nhận vào 4 đối số:

- src: ảnh đầu vào.
- op: loại toán tử morphological operation cần thực hiện. Ở đây, ta sử dụng giá trị `cv2.MORPH_OPEN` để thực hiện toán tử opening.

- kernel: ma trận kernel được sử dụng cho toán tử morphological operation. Đây là ma trận 2 chiều với các giá trị phần tử nằm trong khoảng [0, 1] hoặc [0, 255]. Kích thước của kernel phải lẻ để kernel có thể có một điểm trung tâm.

```

binary_img, _ = Read_Img()
exe = Choose_Kernel()

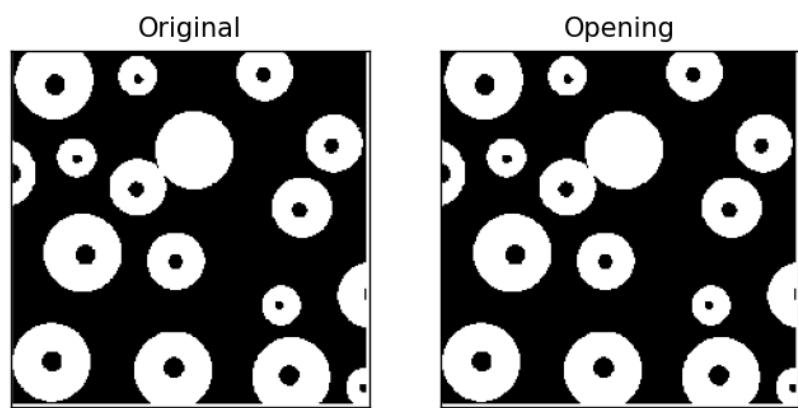
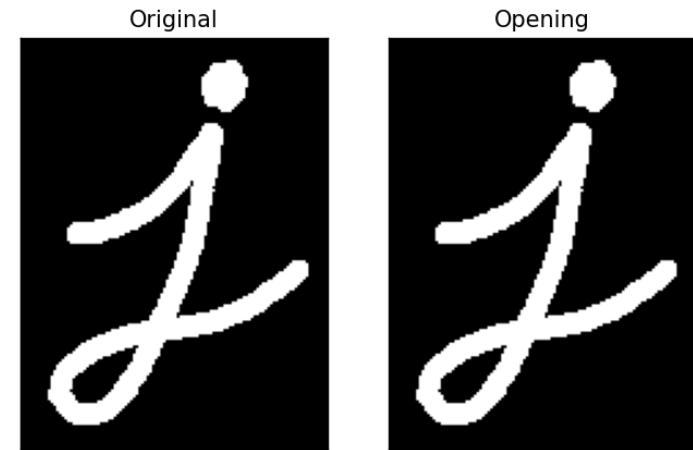
# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử opening
opening = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, Kernel(exe))

# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Opening"), plt.imshow(opening, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()

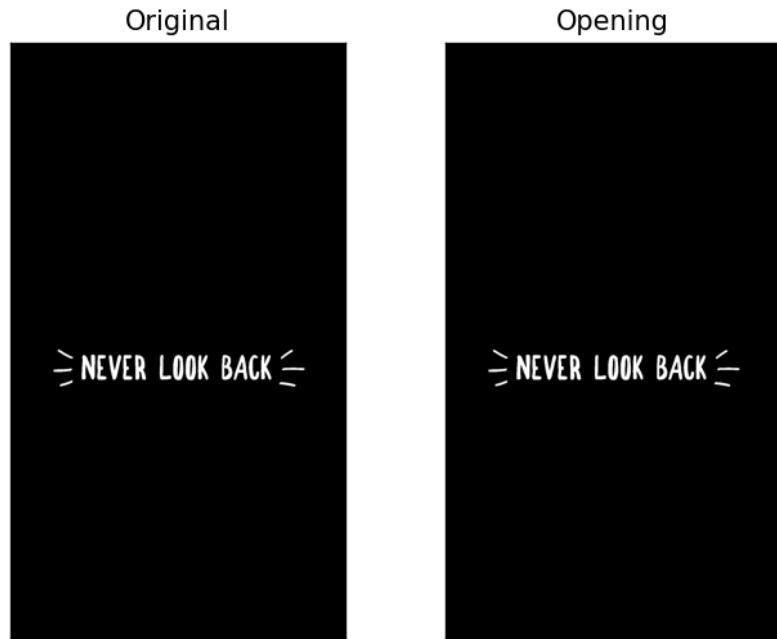
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ ker



## Dùng mặt nạ kernel



Cách thực hiện toán tử opening trên ảnh như sau:

- Đọc ảnh đầu vào và chuyển đổi thành một ma trận NumPy.
- Định nghĩa ma trận kernel, có kích thước và hình dáng tùy ý.
- Thực hiện toán tử erosion trên ảnh đầu vào bằng cách sử dụng kernel đã định nghĩa ở bước 2.
- Thực hiện toán tử dilation trên kết quả erosion ở bước 3 bằng cách sử dụng kernel đã định nghĩa ở bước 2.
- Hiển thị ảnh đầu vào và ảnh đầu ra sau khi thực hiện toán tử opening.

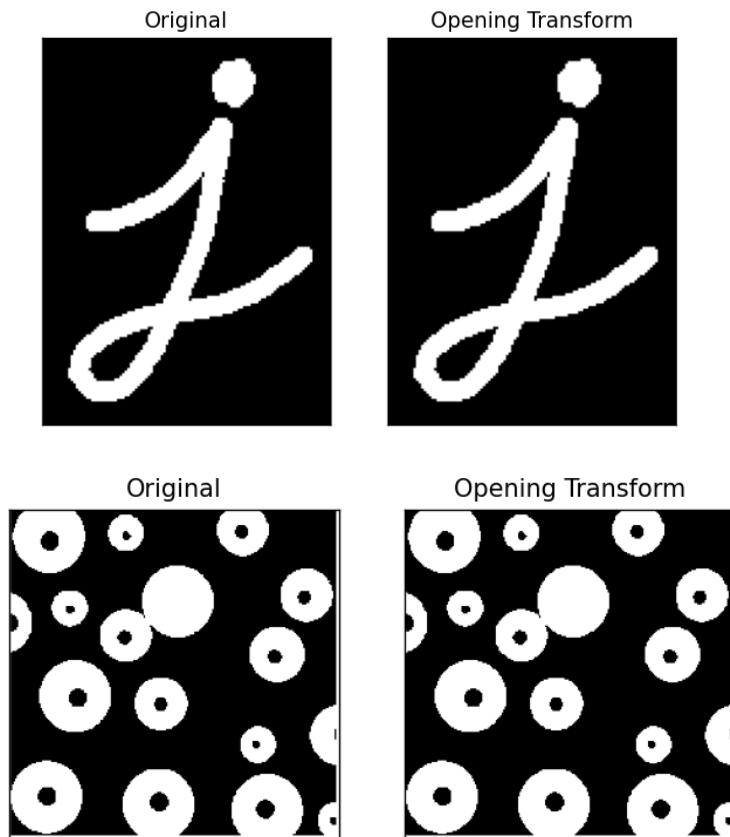
Kết quả của toán tử opening là ảnh đã được xử lý sao cho các đối tượng trên ảnh được giữ nguyên hình dạng và kích thước cơ bản của chúng, nhưng các đối tượng nhỏ hơn hoặc bị nhiễu đã được loại bỏ.

```
from header_bin_noncv import *
from dilation import convolution_dilation
from erosion import erosion

# Function to perform opening on image
def opening(image, kernel):
    #image_eroded = erosion(image, kernel)
    return convolution_dilation(kernel, np.array(erosion(image, kernel)))
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1

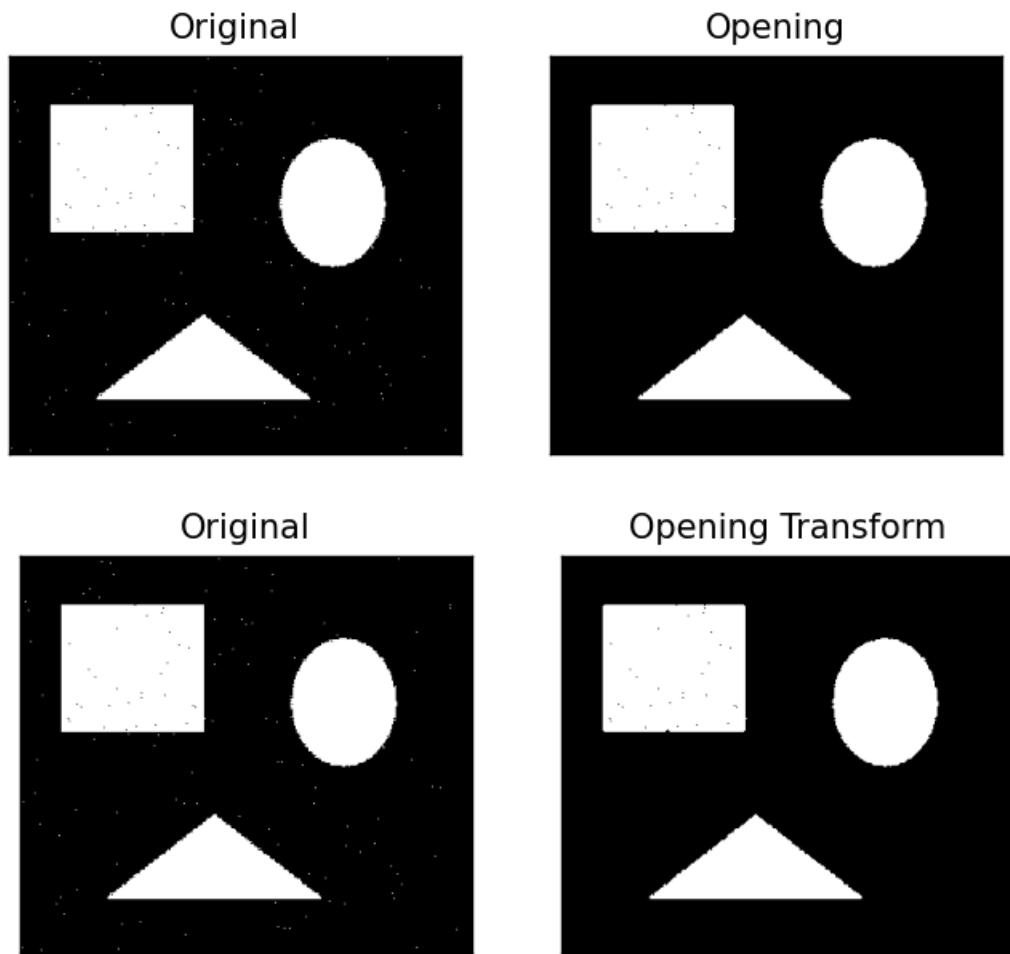


Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử opening ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

Thử lọc nhiễu ảnh (sử dụng ảnh noise.jpg và mặt nạ ker/ kernel\_1)



➔ Ảnh được lọc nhiễu trắng ở vùng ngoài các hình khối

#### 4. Toán tử Closing

Closing: Thực hiện dilation trước rồi erosion để đóng các lỗ trống và nối các đường viền trong vật thể.

Trong xử lý ảnh, toán tử closing là một phép biến đổi hình thái học được sử dụng để loại bỏ các lỗ hổng (holes) nhỏ hoặc các đối tượng nhỏ không mong muốn khác trong các vùng liền kề.

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

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

### Tính chất

- Mở rộng:  $(0,0) \in B \Rightarrow X \subseteq X \bullet B$
- Lũy đẳng:  $X \bullet B = (X \bullet B) \bullet B$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \bullet B \subseteq Y \bullet B$

"Toán tử closing" trong OpenCV là một phép toán kết hợp giữa phép toán dilation (phóng to) và phép toán erosion (co lại). Phép toán dilation được sử dụng để mở rộng các đối tượng trắng (foreground) trong ảnh, trong khi phép toán erosion được sử dụng để thu nhỏ các đối tượng trắng. Khi kết hợp cả hai phép toán, toán tử closing sẽ giúp loại bỏ các lỗ hổng nhỏ và kết nối các phần của đối tượng mà bị phân tách trong ảnh.

`cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)`

Trong đó, img là ảnh đầu vào, kernel là kernel được sử dụng để thực hiện toán tử closing.

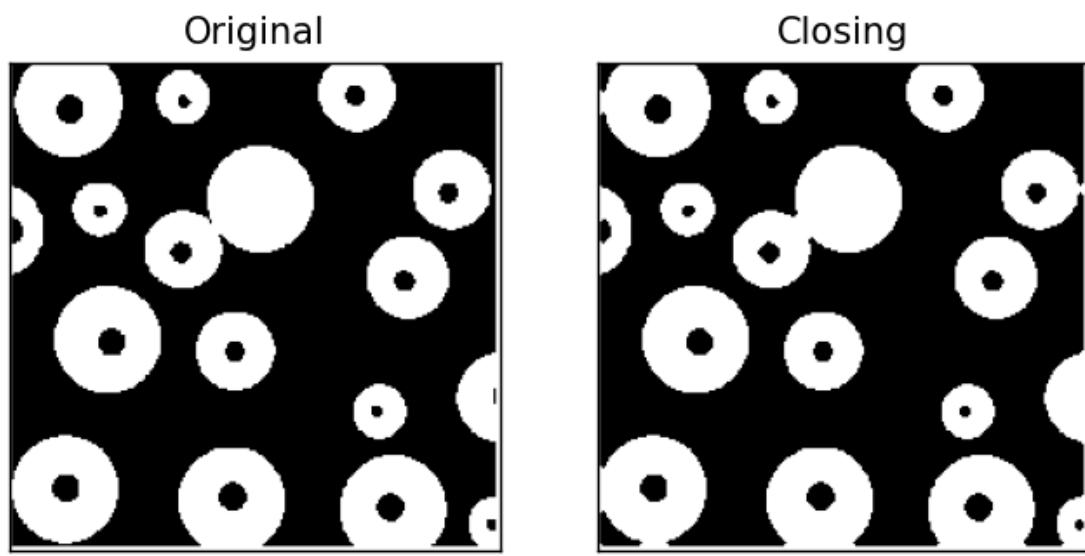
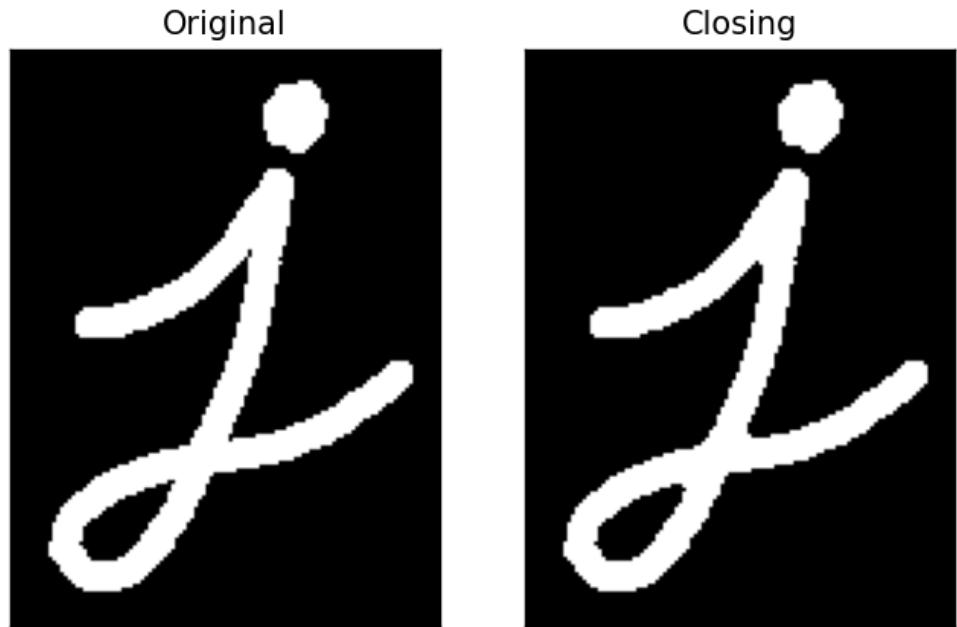
```
binary_img, _ = Read_Img()
exe = Choose_Kernel()

# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử closing
closing = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, Kernel(exe))

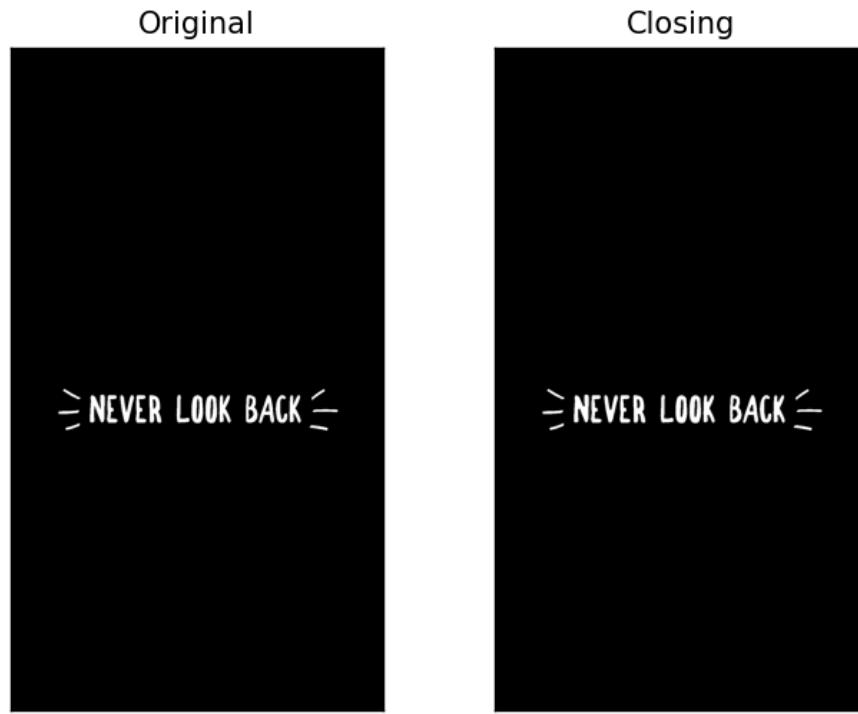
# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Closing"), plt.imshow(closing, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ ker



## Dùng mặt nạ kernel



Để thực hiện toán tử closing trong xử lý ảnh khi không sử dụng thư viện OpenCV, bạn có thể thực hiện các bước sau:

- Xác định kích thước của một cửa sổ (window) và hình dạng của phần tử cấu thành cửa sổ này. Hình dạng phổ biến của phần tử là hình vuông hay hình tròn.
- Duyệt qua toàn bộ ảnh và thực hiện phép giãn nở (dilation) trên từng điểm ảnh của ảnh đầu vào bằng cách sử dụng phần tử đã xác định ở bước trước đó. Phép giãn nở tại một điểm ảnh sẽ gán giá trị cao nhất trong cửa sổ đang xét cho điểm ảnh đó.
- Thực hiện phép co ngắn (erosion) trên ảnh đã được giãn nở ở bước trên bằng cách sử dụng cùng phần tử đã xác định ở bước trên. Phép co ngắn tại một điểm ảnh sẽ gán giá trị thấp nhất trong cửa sổ đang xét cho điểm ảnh đó.

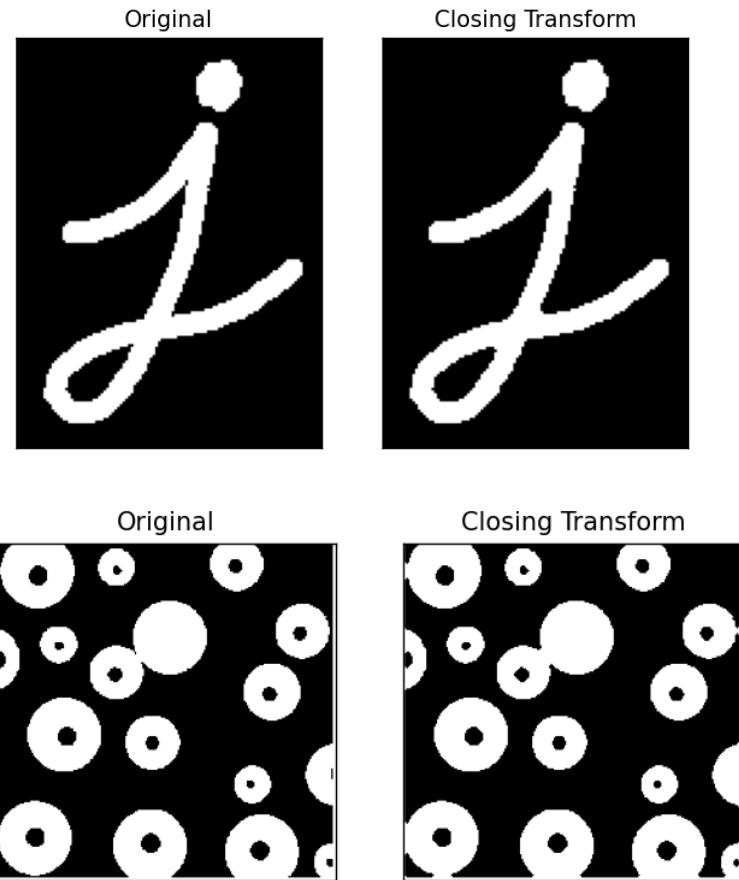
Kết quả sau khi thực hiện toán tử closing là ảnh đã được loại bỏ các lỗ hổng nhỏ trong vật thể được nhận dạng trong ảnh.

```
from header_bin_noncv import *
from dilation import convolution_dilation
from erosion import erosion

# Function to perform closing on image, given image, structuring element and origin
def closing(image, kernel):
    #image_dilated = convolution_dilation(kernel, image)
    #op_img = erosion(image_dilated, kernel)
    return erosion(np.array(convolution_dilation(kernel, image)), kernel)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1

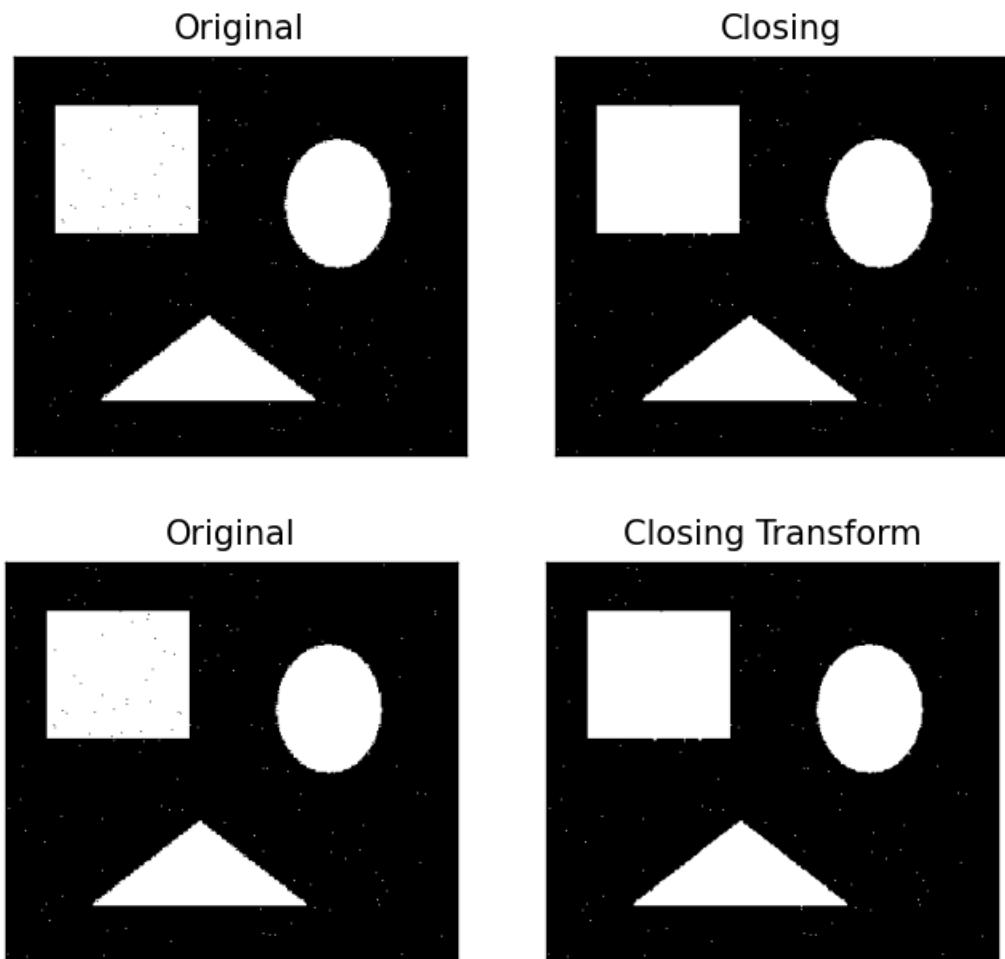


Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử closing ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

Thử lọc nhiễu ảnh (sử dụng ảnh noise.jpg và mặt nạ ker/ kernel\_1)



➔ Ảnh được lọc nhiễu đen ở vùng trong các hình khối

## 5. Toán tử Hit or Miss

Toán tử Hit or Miss là một loại toán tử được sử dụng trong xử lý hình ảnh để phân đoạn ảnh, hay cụ thể là để tách vật thể trong ảnh. Toán tử này hoạt động dựa trên các mẫu hình ảnh được xác định trước (thường là các mẫu hình hình vuông) và kiểm tra xem các điểm ảnh trong ảnh có khớp với mẫu hình nào hay không.

Các điểm ảnh trong ảnh được coi là "hit" (khớp) nếu chúng giống với mẫu hình được xác định trước, và được coi là "miss" (không khớp) nếu chúng không giống với mẫu hình. Khi toán tử Hit or Miss được áp dụng, các điểm ảnh được xử lý và tách vật thể được cô lập bằng cách lọc bỏ những điểm ảnh không khớp với mẫu hình.

Công thức của toán tử Hit or Miss có thể được biểu diễn như sau:

$$B = (B_1, B_2)$$

$$B_1 = A \quad \text{and} \quad B_2 = W - A$$

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

Trong đó:

- $\Theta$  là toán tử thu nhỏ (erosion) của ảnh f với mẫu hình tương ứng.
- $\wedge$  là toán tử logic AND.
- A và B là các mẫu hình được xác định trước.

Toán tử Hit or Miss có một số tính chất quan trọng sau đây:

- **Tính chất đối xứng:** Toán tử Hit or Miss có tính chất đối xứng, nghĩa là kết quả giữa việc áp dụng Mẫu hình A trước rồi áp dụng Mẫu hình B hoặc ngược lại sẽ cho kết quả tương tự nhau.
- **Tính chất tham số:** Toán tử Hit or Miss được xác định bởi các mẫu hình A và B. Việc thay đổi các mẫu hình này sẽ cho kết quả khác nhau.
- **Tính chất giao hoán:** Toán tử Hit or Miss có tính chất giao hoán, nghĩa là việc thực hiện áp dụng Mẫu hình A trước rồi áp dụng Mẫu hình B hoặc ngược lại sẽ cho kết quả tương tự nhau.
- **Tính chất đơn điệu:** Toán tử Hit or Miss có tính chất đơn điệu, nghĩa là việc tăng cường Mẫu hình A hoặc giảm thiểu Mẫu hình B sẽ không làm mất thông tin về vật thể trong ảnh.
- **Tính chất tương đương:** Toán tử Hit or Miss có tính chất tương đương với việc sử dụng các toán tử dạng như toán tử khuếch đại (dilation) và toán tử lùi (complement) để phân đoạn vật thể trong ảnh.

Các tính chất này cho phép toán tử Hit or Miss được sử dụng hiệu quả trong các ứng dụng xử lý hình ảnh để phân đoạn vật thể trong ảnh.

Trong thư viện OpenCV, toán tử Hit or Miss được cung cấp bởi hàm cv2.morphologyEx() với tham số cv2.MORPH\_HITMISS.

Cú pháp sử dụng như sau:

`cv2.morphologyEx(src, cv2.MORPH_HITMISS, kernel)`

Trong đó:

- src: là ảnh đầu vào.
- kernel: là một kernel (mẫu hình) được xác định trước, thường là các kernel hình vuông hoặc hình chữ nhật.

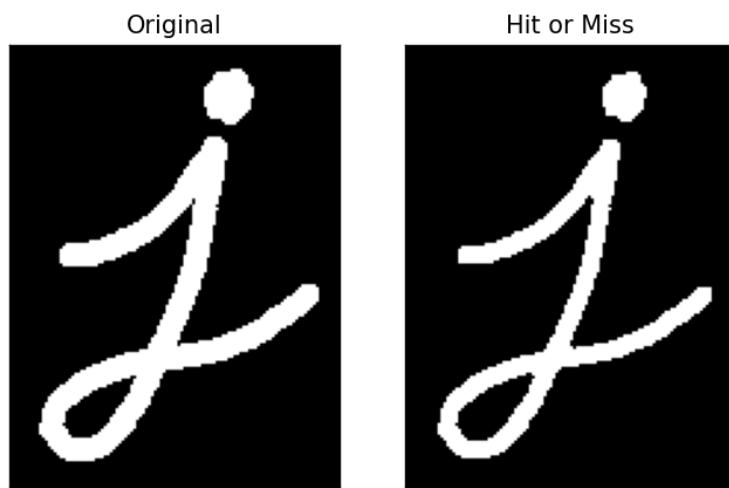
```
binary_img, _ = Read_Img()
exe = Choose_Kernel()

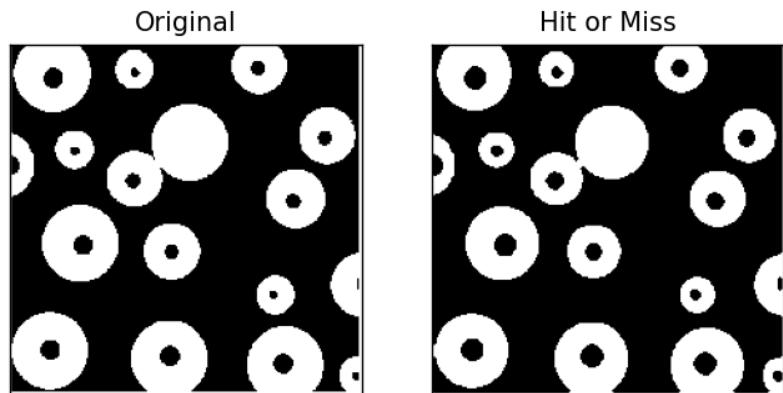
# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Perform Hit-or-Miss Transformation
hit_miss = cv2.morphologyEx(binary_img, cv2.MORPH_HITMISS, Kernel(exe))

# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Hit or Miss"), plt.imshow(hit_miss, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ ker





Dùng mặt nạ kernel



Để áp dụng toán tử Hit or Miss khi không sử dụng thư viện OpenCV, chúng ta có thể thực hiện các bước sau:

- Xác định kernel (mẫu hình) A và kernel B, thường là các kernel hình vuông hoặc hình chữ nhật.
- Lấy ảnh đầu vào và tạo một ma trận kết quả có kích thước bằng với ảnh đầu vào.
- Với mỗi điểm ảnh trong ảnh đầu vào, thực hiện các bước sau:
- Kiểm tra xem kernel A có trùng khớp với một vùng xung quanh điểm ảnh đó không. Nếu không, gán giá trị 0 cho điểm ảnh tương ứng trong ma trận kết quả.

- Kiểm tra xem kernel B có trùng khớp với một vùng xung quanh điểm ảnh đó không. Nếu không, gán giá trị 0 cho điểm ảnh tương ứng trong ma trận kết quả.
- Nếu cả kernel A và kernel B đều trùng khớp với vùng xung quanh điểm ảnh đó, gán giá trị 1 cho điểm ảnh tương ứng trong ma trận kết quả.

```
def hit_or_miss(img, kernel):
    """Hit or Miss operator for binary image"""
    # Tạo các bản sao của kernel
    b1 = kernel.copy()
    b2 = kernel.copy()

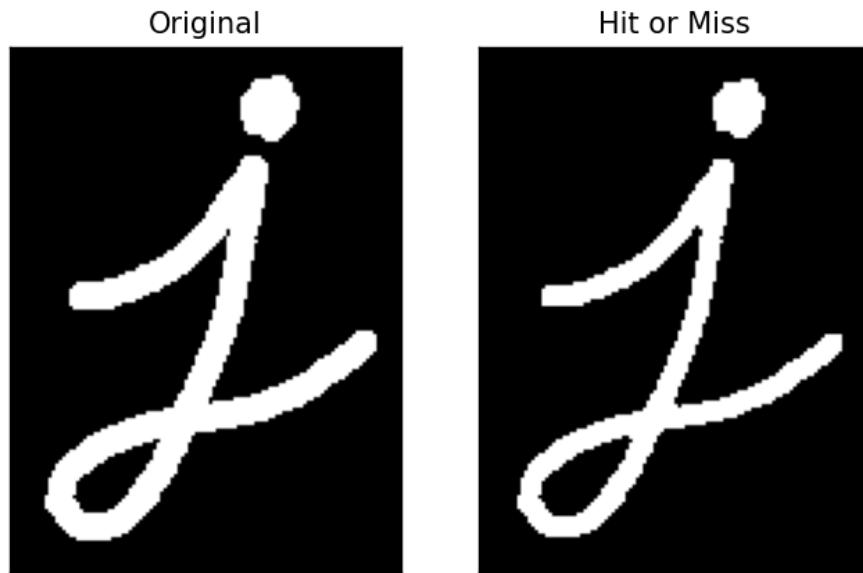
    # Thay đổi giá trị -1 thành 0 trong bản sao b1 của kernel
    b1[b1== -1] = 0
    # Thay đổi giá trị -1 thành 1 và 0 thành -1 trong bản sao b2 của kernel
    b2[b2== -1] = 1
    b2[b2== 0] = -1

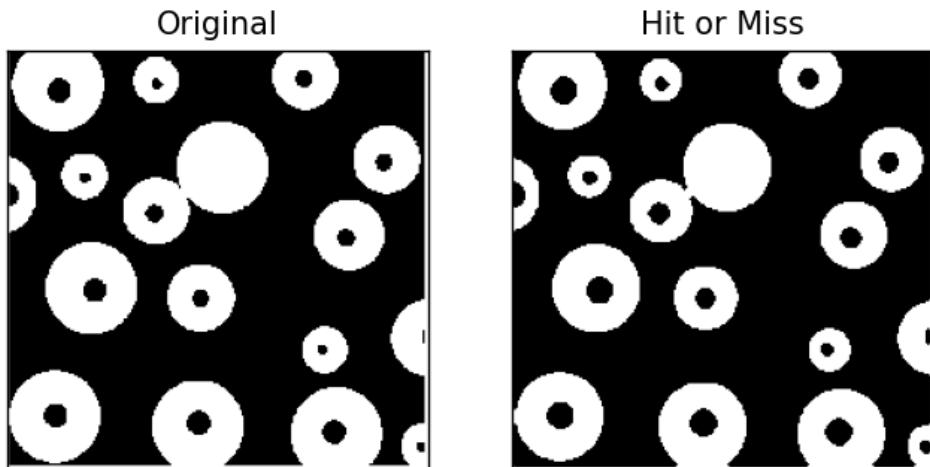
    # Thực hiện phép xói mòn với kernel b1, b2 trên ảnh đầu vào
    a = erosion(img,b1)
    b = erosion(~(np.where(img == 255, 1, img)), b2)

    # Lấy phép giao giữa 2 ảnh đã xói mòn
    hitmiss_img = np.logical_and(a,b)
    return np.uint8(hitmiss_img)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử hit or miss ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

## 6. Toán tử Boundary Extraction

Toán tử Boundary Extraction (hay còn gọi là toán tử Sobel) là một phương pháp xác định biên của các đối tượng trong ảnh. Toán tử này dựa trên việc tính toán đạo hàm của ảnh và sử dụng các bộ lọc Sobel để phát hiện các đường biên.

Toán tử này thực hiện tính toán đạo hàm theo hai hướng theo các trục x và y của ảnh bằng cách áp dụng hai bộ lọc Sobel lên ảnh, sau đó kết hợp kết quả bằng cách tính biên độ (gradient magnitude) của mỗi điểm ảnh, được định nghĩa bằng công thức sau:

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

Sau khi tính toán được biên độ của các điểm ảnh, ta có thể áp dụng một ngưỡng (threshold) để xác định các điểm ảnh thuộc biên của đối tượng trong ảnh. Các điểm ảnh có giá trị biên độ lớn hơn ngưỡng được xem là thuộc biên của đối tượng, và ngược lại.

Các tính chất của toán tử Boundary Extraction (hay còn gọi là toán tử Sobel) bao gồm:

- Phát hiện biên của đối tượng: Toán tử Boundary Extraction được sử dụng để phát hiện các biên của đối tượng trong ảnh. Các biên này có thể được sử dụng để trích xuất các đặc trưng của đối tượng hoặc để phân tích và nhận dạng đối tượng.
- Độ chính xác cao: Toán tử Boundary Extraction cho kết quả với độ chính xác cao trong việc xác định các biên của đối tượng. Tuy nhiên, độ chính xác phụ thuộc vào ngưỡng được sử dụng để phân đoạn ảnh.
- Độc lập với tỷ lệ: Toán tử Boundary Extraction không phụ thuộc vào tỷ lệ của đối tượng trong hình ảnh. Điều này có nghĩa là toán tử có thể xử lý hình ảnh có độ phân giải khác nhau mà không làm thay đổi kết quả trích xuất biên.
- Nhạy cảm với nhiễu: Toán tử Boundary Extraction có thể bị ảnh hưởng bởi nhiễu trong hình ảnh. Nếu hình ảnh có nhiễu nhiều, kết quả trích xuất biên có thể không chính xác.
- Tính đơn giản: Toán tử Boundary Extraction là một phương pháp đơn giản và nhanh chóng để trích xuất biên của đối tượng trong hình ảnh. Điều này làm cho nó trở thành một công cụ hữu ích trong các ứng dụng thời gian thực và xử lý hình ảnh trong các hệ thống nhúng.
- Không phụ thuộc vào màu sắc: Toán tử Boundary Extraction chỉ phụ thuộc vào độ tương phản của hình ảnh để trích xuất biên, nó không phụ thuộc vào màu sắc của đối

tượng. Điều này làm cho nó trở thành một công cụ hữu ích trong việc trích xuất đối tượng trong các hình ảnh màu.

- Phù hợp với các đối tượng đa giác: Toán tử Boundary Extraction được sử dụng phổ biến để trích xuất biên của các đối tượng đa giác trong hình ảnh. Nó cũng có thể được sử dụng để trích xuất biên của các đối tượng khác trong hình ảnh, nhưng độ chính xác sẽ phụ thuộc vào độ phức tạp của đối tượng và nhiễu trong hình ảnh.

Thư viện OpenCV cũng cung cấp toán tử Sobel để tính toán đạo hàm của một ảnh số. Toán tử Sobel thường được sử dụng để tính toán gradient của một ảnh và từ đó phát hiện biên cạnh của các vật thể trong ảnh.

`cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])`

Trong đó:

- src: ảnh đầu vào.
- ddepth: độ sâu của ảnh đầu ra. Nó có thể là một trong các giá trị sau đây: cv2.CV\_8U, cv2.CV\_16U, cv2.CV\_16S, cv2.CV\_32F hoặc cv2.CV\_64F.
- dx và dy: độ dốc theo chiều ngang và chiều dọc. Một trong hai tham số này phải khác 0.
- dst: ảnh đầu ra, chứa kết quả tính toán gradient của ảnh đầu vào.
- ksize: kích thước của kernel sử dụng cho toán tử Sobel. Nó có thể là một số lẻ dương, chẳng hạn như 3 hoặc 5.
- scale và delta: các tham số tinh chỉnh để điều chỉnh độ nhạy và độ tối đa của đạo hàm gradient.
- borderType: xác định cách xử lý các giá trị pixel trên cạnh ảnh. Tham số này có thể là cv2.BORDER\_DEFAULT, cv2.BORDER\_CONSTANT, cv2.BORDER\_REFLECT, cv2.BORDER\_WRAP hoặc cv2.BORDER\_REPLICATE.

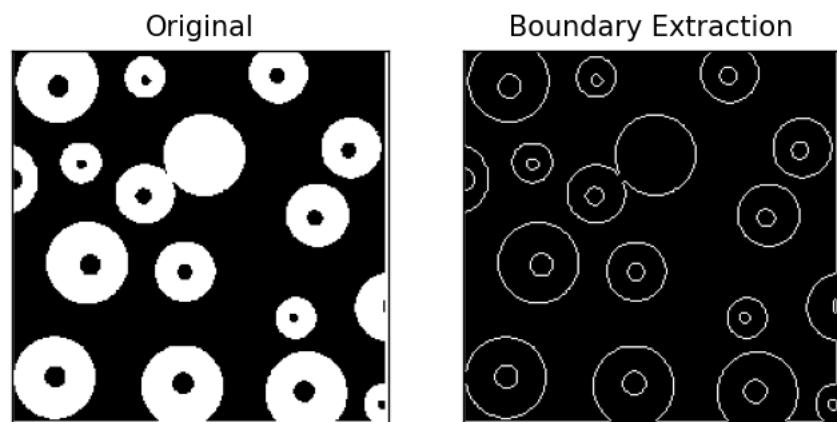
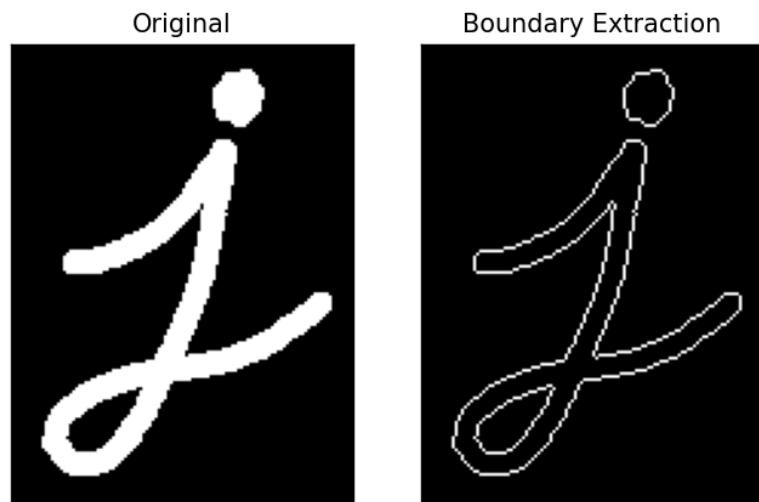
```
binary_img, _ = Read_Img()
exe = Choose_Kernel()

# Thực hiện toán tử tìm biên
erode = cv2.erode(binary_img, Kernel(exe), iterations = 1) # co đối tượng lại
boun_ext = binary_img - erode # áp dụng công thức

# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Boundary Extraction"), plt.imshow(boun_ext, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Có nhiều phương pháp để thực hiện toán tử Boundary Extraction, trong đó phương pháp đơn giản nhất là sử dụng toán tử gradient.

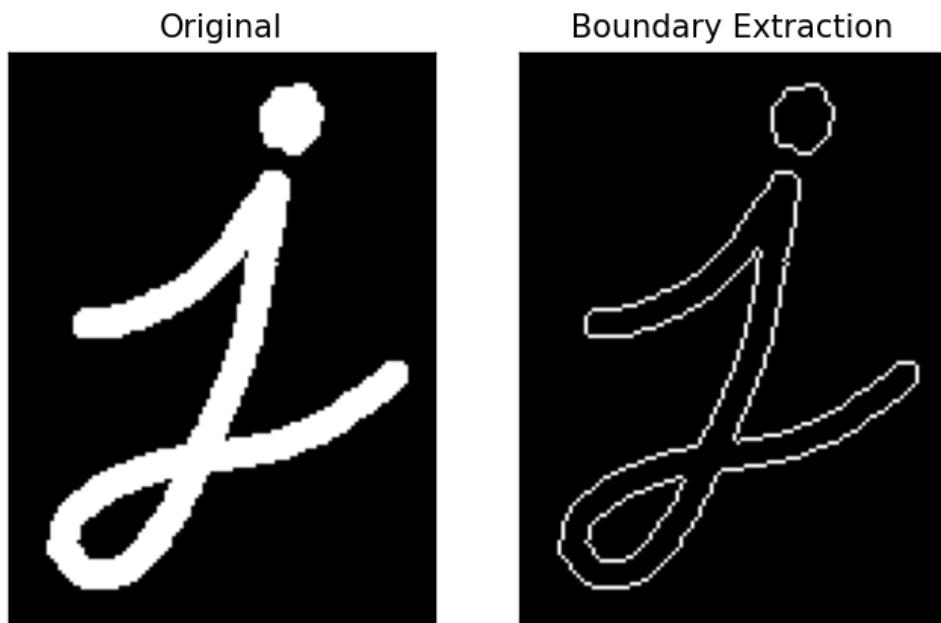
Toán tử gradient tính toán độ dốc tại mỗi điểm ảnh trong ảnh. Độ dốc này có thể được tính bằng cách sử dụng một bộ lọc gradient, chẳng hạn như bộ lọc Sobel hoặc bộ lọc Prewitt. Các bộ lọc này có thể được áp dụng trên cả hai chiều của ảnh, để tính toán độ dốc theo chiều ngang và chiều dọc của ảnh.

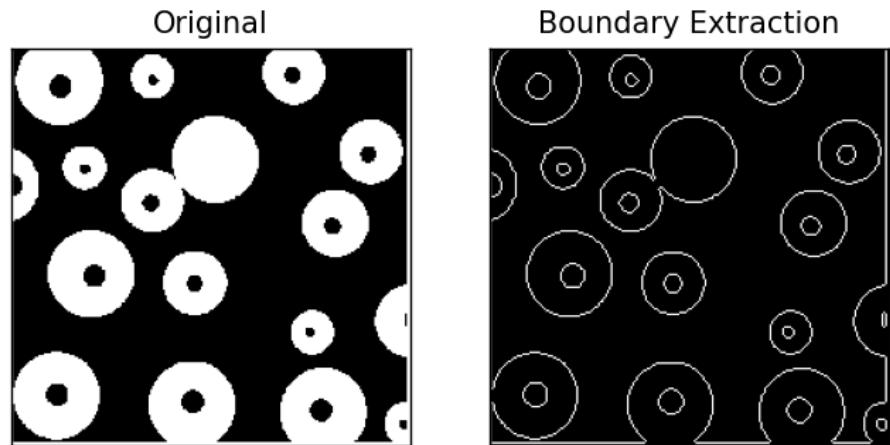
Sau khi tính toán gradient của ảnh, biên cạnh có thể được tìm thấy bằng cách tìm các điểm ảnh có độ dốc lớn hơn ngưỡng nào đó. Điểm ảnh nào có độ dốc lớn hơn ngưỡng đó được coi là một điểm biên cạnh.

```
def boundary_extraction(image, kernel):
    #Thực hiện toán tử tìm biên
    erode = erosion(image, kernel)
    return image - erode
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử boundary extraction ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

## 7. Toán tử Convex Hull

Toán tử Convex Hull là một kỹ thuật xử lý ảnh được sử dụng để tìm ra đường viền lồi (convex) của một tập hợp các điểm trên một hình ảnh. Đường viền lồi được tạo bởi các đoạn thẳng nối các điểm cực trị (các điểm chính là biên của hình ảnh) và các điểm lồi nằm bên trong tập hợp các điểm.

Toán tử Convex Hull thường được sử dụng trong các ứng dụng xử lý ảnh như phân tích hình dạng, nhận dạng đối tượng, và phát hiện vật thể.

$$X_0^i = A, \quad i = 1, 2, 3, 4$$

$$X_k^i = (X_{k-1} \otimes B^i) \cup A, \quad i = 1, 2, 3, 4 \quad \text{and} \quad k = 1, 2, 3, \dots$$

$$D^i = X_{conv}^i \quad (X_k^i = X_{k-1}^i)$$

$$C(A) = \bigcup_{i=1}^4 D^i$$

Toán tử Convex Hull có một số tính chất đáng chú ý như sau:

- Convex Hull là một tập con lồi (convex): Nghĩa là nếu ta lấy hai điểm bất kỳ trong Convex Hull, thì đường thẳng nối hai điểm đó cũng nằm hoàn toàn trong Convex Hull.
- Convex Hull là tập con lồi nhỏ nhất: Nghĩa là Convex Hull chứa tất cả các điểm trong tập hợp ban đầu, và không chứa bất kỳ điểm nào khác nằm bên trong đường viền lồi của nó.
- Convex Hull có thể được sử dụng để phát hiện các đối tượng trong hình ảnh: Như trong trường hợp phát hiện đối tượng trên nền đen, ta có thể sử dụng Convex Hull để xác định đường viền lồi của đối tượng.
- Convex Hull có thể được sử dụng để tính toán diện tích, chu vi, và các thuộc tính khác của tập hợp các điểm.
- Convex Hull cũng được sử dụng để xác định tập con lồi của các điểm trong không gian nhiều chiều (không chỉ trong mặt phẳng hai chiều).

Trong thư viện OpenCV, để tính Convex Hull của một tập hợp các điểm, ta có thể sử dụng hàm cv2.convexHull(). Hàm này có các tham số đầu vào như sau:

`cv2.convexHull(points[, hull[, clockwise[, returnPoints]]])`

- points: là tập hợp các điểm ban đầu, có thể là một mảng numpy hoặc một danh sách các điểm.
- hull: (tùy chọn) là một mảng numpy để lưu kết quả Convex Hull, nếu không được cung cấp thì hàm sẽ trả về Convex Hull dưới dạng một mảng numpy.
- clockwise: (tùy chọn) là một cờ bool cho biết xem Convex Hull được trả về có được sắp xếp theo chiều kim đồng hồ hay ngược chiều kim đồng hồ.
- returnPoints: (tùy chọn) là một cờ bool cho biết xem Convex Hull được trả về có được trả về dưới dạng các điểm hay các chỉ số của các điểm trong tập hợp ban đầu.

Các bước cơ bản để thực hiện toán tử Convex Hull trên một tập hợp các điểm trên một hình ảnh là:

- Xác định các điểm cực trị trên hình ảnh (ví dụ như các điểm trắng trên hình ảnh).
- Xác định các đỉnh của Convex Hull bằng cách sử dụng một thuật toán.
- Kết nối các đỉnh của Convex Hull với nhau để tạo ra đường viền lồi.

```
def main():
    binary_img, _ = Read_Img()

    blur = cv2.blur(binary_img, (3, 3)) #blur the image

    _, thresh = cv2.threshold(blur, 50, 255, cv2.THRESH_BINARY) #apply binary thresholding for blur

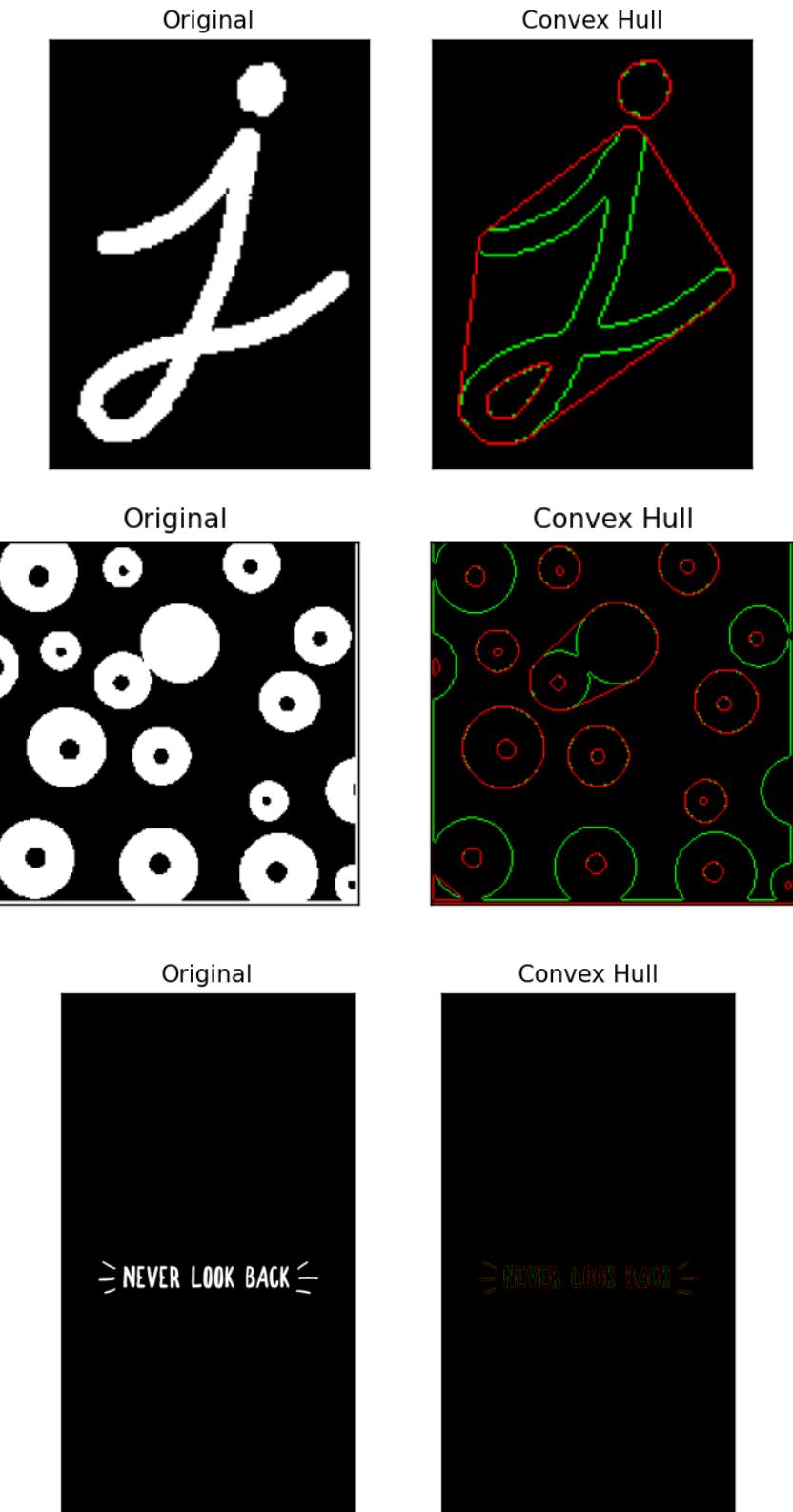
    # tìm contour trong ảnh threshhold
    contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    # Tìm Convex Hull của các contour
    hull = []
    for i in range(len(contours)):
        hull.append(cv2.convexHull(contours[i], False))

    # Vẽ Convex Hull lên hình ảnh gốc
    # create an empty black image
    drawing = np.zeros((thresh.shape[0], thresh.shape[1], 3), np.uint8)

    # draw contours and hull points
    for i in range(len(contours)):
        color_contours = (0, 255, 0) # green - color for contours
        color = (255, 0, 0) # blue - color for convex hull
        # draw ith contour
        cv2.drawContours(drawing, contours, i, color_contours, 1, 8, hierarchy)
        # draw ith convex hull object
        cv2.drawContours(drawing, hull, i, color, 1, 8)
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)



Để tính Convex Hull của một tập hợp các điểm mà không sử dụng thư viện OpenCV, ta có thể sử dụng thuật toán Graham Scan.

Thuật toán Graham Scan hoạt động theo các bước sau:

- Tìm điểm có tọa độ y nhỏ nhất, nếu có nhiều điểm có cùng tọa độ y, chọn điểm có tọa độ x nhỏ nhất.
- Sắp xếp các điểm còn lại theo thứ tự góc tương đối so với điểm được chọn ở bước 1.
- Duyệt qua từng điểm trong tập hợp các điểm đã được sắp xếp, với mỗi điểm:
  - Nếu điểm đó không thể thêm vào Convex Hull, ta loại bỏ nó.
  - Nếu điểm đó có thể thêm vào Convex Hull, ta thêm nó vào và loại bỏ các điểm mà nó che phủ trên Convex Hull.

Sau khi hoàn thành các bước trên, ta sẽ có được tập hợp các điểm trên Convex Hull.

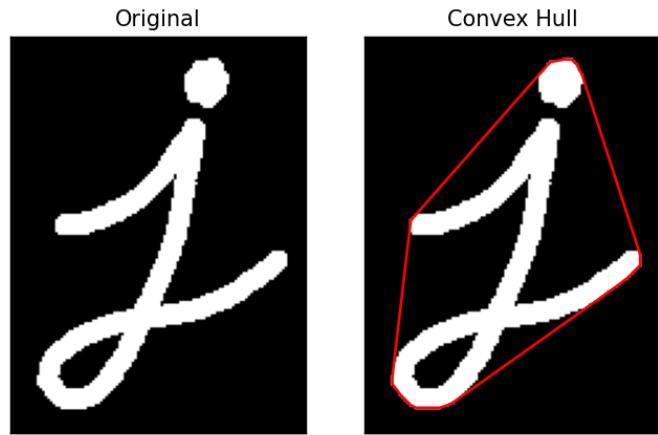
```
def compute_convex_hull(image):
    # Lấy tọa độ của các điểm trên đường biên của hình ảnh
    y, x = np.where(image > 0)
    points = np.column_stack((x, y))

    # Tính Convex Hull của các điểm
    hull = []
    # duyệt qua các điểm trên đường biên của hình ảnh
    for i in range(len(points)):
        # kiểm tra các điểm trên Convex Hull hiện tại
        while len(hull) >= 2 and np.cross(hull[-1] - hull[-2], points[i] - hull[-2]) <= 0:
            hull.pop() # xóa điểm cuối cùng của Convex Hull nếu điểm này không nằm trên đường convex
        hull.append(points[i])

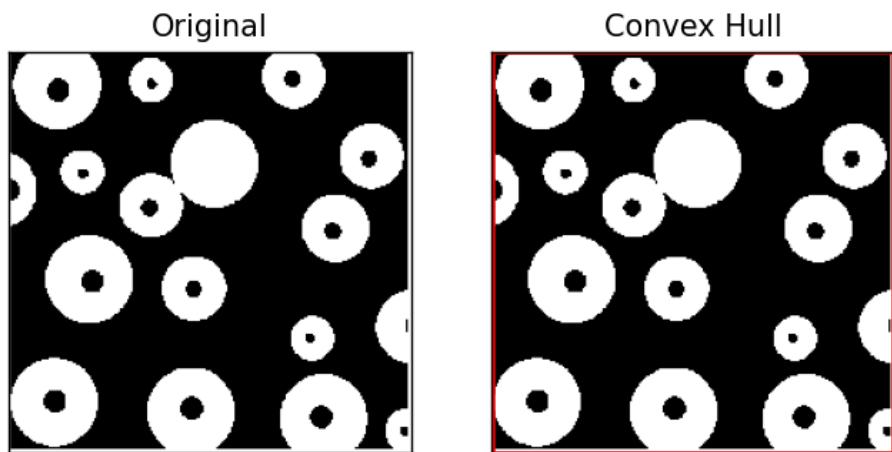
    # chọn vì điểm cuối cùng đã được thêm vào Convex Hull ở vòng lặp trước
    for i in range(len(points)-2, -1, -1):
        while len(hull) >= 2 and np.cross(hull[-1] - hull[-2], points[i] - hull[-2]) <= 0:
            hull.pop()
        hull.append(points[i])

    return np.array(hull)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *quote.png*)



Tuy vậy, vẫn chưa thực hiện được chính xác trên các ảnh còn lại



## 8. Toán tử rút trích thành phần liên thông

Toán tử liên thông (connectivity operator) là một công cụ phân vùng hình ảnh được sử dụng để tách các vùng liên thông trong ảnh. Nó được sử dụng để phát hiện và phân tích các đối tượng có trong hình ảnh bằng cách xác định các pixel nào cùng thuộc về một vùng liên thông bằng cách so sánh giá trị của các pixel lân cận. Nó sẽ tạo ra một vùng liên thông bằng cách kết hợp các pixel có giá trị giống nhau và lân cận với nhau theo một số quy tắc.

Có nhiều loại toán tử liên thông khác nhau được sử dụng trong xử lý ảnh, bao gồm toán tử liên thông 4-neighbor (xét 4 pixel lân cận), toán tử liên thông 8-neighbor (xét 8 pixel lân cận), và toán tử liên thông có hướng (xét các pixel theo hướng xác định). Sự lựa chọn của toán tử liên thông phụ thuộc vào loại ảnh và mục đích sử dụng của người dùng.

$$X_k = (X_{k-1} \oplus B) \cap A \quad k = 1, 2, 3, \dots$$

Các tính chất cơ bản của toán tử liên thông bao gồm:

- Đối xứng: Toán tử liên thông là một toán tử đối xứng, nghĩa là kết quả của việc áp dụng nó trên một đồ thị sẽ không thay đổi nếu chúng ta hoán đổi các đỉnh trong đồ thị đó.
- Liên tục: Toán tử liên thông là một toán tử liên tục, nghĩa là nếu chúng ta xóa một đỉnh từ một đồ thị, thì tập hợp các thành phần liên thông của đồ thị sẽ thay đổi liên tục.
- Tính kết hợp: Toán tử liên thông có tính kết hợp, nghĩa là khi áp dụng nó lần lượt trên các đồ thị con của một đồ thị lớn, kết quả sẽ giống nhau nếu chúng ta áp dụng nó trực tiếp trên đồ thị lớn đó.
- Đóng: Toán tử liên thông là một toán tử đóng, nghĩa là tất cả các đồ thị liên thông đều có thể được tạo ra từ việc áp dụng nó trên các đồ thị đơn giản.
- Liên tục đoạn: Toán tử liên thông là một toán tử liên tục đoạn, nghĩa là nếu chúng ta thay đổi một cách nhỏ đồ thị (chẳng hạn như thêm hoặc xóa một cạnh), thì tập hợp các thành phần liên thông sẽ chỉ thay đổi một cách đáng kể.

Trong thư viện OpenCV, chúng ta có thể sử dụng toán tử liên thông để phân tích các thành phần liên thông trên một hình ảnh nhị phân (binary image). Thư viện OpenCV cung cấp hàm *connectedComponents* để tính toán các thành phần liên thông và trả về số lượng thành phần

và một ma trận kết quả, trong đó mỗi phần tử của ma trận biểu diễn một điểm trên hình ảnh và chứa một số nguyên dương đại diện cho nhãn của thành phần liên thông tương ứng.

```
binary_img, path = Read_Img()
#exe = Choose_Kernel()

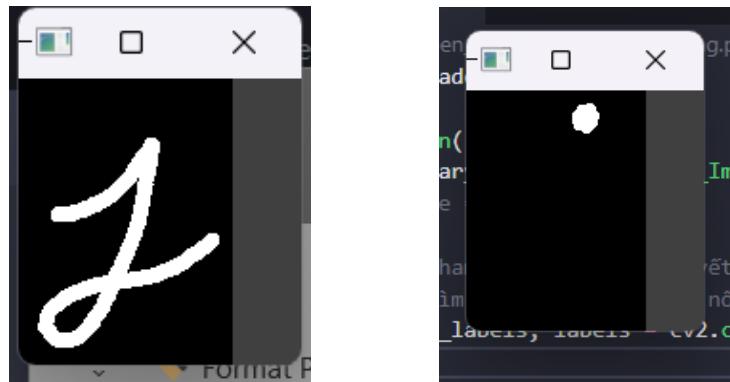
# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# tìm các đối tượng kết nối bằng cách sử dụng toán tử liên thông
num_labels, labels = cv2.connectedComponents(binary_img)

# Hiển thị ảnh kết quả
plt.subplot(1,1,1), plt.title("Original"),plt.imshow(binary_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()

# hiển thị ảnh kết quả cho toán tử liên thông
if path != "quote.png" and path != "noise.jpg" and path != "test.jpeg" and path != "input3.jpg":      #vì ảnh có n
    for i in range(1, num_labels):
        component = np.uint8(labels == i) * 255      #làm sáng các thành phần được chọn
        cv2.imshow('Component ' + str(i), component)    #thể hiện từng thành phần liên thông

    cv2.waitKey(0) #bổ trợ cho hàm cv2.imshow
    cv2.destroyAllWindows()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng hình ảnh *input.png*)



Nếu không sử dụng thư viện OpenCV, chúng ta có thể tự viết mã để tính toán các thành phần liên thông trên một hình ảnh nhị phân. Các bước chính để tính toán toán tử liên thông bao gồm:

- Xác định các đỉnh và cạnh trong đồ thị liên quan đến các điểm trên hình ảnh. Để làm điều này, chúng ta có thể duyệt qua các điểm trên hình ảnh và tìm các điểm liền kề (hoặc 8 điểm liền kề) có giá trị bằng 1.
- Tạo một danh sách các thành phần liên thông ban đầu rỗng.

- Duyệt qua tất cả các điểm trên hình ảnh. Nếu một điểm có giá trị bằng 1 và chưa được gán cho bất kỳ thành phần liên thông nào, thì tạo một thành phần liên thông mới và gán cho điểm đó và tất cả các điểm liền kề có giá trị bằng 1.
- Tiếp tục duyệt qua các điểm trên hình ảnh và gán các điểm có giá trị bằng 1 cho thành phần liên thông tương ứng. Nếu các điểm này đã được gán cho một thành phần liên thông khác, thì ta kết hợp các thành phần liên thông này thành một thành phần liên thông duy nhất.
- Lặp lại bước 4 cho tất cả các điểm trên hình ảnh.
- Trả về danh sách các thành phần liên thông.

```
def connected_components(image):
    # Tạo một ma trận lưu trữ kết quả
    labels = np.zeros_like(image)
    label = 1

    # Duyệt qua từng pixel của hình ảnh
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Nếu pixel đã được gán nhãn, bỏ qua
            if image[i][j] == 0 or labels[i][j] != 0:
                continue

            # Khởi tạo danh sách liên thông ban đầu
            stack = [(i, j)]

            # Lặp qua danh sách liên thông
            while stack:
                # Lấy một pixel từ danh sách liên thông
                pixel = stack.pop()
                x, y = pixel

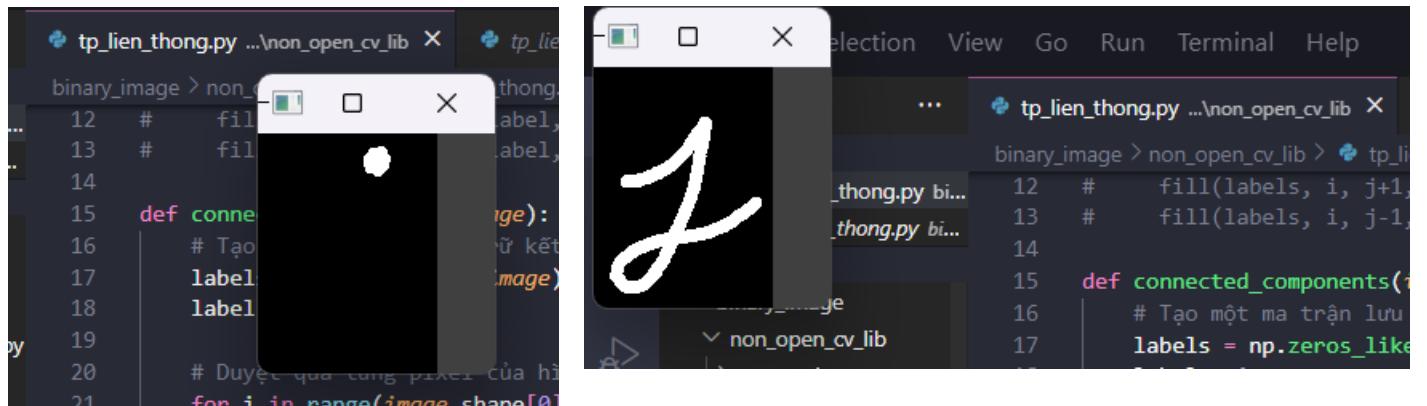
                # Gán nhãn cho pixel
                labels[x][y] = label

                # Kiểm tra các pixel lân cận
                neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
                for neighbor in neighbors:
                    nx, ny = neighbor
                    # Kiểm tra xem pixel lân cận có nằm trong hình ảnh và chưa được gán nhãn
                    if 0 <= nx < image.shape[0] and 0 <= ny < image.shape[1] and image[nx][ny] != 0 and labels[nx][ny] == 0:
                        # Thêm pixel vào danh sách liên thông
                        stack.append((nx, ny))

                # Tăng nhãn lên 1 để gán nhãn cho liên thông tiếp theo
                label += 1

    # Đếm số lượng thành phần liên thông và trả về kết quả
    num_labels = label
    return labels, num_labels
```

Kết quả khi không sử dụng hàm thư viện (sử dụng hình ảnh *input.png*)



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Connected Components ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện.

## 9. Toán tử Region Filling

Toán tử Region Filling là một trong những phương pháp xử lý hình ảnh được sử dụng để tạo ra các vùng mới trên hình ảnh dựa trên một vùng đã cho. Nó được sử dụng để xác định tất cả các điểm trong vùng đã cho mà có thể được kết nối với nhau thông qua các điểm lân cận để tạo ra một vùng mới.

$$X_0 = p \quad (\text{inside boundary})$$

$$X_k = (X_{k-1} \oplus B) \cap A^c, k = 1, 2, 3, \dots$$

Cách thức hoạt động của toán tử Region Filling bao gồm các bước sau:

- Chọn một điểm bất kỳ trong vùng đã cho làm điểm khởi đầu.
- Kiểm tra các điểm lân cận với điểm khởi đầu để xác định các điểm tiếp theo có thể được thêm vào vùng mới.
- Thêm các điểm mới này vào vùng mới và tiếp tục kiểm tra các điểm lân cận với các điểm mới này để tìm các điểm mới khác có thể được thêm vào vùng.
- Lặp lại bước 3 cho đến khi không còn điểm nào được thêm vào vùng mới.

Các tính chất của toán tử Region Filling là:

- **Tính chất lân cận:** Toán tử Region Filling hoạt động dựa trên các điểm lân cận với điểm khởi đầu. Các điểm này được xác định bởi một số tiêu chuẩn như khoảng cách, hướng, hình dạng, màu sắc,...
- **Tính chất kết nối:** Toán tử Region Filling chỉ kết nối các điểm nằm gần nhau và cùng thuộc một vùng để tạo thành một vùng mới. Việc kết nối các vùng có thể được thực hiện thông qua các điểm nằm chung hoặc dựa trên một số tiêu chuẩn kết nối.
- **Tính chất đa dạng:** Toán tử Region Filling có thể được sử dụng trên nhiều loại hình ảnh và cho nhiều mục đích khác nhau như phân đoạn ảnh, loại bỏ nhiễu, phát hiện vùng chứa đối tượng,...
- **Tính chất độ phức tạp:** Thời gian thực hiện toán tử Region Filling phụ thuộc vào kích thước và độ phức tạp của vùng cần được xử lý. Việc tìm kiếm các điểm lân cận và xác định kết nối giữa các điểm có thể tốn nhiều thời gian và tài nguyên tính toán.
- **Tính chất độc lập:** Toán tử Region Filling có thể được thực hiện độc lập trên từng vùng của hình ảnh mà không ảnh hưởng đến các vùng khác.

Trong thư viện OpenCV, để thực hiện toán tử Region Filling, chúng ta có thể sử dụng hàm cv2.floodFill(). Hàm này cho phép chúng ta tìm và điền màu vào các vùng liên thông trên hình ảnh dựa trên một điểm khởi đầu.

Cú pháp của hàm cv2.floodFill() như sau:

`cv2.floodFill(image, mask, seedPoint, newVal, loDiff=None, upDiff=None, flags=None)`

Trong đó:

**image:** hình ảnh đầu vào (phải là hình ảnh grayscale hoặc ảnh RGB)

**mask:** một mảng numpy có cùng kích thước với hình ảnh, được sử dụng để đánh dấu các điểm đã được thăm

**seedPoint:** điểm khởi đầu để tìm kiếm các điểm liên thông

**newVal:** giá trị mới được sử dụng để điền vào vùng được tìm thấy

**loDiff** và **upDiff:** ngưỡng khác biệt giữa giá trị pixel cần điền vào và giá trị pixel hiện tại của điểm liên thông (**loDiff** được sử dụng để xác định giới hạn dưới và **upDiff** được sử dụng để xác định giới hạn trên). Nếu không có giá trị được cung cấp, các giá trị mặc định sẽ được sử dụng (**loDiff** = **upDiff** = 0).

**flags:** cờ để chỉ định các tham số khác (vd: Kết hợp giữa 4 hoặc 8 điểm xung quanh)

```

binary_img, _ = Read_Img()
#exe = Choose_Kernel()

#thực hiện toán tử Region Filling
# Threshold.
# Set values equal to or above 220 to 0.
# Set values below 220 to 255.
th_, img_in = cv2.threshold(binary_img, 220, 255, cv2.THRESH_BINARY_INV)

# Copy the thresholded image.
im_floodfill = img_in.copy()

# Mask used to flood filling.
# Notice the size needs to be 2 pixels than the image.
h, w = img_in.shape[:2]
mask = np.zeros((h+2, w+2), np.uint8)

# Floodfill from point (0, 0)
cv2.floodFill(im_floodfill, mask, (0,0), 255);

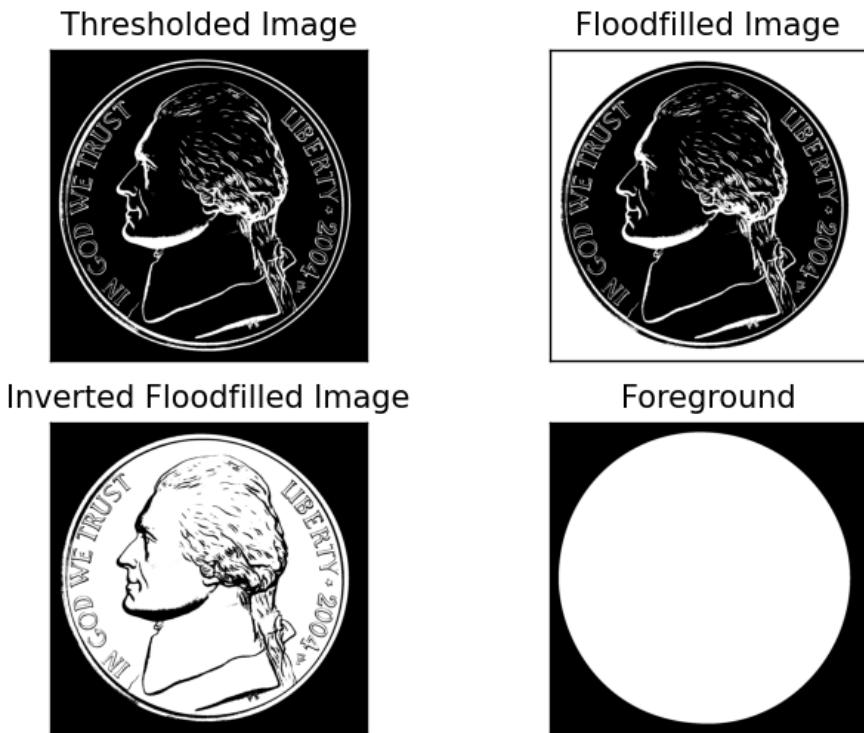
# Invert floodfilled image
im_floodfill_inv = cv2.bitwise_not(im_floodfill)

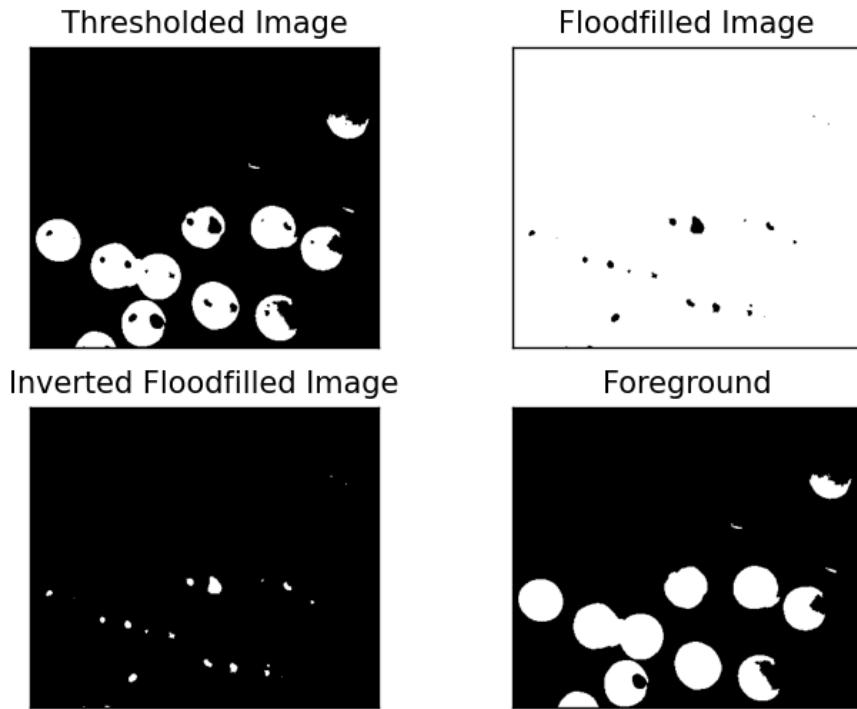
# Combine the two images to get the foreground.
im_out = img_in | im_floodfill_inv

# Hiển thị ảnh kết quả

```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input2.png*, *input3.jpg*)





Nếu không sử dụng thư viện OpenCV, bạn có thể triển khai toán tử Region Filling bằng cách sử dụng các thủ tục xử lý hình ảnh cơ bản. Dưới đây là một số bước thực hiện toán tử Region Filling:

- Đọc ảnh đầu vào và chuyển đổi nó sang ảnh xám nếu cần thiết.
- Chọn một điểm khởi đầu trên hình ảnh.
- Thiết lập ngưỡng cho phép điền vào các vùng, ví dụ như ngưỡng màu sắc hoặc ngưỡng độ xám.
- Tìm các điểm lân cận của điểm khởi đầu và kiểm tra xem chúng có thể được thêm vào vùng cần điền không.
- Nếu điểm lân cận có màu hoặc độ xám thích hợp, thêm nó vào vùng cần điền.
- Lặp lại các bước 4 và 5 cho tất cả các điểm lân cận của tất cả các điểm mới được thêm vào vùng cần điền, cho đến khi không còn điểm mới được thêm vào.
- Lưu lại hình ảnh sau khi đã điền vùng.

Tuy nhiên, cách triển khai này có thể không hiệu quả bằng việc sử dụng thư viện OpenCV, vì OpenCV cung cấp nhiều hàm xử lý hình ảnh hiệu quả hơn và được tối ưu hóa để xử lý các hình ảnh lớn và phức tạp.

```

def flood_fill(image, start_pixel, fill_color):
    """
    Fill the region starting from start_pixel with fill_color
    """
    # create a copy of the input image
    filled_image = image.copy()

    # get the dimensions of the input image
    height, width = filled_image.shape[:2]

    # define the directions we will search around the start_pixel
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    # create a queue to keep track of pixels to be filled
    queue = [start_pixel]

    # get the color of the start_pixel
    start_color = filled_image[start_pixel[0], start_pixel[1]]

    # loop until the queue is empty
    while queue:
        # get the next pixel to fill
        x, y = queue.pop(0)

        # if the pixel is outside the image, skip it
        if x < 0 or x >= height or y < 0 or y >= width:

            # if the pixel is outside the image, skip it
            if x < 0 or x >= height or y < 0 or y >= width:
                continue

            # if the pixel has already been filled or does not match the start color, skip it
            if filled_image[x, y] == fill_color or filled_image[x, y] != start_color:
                continue

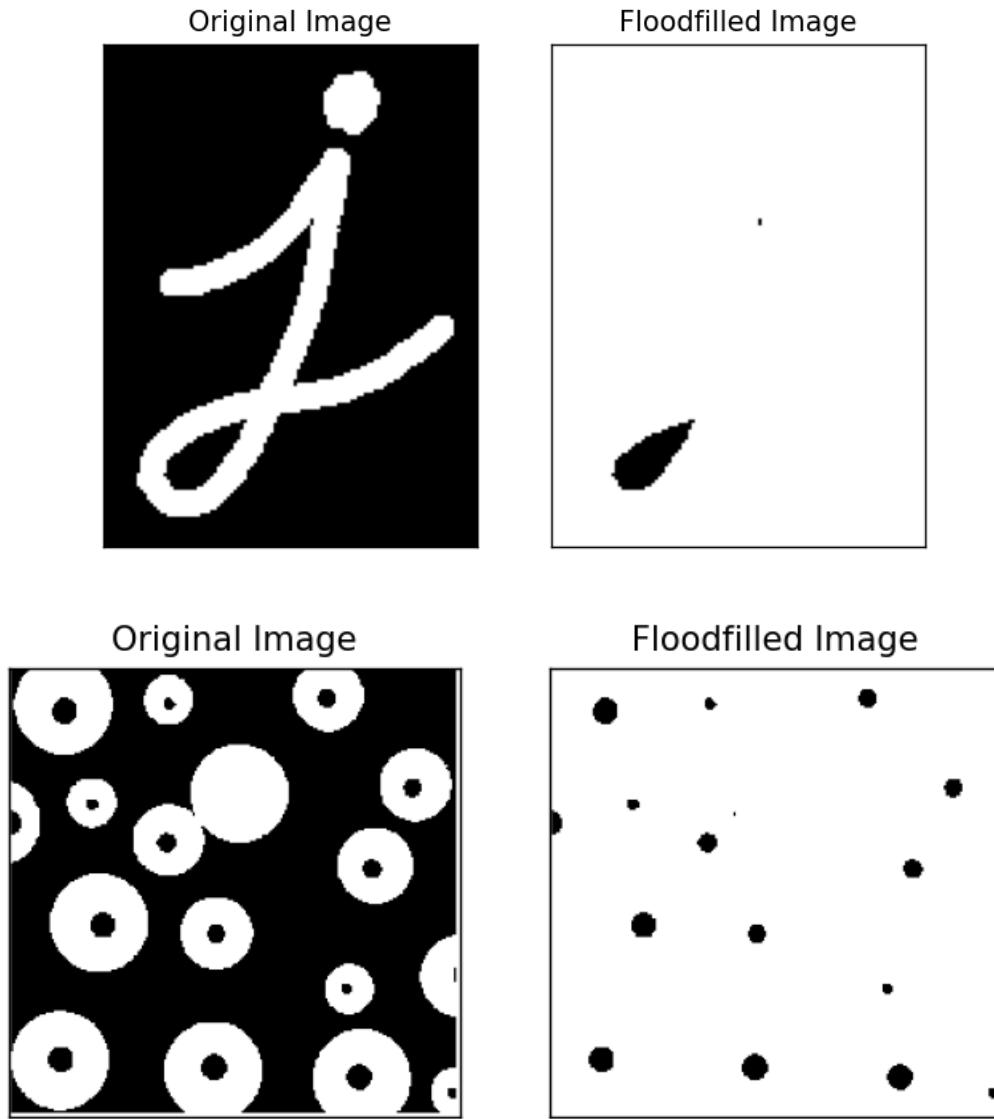
            # fill the pixel with the fill color
            filled_image[x, y] = fill_color

            # add the neighboring pixels to the queue to be filled
            for dx, dy in directions:
                queue.append((x + dx, y + dy))

    # return the filled image
    return filled_image

```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*)



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử region filling ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

## 10. Toán tử Thinning

Toán tử Thinning là một kỹ thuật trong xử lý ảnh, được sử dụng để làm mịn và làm mảnh các đối tượng trên ảnh. Toán tử này thường được sử dụng trong Morphological image processing, một kỹ thuật xử lý ảnh dựa trên hình thái học.

Thinning morphology là quá trình loại bỏ các điểm của đối tượng trong ảnh sao cho vẫn giữ được hình dạng tổng thể của đối tượng đó. Thinning thường được sử dụng để giảm kích thước của đối tượng và loại bỏ các chi tiết không cần thiết của ảnh.

Trong ảnh độ xám, toán tử Thinning được áp dụng cho các pixel đơn lẻ hoặc các nhóm pixel (gọi là structuring element) trên ảnh để xác định các điểm cần loại bỏ. Các điểm này thường được xác định bằng cách so sánh các giá trị của pixel trong structuring element với các giá trị pixel xung quanh. Nếu các giá trị này thỏa mãn một số điều kiện nhất định, pixel đó sẽ được loại bỏ.

Toán tử Thinning (hay Skeletonization) có các đặc điểm chính như sau:

- Loại bỏ các điểm dư thừa: Toán tử Thinning giúp loại bỏ các điểm dư thừa trong đối tượng trên ảnh, giúp giảm kích thước của đối tượng và giảm nhiễu trong ảnh.
- Giữ nguyên hình dạng và đặc trưng của đối tượng: Toán tử Thinning giữ nguyên hình dạng và các đặc trưng quan trọng của đối tượng trong ảnh, giúp giữ được tính chính xác của thông tin trên ảnh.
- Dễ dàng kết hợp với các toán tử khác: Toán tử Thinning có thể được kết hợp với các toán tử khác như toán tử Erosion, Dilation, Opening, Closing,... để tạo ra các kết quả phức tạp hơn và đạt được hiệu quả cao hơn trong việc xử lý ảnh.

Toán tử Thinning (hay còn gọi là Skeletonization) thường được biểu diễn bằng công thức:

$$B = \{b_1, b_2, \dots, b_n\}$$

Trong đó, B là tập hợp các structuring element (nhóm pixel) được sử dụng để xác định các điểm cần loại bỏ. Các structuring element này được chọn sao cho chúng phù hợp với kích thước và hình dạng của đối tượng trên ảnh.

Quá trình Thinning được thực hiện bằng cách so sánh các giá trị pixel trong structuring element với các giá trị pixel xung quanh để xác định các điểm cần loại bỏ. Nếu các giá trị này thỏa mãn một số điều kiện nhất định (ví dụ như giá trị pixel nằm trong structuring element phải là giá trị nhỏ nhất), pixel đó sẽ được loại bỏ.

Trong thư viện OpenCV, để thực hiện toán tử Thinning trên ảnh độ xám, bạn có thể sử dụng hàm "cv2.ximgproc.thinning" trong module "ximgproc". Dưới đây là các bước thực hiện:

- Đọc ảnh đầu vào bằng hàm "cv2.imread".
- Chuyển ảnh đầu vào sang ảnh độ xám bằng hàm "cv2.cvtColor".
- Áp dụng toán tử Thinning bằng hàm "cv2.ximgproc.thinning".

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *input.png*, *input1.png*, *input2.png*, *input3.jpg*, *quote.png*)

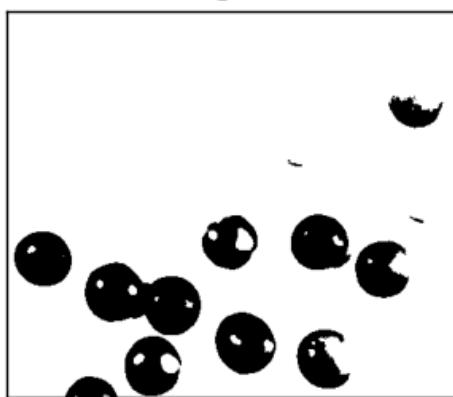
Original



Thinning



Original



Thinning

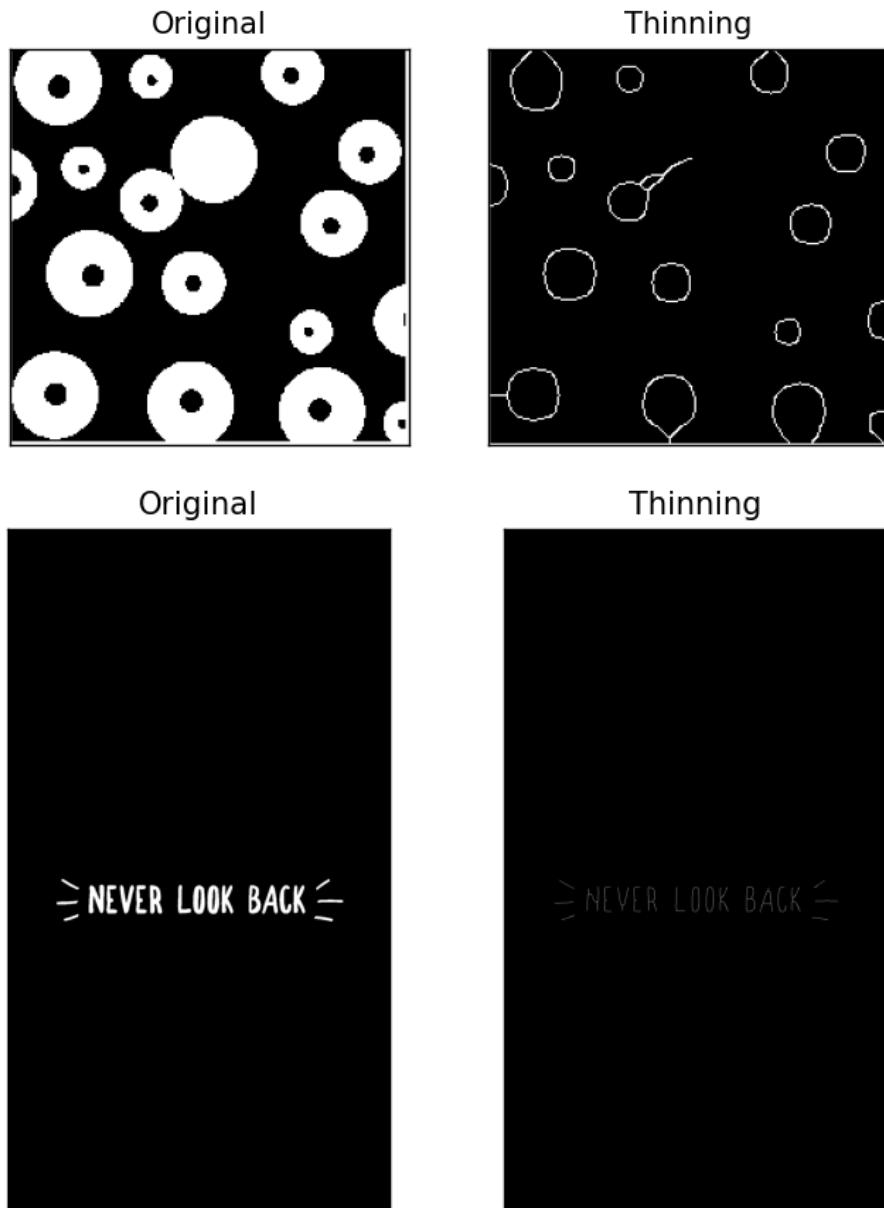


Original



Thinning





Để thực hiện thuật toán này mà không sử dụng thư viện OpenCV, bạn có thể làm theo các bước sau:

- Tạo ma trận skeleton: Ma trận này sẽ được sử dụng để lưu trữ các điểm trên biên của hình ảnh sau khi áp dụng thuật toán thinning morphology. Kích thước của ma trận skeleton phải bằng với kích thước của hình ảnh đầu vào. Ban đầu, giá trị của tất cả các điểm trong ma trận này được gán là 0.
- Chuyển các điểm trên hình ảnh đầu vào có giá trị khác 0 thành 1 và gán vào ma trận skeleton.

- Đảo ngược giá trị các điểm trong ma trận skeleton, sao cho những điểm ban đầu có giá trị khác 0 sẽ được đặt thành 0 và ngược lại. Mục đích của bước này là để thuật toán thinning morphology có thể loại bỏ các điểm trên biên của hình ảnh theo đúng quy trình.
- Áp dụng thuật toán thinning morphology trên ma trận skeleton theo vòng lặp chính của thuật toán:
  - Bước 1: Loại bỏ các điểm trên biên của hình ảnh theo quy tắc được xác định trước (thường dùng quy tắc Zhang-Suen).
  - Bước 2: Loại bỏ các điểm trên biên của hình ảnh theo quy tắc khác được xác định trước (thường dùng quy tắc Zhang-Suen).
- Lặp lại bước 5 cho đến khi không còn có điểm nào được loại bỏ. Kết quả cuối cùng của thuật toán thinning morphology là ma trận skeleton sau khi loại bỏ tất cả các điểm trên biên không còn cần thiết của hình ảnh.

```

def thinning(img):
    """
    Thinning morphology using Zhang-Suen algorithm.
    """

    skeleton = np.zeros(img.shape, np.uint8)
    skeleton[img > 0] = 1 # Lấy ảnh nền đen và chuyển thành ảnh trắng
    skeleton = 1 - skeleton # Đảo ngược ảnh để tiện thao tác

    while True:
        to_delete = []

        # Step 1: Loại bỏ các điểm không cần thiết
        for i in range(1, skeleton.shape[0] - 1):
            for j in range(1, skeleton.shape[1] - 1):
                if skeleton[i][j] == 0:
                    continue

                # Tính số lượng điểm lân cận
                neighbors = [skeleton[i-1][j], skeleton[i-1][j+1], skeleton[i][j+1],
                             skeleton[i+1][j+1], skeleton[i+1][j], skeleton[i+1][j-1],
                             skeleton[i][j-1], skeleton[i-1][j-1]]

                # Kiểm tra số lượng điểm lân cận có thỏa mãn yêu cầu không
                if sum(neighbors) < 2 or sum(neighbors) > 6:
                    continue

                # Kiểm tra số lượng điểm lân cận thay đổi từ 0 sang 1 có chính xác 1 lần không
                trans = 0
                for k in range(len(neighbors) - 1):
                    if neighbors[k] == 0 and neighbors[k+1] == 1:
                        trans += 1
                if neighbors[-1] == 0 and neighbors[0] == 1:
                    trans += 1

                if trans != 1:
                    continue

                to_delete.append((i, j))

        # Xóa các điểm không cần thiết
        for i, j in to_delete:

```

```

# Xóa các điểm không cần thiết
for i, j in to_delete:
    skeleton[i][j] = 0

# Step 2: Loại bỏ các điểm không cần thiết
to_delete = []
for i in range(1, skeleton.shape[0] - 1):
    for j in range(1, skeleton.shape[1] - 1):
        if skeleton[i][j] == 0:
            continue

        # Tính số lượng điểm lân cận
        neighbors = [skeleton[i-1][j], skeleton[i-1][j+1], skeleton[i][j+1],
                     skeleton[i+1][j+1], skeleton[i+1][j], skeleton[i+1][j-1],
                     skeleton[i][j-1], skeleton[i-1][j-1]]

        # Kiểm tra số lượng điểm lân cận có thỏa mãn yêu cầu không
        if sum(neighbors) < 2 or sum(neighbors) > 6:
            continue

        # Kiểm tra số lượng điểm lân cận thay đổi từ 0 sang 1 có chính xác 1 lần không
        trans = 0
        for k in range(len(neighbors) - 1):
            if neighbors[k] == 0 and neighbors[k+1] == 1:
                trans += 1
        if neighbors[-1] == 0 and neighbors[0] == 1:
            trans += 1

        if trans != 1:
            continue

        to_delete.append((i, j))

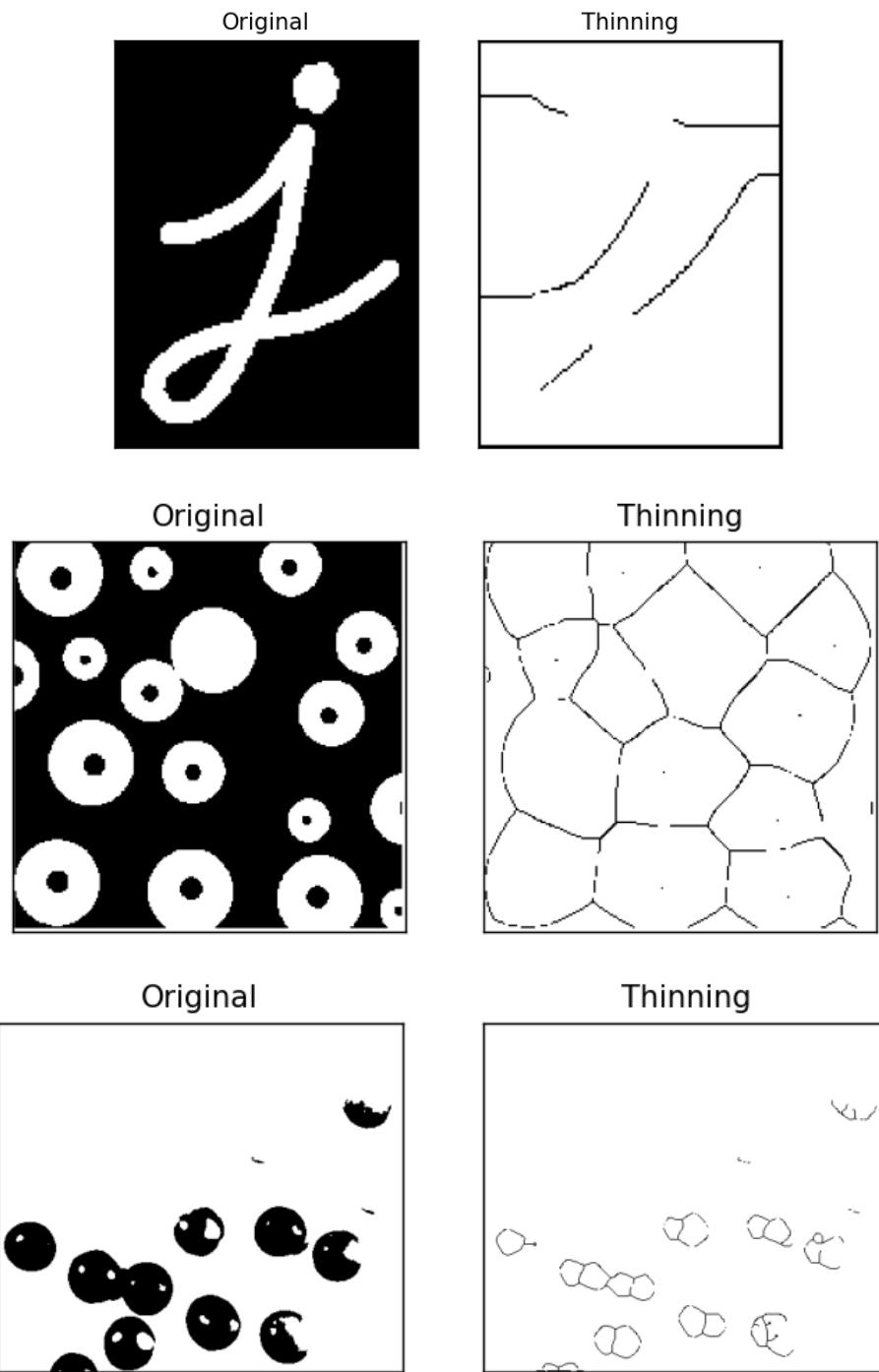
for i, j in to_delete:
    skeleton[i][j] = 0

if len(to_delete) == 0:
    break

skeleton = 1 - skeleton
return skeleton

```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *input.png*, *input1.png*, *input2.png*)



Nhìn chung, không có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Thinning khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện sẽ nhanh hơn hẳn.

## BÀI TẬP HAI THỰC HÀNH – GRAY IMAGE

### IV. Các toán tử Morphology được sử dụng trong bài thực hành hai

Morphology là một phương pháp xử lý hình ảnh được sử dụng để phân tích và xử lý các cấu trúc hình học trong hình ảnh như đường viền, tăng cường cấu trúc đối tượng, hình dạng và kích thước của các đối tượng. Toán tử morphology được sử dụng để thực hiện các thao tác như xóa bỏ nhiễu, tinh giản hình ảnh, phóng to hoặc thu nhỏ các đối tượng, tìm kiếm đường biên, phát hiện vật thể, ...

Các toán tử morphology được sử dụng rộng rãi trong xử lý hình ảnh để tạo ra các phép biến đổi và xử lý ảnh phức tạp như phát hiện khu vực, tách đối tượng, định lượng đối tượng, nâng cao chất lượng ảnh,... Dựa trên cơ sở phép toán đại số của các toán tử phi tuyến tác động trên hình dáng đối tượng (Algebra of non-linear operator), thay thế phép tích chập (Linear algebraic system of convolution) để thực hiện các phép biến đổi.

#### 1. Toán tử Dilation

Dilation (Dilate): Tăng kích thước của một vật thể bằng cách thêm các pixel vào vật thể để làm nó dày và rộng hơn. Mục đích chính là lấp kẽ hở, lỗ hổng của đối tượng.

Trong xử lý ảnh, toán tử dilation được sử dụng để mở rộng các đối tượng trong hình ảnh bằng cách thêm vào các điểm còn lại trong khu vực lân cận của đối tượng. Các điểm này được thêm vào để làm tăng kích thước của đối tượng và giúp đối tượng trở nên mượt mà hơn.

$$X \oplus B = \{p \in \varepsilon^2 : p = x + b, x \in X \text{ and } b \in B\}$$

$$X \oplus B = \{p \in \varepsilon^2 : (\hat{B})_p \cap X \neq \emptyset\}$$

$$X \oplus B = \bigcup_{b \in B} X_b$$

#### Tính chất

- Giao hoán:  $X \oplus B = B \oplus X$
- Kết hợp:  $X \oplus (B \oplus D) = (X \oplus B) \oplus D$
- **Hội tập tịnh tiến:**  $X \oplus B = \bigcup_{b \in B} X_b$
- Bất biến với phép tịnh tiến:  $X_h \oplus B = (X \oplus B)_h$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \oplus B \subseteq Y \oplus B$

Trong thư viện OpenCV, toán tử dilation được thực hiện bằng hàm cv2.dilate(). Hàm này có cú pháp như sau:

*cv2.dilate(src, dst, kernel, anchor, iterations, borderType, borderColor)*

Trong đó:

- src: là ảnh đầu vào (input image).
- dst: là ảnh đầu ra (output image).
- kernel: là ma trận (kernel) cỡ lớn được sử dụng cho toán tử dilation. Kernel này được tạo bằng hàm cv::getStructuringElement().
- anchor: là điểm trung tâm của kernel.
- iterations: là số lần lặp lại thực hiện toán tử dilation.
- borderType: là kiểu viền (border type) được sử dụng khi kernel trượt ra khỏi ảnh đầu vào. Có thể sử dụng các giá trị như cv::BORDER\_CONSTANT, cv::BORDER\_REPLICATE, cv::BORDER\_REFLECT,...
- borderColor: là giá trị được sử dụng khi kiểu viền là cv::BORDER\_CONSTANT.

```
img = Read_Img()
exe = Choose_Kernel()

# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử dilation
dilation = cv2.dilate(img, Kernel(exe), iterations = 1) #iterations = 1 nghĩa là bào mòn 1 lần

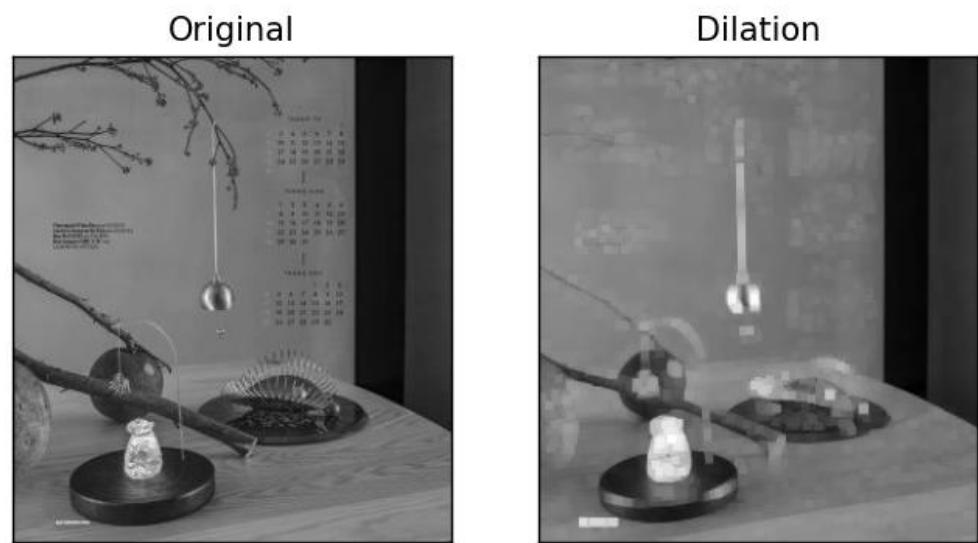
# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Dilation"), plt.imshow(dilation, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



Dùng mặt nạ kernel



Để thực hiện toán tử dilation trên một ảnh mà không sử dụng thư viện OpenCV, chúng ta cần thực hiện các bước sau:

- Định nghĩa kernel: Ta cần định nghĩa kernel (ma trận) có kích thước và hình dáng tùy ý. Kernel này có thể có các giá trị nhị phân, với giá trị bằng 1 tại các vị trí cần được thực hiện dilation, và giá trị bằng 0 tại các vị trí khác.
- Duyệt ảnh đầu vào: Ta duyệt qua từng điểm ảnh của ảnh đầu vào. Với mỗi điểm ảnh, ta lấy phần của ảnh tương ứng với kernel và thực hiện phép tính AND giữa kernel và phần ảnh này.
- Tính toán giá trị điểm ảnh đầu ra: Giá trị điểm ảnh đầu ra được tính bằng giá trị lớn nhất trong số các giá trị được tính toán ở bước 2.
- Lưu kết quả: Giá trị điểm ảnh tính được ở bước 3 được lưu vào vị trí tương ứng trong ảnh đầu ra.

Sau khi thực hiện các bước trên cho toàn bộ điểm ảnh của ảnh đầu vào, ta sẽ thu được ảnh đầu ra chứa đối tượng đã được mở rộng kích thước theo kernel khi thực hiện toán tử dilation.

```
# thực hiện phép tích chập 2D với ma trận kernel và ảnh đầu vào
def convolution_dilation(ker_mat, img):
    h, w = img.shape
    out = np.zeros((h, w), np.uint8)
    ksize = ker_mat.shape[0]
    pad_size = ksize // 2
    img_pad = np.pad(img, (pad_size, pad_size), mode='edge')
    for i in range(h):
        for j in range(w):
            out[i, j] = np.max(img_pad[i:i+ksize, j:j+ksize] * ker_mat)
    return out
```

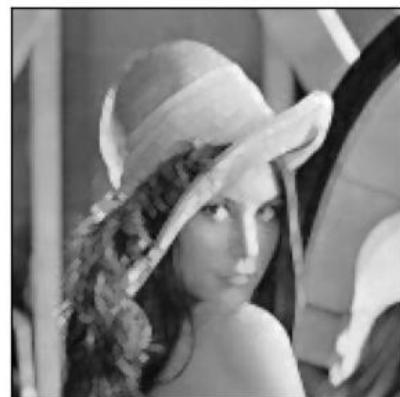
Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.png*, *inp.png*, *decor.jpg*)

Dùng mặt nạ kernel\_1

Original



Dilation



Original



Dilation



Dùng mặt nạ kernel\_4

Original



Dilation



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử dilation ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện theo như ghi nhận hai bên vẫn đang bắt kịp nhau.

## 2. Toán tử Erosion

Erosion (Erode): Loại bỏ các pixel ngoại vi của một vật thể để thu nhỏ nó lại. Mục đích chính là loại bỏ chi tiết không thích hợp (theo nghĩa về kích thước).

Trong xử lý ảnh, toán tử erosion được sử dụng để co lại kích thước của các đối tượng trong hình ảnh bằng cách loại bỏ các điểm nằm ở lân cận của đối tượng. Các điểm này được loại bỏ để giảm kích thước của đối tượng và giúp đối tượng trở nên sắc nét hơn.

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

### Tính chất

- Chống mở rộng:  $(0,0) \in B \Rightarrow X \Theta B \subseteq X$
- Không giao hoán:  $X \Theta B \neq B \Theta X$
- **Giao tập tịnh tiến ngược:**  $X \Theta B = \bigcap_{b \in B} X_{-b}$
- Bất biến với phép tịnh tiến:  $X_h \Theta B = (X \Theta B)_h$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \Theta B \subseteq Y \Theta B$

Trong thư viện OpenCV, chúng ta có thể sử dụng hàm erode() để thực hiện toán tử erosion trên ảnh. Hàm này có các tham số đầu vào như sau:

```
cv2.erode(src, kernel, dst=None, anchor=None, iterations=None, borderType=None,
borderValue=None)
```

Trong đó:

- src: ảnh đầu vào cần được thực hiện toán tử erosion.

- kernel: ma trận kernel có kích thước và hình dáng tùy ý, được sử dụng để thực hiện toán tử erosion.
- dst: ảnh đầu ra sau khi thực hiện toán tử erosion. Nếu không được cung cấp, hàm sẽ tạo ra ảnh đầu ra mới với cùng kích thước và kiểu dữ liệu với ảnh đầu vào.
- anchor: điểm neo, được sử dụng để xác định vị trí của kernel trong quá trình tính toán. Mặc định là (-1, -1), nghĩa là điểm neo ở trung tâm của kernel.
- iterations: số lần lặp lại toán tử erosion. Mặc định là 1.
- borderType: cách xử lý viền ảnh, có các giá trị như cv2.BORDER\_CONSTANT, cv2.BORDER\_REPLICATE, cv2.BORDER\_REFLECT, cv2.BORDER\_WRAP, v.v.
- borderColor: giá trị được sử dụng khi borderType được chọn là cv2.BORDER\_CONSTANT.

```



```

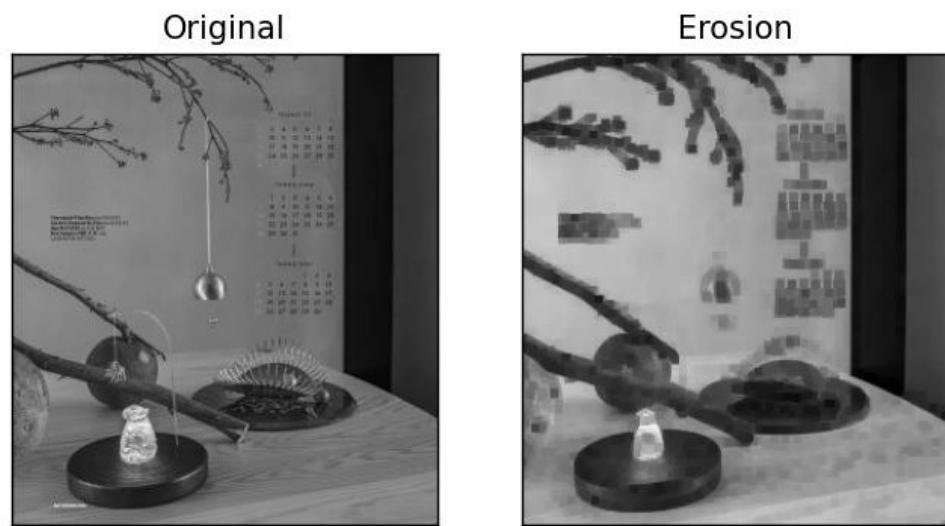
Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker





Dùng mặt nạ kernel



Khi không sử dụng thư viện OpenCV, chúng ta có thể thực hiện toán tử erosion bằng cách sử dụng các phép toán ma trận đơn giản như tích chập hoặc xử lý ma trận. Cụ thể, để thực hiện toán tử erosion trên ảnh, chúng ta có thể làm như sau:

- Đọc ảnh đầu vào và chuyển đổi thành một ma trận NumPy.
- Định nghĩa ma trận kernel, có kích thước và hình dáng tùy ý.
- Duyệt qua từng điểm ảnh của ảnh đầu vào.
- Tại mỗi điểm ảnh, tạo ra một vùng lân cận bằng cách lấy ma trận con trong ảnh đầu vào, có kích thước bằng kích thước của kernel và tâm là điểm đó.
- Tính toán giá trị tối thiểu trong vùng lân cận đó.
- Gán giá trị tối thiểu này vào điểm ảnh tương ứng trong ảnh đầu ra.
- Lặp lại quá trình từ bước 3 đến bước 6 cho tất cả các điểm ảnh trong ảnh đầu vào.

- Hiển thị ảnh đầu vào và ảnh đầu ra sau khi thực hiện toán tử erosion.

```

def erosion(img, kernel):
    # Lấy kích thước ảnh và kernel
    rows, cols = img.shape
    k_rows, k_cols = kernel.shape

    # Khởi tạo ảnh kết quả
    result = np.zeros((rows, cols))

    # Duyệt qua từng pixel của ảnh đầu vào
    for i in range(rows):
        for j in range(cols):
            # Tìm giá trị nhỏ nhất trong vùng lân cận
            min_val = 255
            for m in range(k_rows):
                for n in range(k_cols):
                    if kernel[m, n] != 0:
                        x = i + m - k_rows // 2
                        y = j + n - k_cols // 2
                        if x >= 0 and x < rows and y >= 0 and y < cols:
                            if img[x, y] < min_val:
                                min_val = img[x, y]
            result[i, j] = min_val

    return result.astype(np.uint8)

```

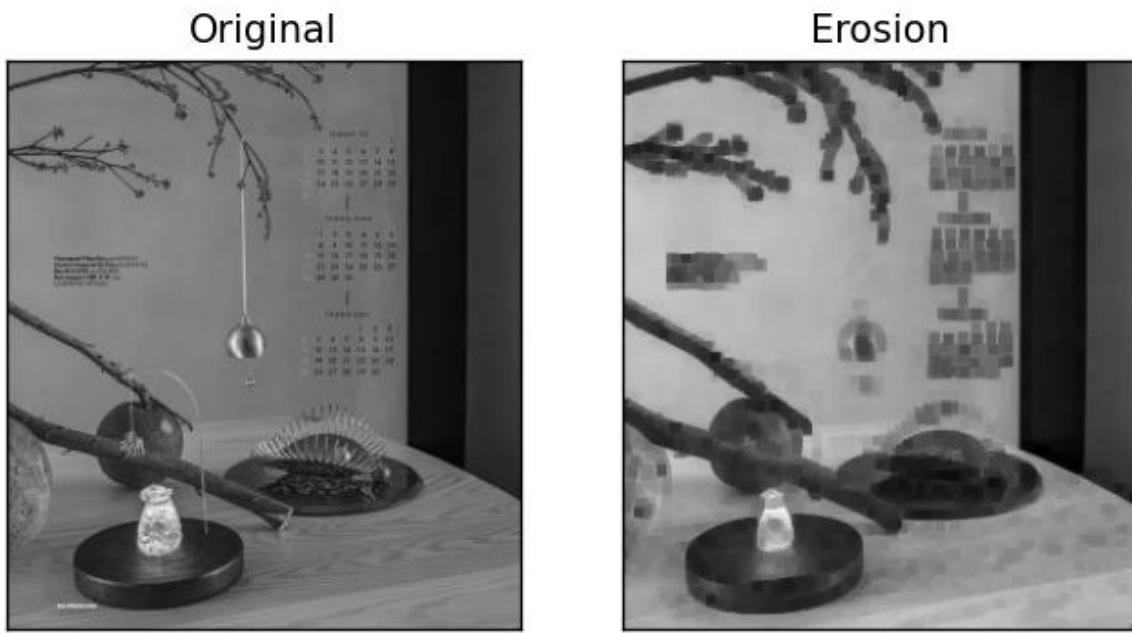
Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử erosion ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn.

### 3. Toán tử Opening

Opening: Thực hiện erosion trước rồi dilation để loại bỏ các chi tiết không mong muốn như những điểm nhiễu và kẽ đường nhỏ trong vật thể. Mục đích chính là làm trơn biên đối tượng, loại eo hẹp và chỗ lồi mỏng.

Toán tử opening là kết hợp giữa toán tử erosion và toán tử dilation trên ảnh. Kỹ thuật opening được sử dụng để loại bỏ các chi tiết nhỏ và giảm thiểu nhiễu trên ảnh.

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

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

#### Tính chất

- Chỗng mở rộng:  $(0,0) \in B \Rightarrow X \circ B \subseteq X$
- Lũy đẳng:  $X \circ B = (X \circ B) \circ B$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \circ B \subseteq Y \circ B$

Trong thư viện OpenCV, chúng ta có thể sử dụng hàm `cv2.morphologyEx(scr, op, kernel)` để thực hiện toán tử opening trên ảnh. Hàm này nhận vào 4 đối số:

- src: ảnh đầu vào.
- op: loại toán tử morphological operation cần thực hiện. Ở đây, ta sử dụng giá trị `cv2.MORPH_OPEN` để thực hiện toán tử opening.
- kernel: ma trận kernel được sử dụng cho toán tử morphological operation. Đây là ma trận 2 chiều với các giá trị phần tử nằm trong khoảng [0, 1] hoặc [0, 255]. Kích thước của kernel phải lẻ để kernel có thể có một điểm trung tâm.

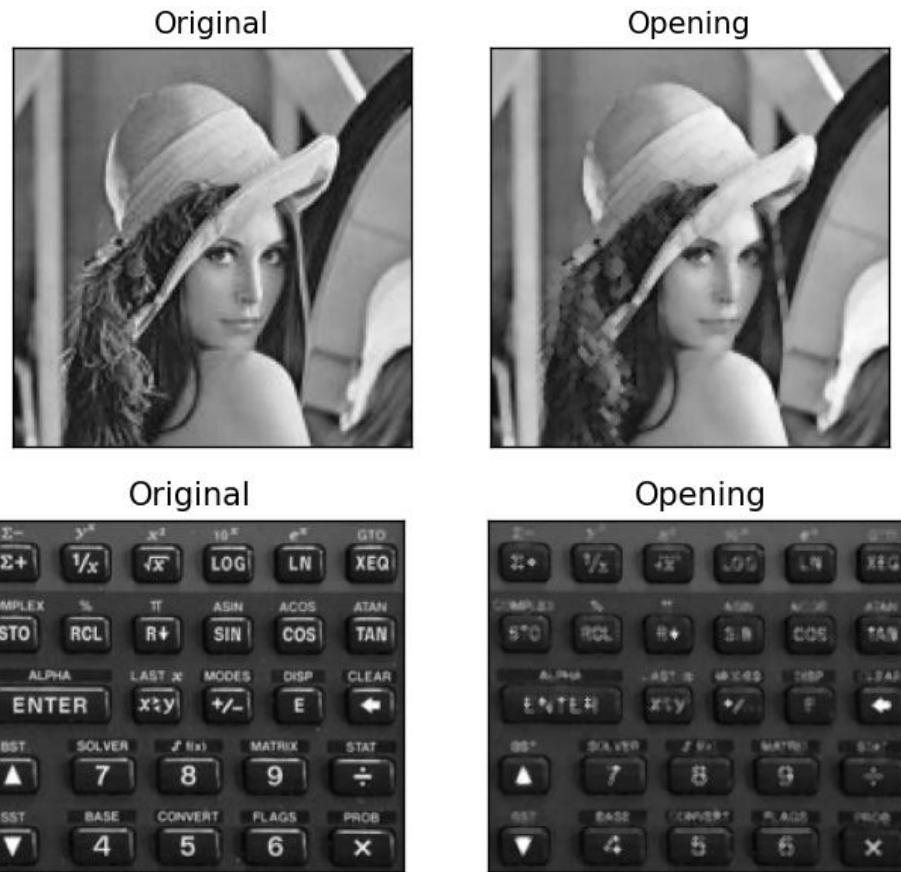
```
img = Read_Img()
exe = Choose_Kernel()

# Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
# Thực hiện toán tử opening
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, Kernel(exe))

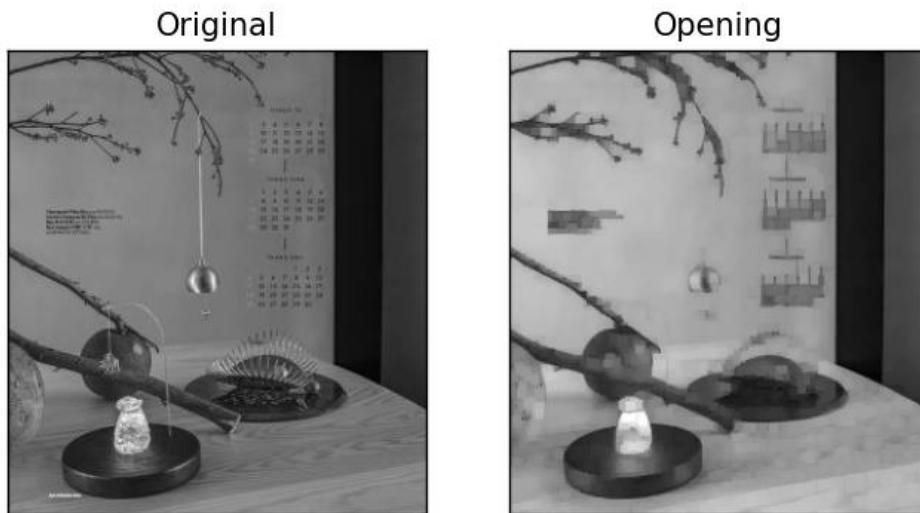
# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Opening"), plt.imshow(opening, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Cách thực hiện toán tử opening trên ảnh như sau:

- Đọc ảnh đầu vào và chuyển đổi thành một ma trận NumPy.
- Định nghĩa ma trận kernel, có kích thước và hình dáng tùy ý.
- Thực hiện toán tử erosion trên ảnh đầu vào bằng cách sử dụng kernel đã định nghĩa ở bước 2.
- Thực hiện toán tử dilation trên kết quả erosion ở bước 3 bằng cách sử dụng kernel đã định nghĩa ở bước 2.
- Hiển thị ảnh đầu vào và ảnh đầu ra sau khi thực hiện toán tử opening.

Kết quả của toán tử opening là ảnh đã được xử lý sao cho các đối tượng trên ảnh được giữ nguyên hình dạng và kích thước cơ bản của chúng, nhưng các đối tượng nhỏ hơn hoặc bị nhiễu đã được loại bỏ.

```
from header_gray_noncv import *
from dilation import convolution_dilation
from erosion import erosion

# Function to perform opening on image
def opening(image, kernel):
    #image_eroded = erosion(image, kernel)
    return convolution_dilation(kernel, np.array(erosion(image, kernel)))
```

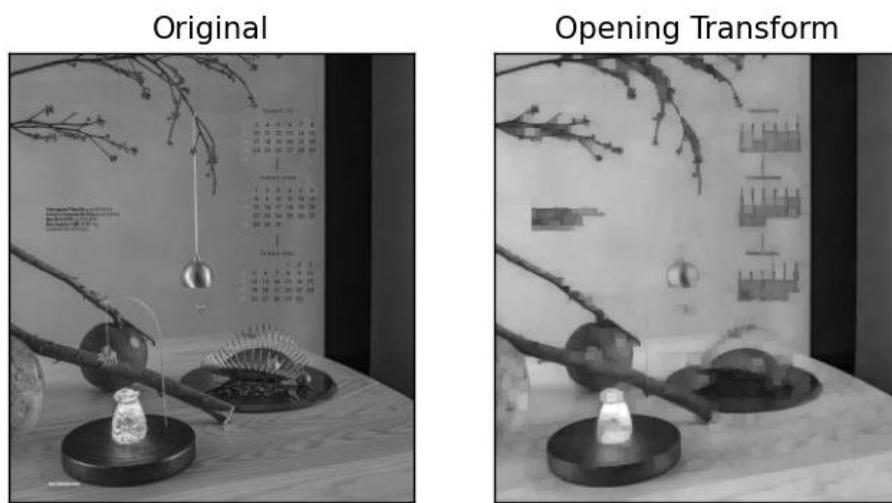
Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử opening ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

#### 4. Toán tử Closing

Closing: Thực hiện dilation trước rồi erosion để đóng các lỗ trống và nối các đường viền trong vật thể.

Trong xử lý ảnh, toán tử closing là một phép biến đổi hình thái học được sử dụng để loại bỏ các lỗ hổng (holes) nhỏ hoặc các đối tượng nhỏ không mong muốn khác trong các vùng liền kề.

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

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

### Tính chất

- Mở rộng:  $(0,0) \in B \Rightarrow X \subseteq X \bullet B$
- Lũy đẳng:  $X \bullet B = (X \bullet B) \bullet B$
- Bảo toàn phép bao hàm:  $X \subseteq Y \Rightarrow X \bullet B \subseteq Y \bullet B$

"Toán tử closing" trong OpenCV là một phép toán kết hợp giữa phép toán dilation (phóng to) và phép toán erosion (co lại). Phép toán dilation được sử dụng để mở rộng các đối tượng trắng (foreground) trong ảnh, trong khi phép toán erosion được sử dụng để thu nhỏ các đối tượng trắng. Khi kết hợp cả hai phép toán, toán tử closing sẽ giúp loại bỏ các lỗ hổng nhỏ và kết nối các phần của đối tượng mà bị phân tách trong ảnh.

`cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)`

Trong đó, img là ảnh đầu vào, kernel là kernel được sử dụng để thực hiện toán tử closing.

```
def main():
    img = Read_Img()
    exe = Choose_Kernel()

    # Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
    # Thực hiện toán tử closing
    closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, Kernel(exe))

    # Hiển thị ảnh kết quả
    plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(1,2,2), plt.title("Closing"), plt.imshow(closing, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.show()
```

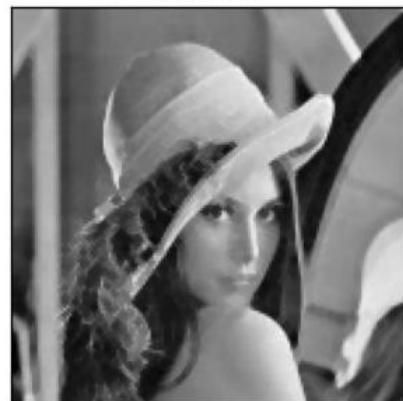
Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker

Original



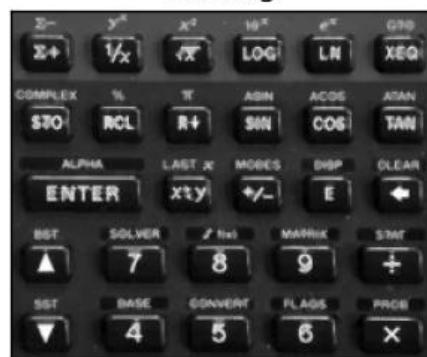
Closing



Original



Closing



Dùng mặt nạ kernel

Original



Closing



Để thực hiện toán tử closing trong xử lý ảnh khi không sử dụng thư viện OpenCV, bạn có thể thực hiện các bước sau:

- Xác định kích thước của một cửa sổ (window) và hình dạng của phần tử cấu thành cửa sổ này. Hình dạng phổ biến của phần tử là hình vuông hay hình tròn.
- Duyệt qua toàn bộ ảnh và thực hiện phép giãn nở (dilation) trên từng điểm ảnh của ảnh đầu vào bằng cách sử dụng phần tử đã xác định ở bước trước đó. Phép giãn nở tại một điểm ảnh sẽ gán giá trị cao nhất trong cửa sổ đang xét cho điểm ảnh đó.
- Thực hiện phép co ngắn (erosion) trên ảnh đã được giãn nở ở bước trên bằng cách sử dụng cùng phần tử đã xác định ở bước trên. Phép co ngắn tại một điểm ảnh sẽ gán giá trị thấp nhất trong cửa sổ đang xét cho điểm ảnh đó.

Kết quả sau khi thực hiện toán tử closing là ảnh đã được loại bỏ các lỗ hổng nhỏ trong vật thể được nhận dạng trong ảnh.

```
from header_gray_noncv import *
from dilation import convolution_dilation
from erosion import erosion

# Function to perform closing on image, given image, structuring element and origin
def closing(image, kernel):
    #image_dilated = convolution_dilation(kernel, image)
    #op_img = erosion(image_dilated, kernel)
    return erosion(np.array(convolution_dilation(kernel, image)), kernel)
```

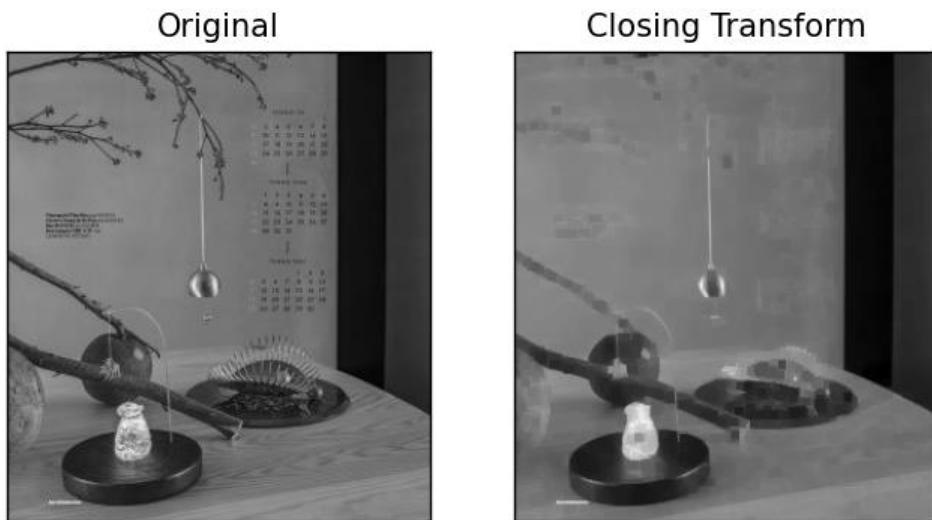
Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử closing ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## 5. Toán tử Gradient

Toán tử morphology gradient trong xử lý ảnh độ xám là một công cụ quan trọng để phát hiện và phân tích các đối tượng trong ảnh. Nó được sử dụng để tìm sự khác biệt giữa các pixel trong ảnh, để phát hiện biên của các đối tượng.

## Định nghĩa

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

Nó có một số tính chất quan trọng như sau:

- Độ nhạy cảm: Toán tử gradient morphology nhạy cảm với sự khác biệt giữa các pixel trong ảnh, do đó nó có thể phát hiện được các biên của các đối tượng với độ chính xác cao.
- Khả năng loại bỏ nhiễu: Toán tử gradient morphology có khả năng loại bỏ nhiễu và các chi tiết không cần thiết trong ảnh, đồng thời vẫn giữ lại các biên quan trọng của đối tượng.
- Tính cục bộ: Toán tử gradient morphology tính toán các biên của đối tượng dựa trên thông tin cục bộ, nghĩa là nó không phụ thuộc vào toàn bộ ảnh. Điều này giúp nó có thể phát hiện được các biên của các đối tượng có kích thước và hình dạng khác nhau.
- Đơn giản: Toán tử gradient morphology là một công cụ đơn giản và dễ sử dụng trong xử lý ảnh, và được tích hợp trong các thư viện xử lý ảnh phổ biến như OpenCV.

`cv2.morphologyEx(img, cv2.MORPH_GRADIENT, Kernel(ex))`

Hàm cv2.morphologyEx() được sử dụng để áp dụng các phép toán xử lý hình thái học lên ảnh đầu vào img. Tham số thứ hai cv2.MORPH\_GRADIENT cho biết phép toán cần thực hiện là morphology gradient.

Tham số thứ ba Kernel là một ma trận kernel được sử dụng để thực hiện phép toán morphology gradient.

```
def main():
    img = Read_Img()
    exe = Choose_Kernel()

    # Tham số iterations quyết định số lần quá trình bào mòn với lớp mặt nạ kernel
    # Thực hiện toán tử tìm biên
    boun_ext = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, Kernel(ex))

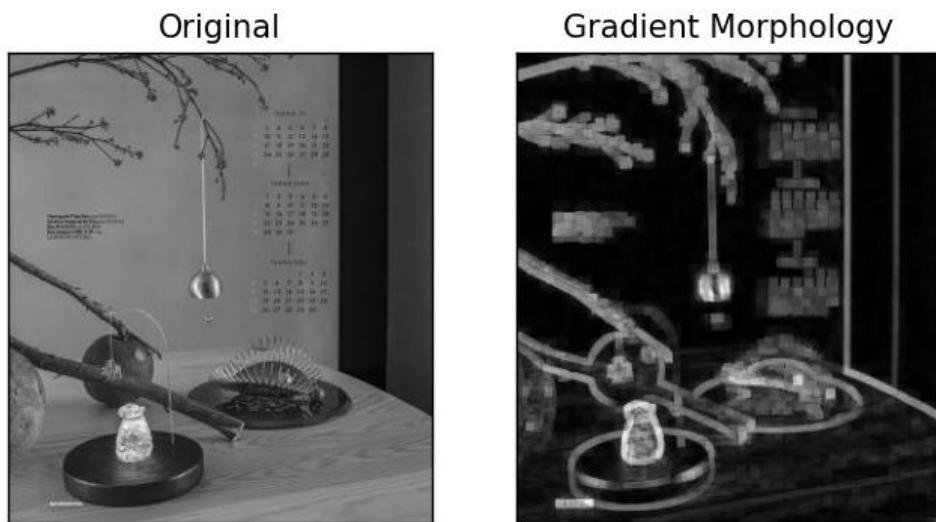
    # Hiển thị ảnh kết quả
    plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(1,2,2), plt.title("Gradient Morphology"), plt.imshow(boun_ext, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Toán tử morphology gradient là một phép toán xử lý hình thái học trên ảnh độ xám, được thực hiện bằng cách áp dụng phép toán trừ giữa hình ảnh được mở rộng và hình ảnh được co lại. Điều này cho phép ta xác định sự khác biệt giữa các pixel trong ảnh và xác định được các biên của các đối tượng.

Cụ thể, để tính toán morphology gradient, ta áp dụng phép toán dilation (mở rộng) và erosion (co lại) lên ảnh đầu vào. Sau đó, ta trừ ảnh được co lại từ ảnh được mở rộng. Kết quả là một ảnh mới, trong đó các điểm sáng tương ứng với các biên của các đối tượng trong ảnh ban đầu.

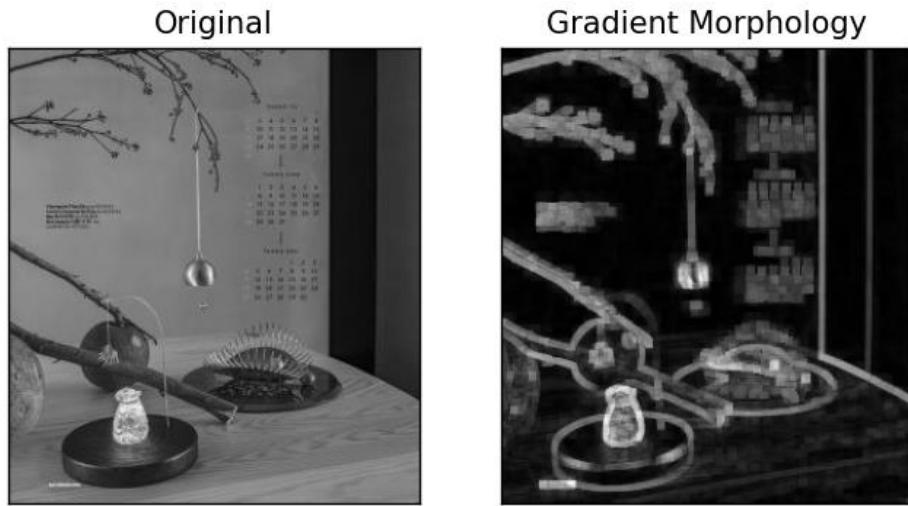
```
def gradient(image, kernel):
    return convolution_dilation(kernel, image) - erosion(image, kernel)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1



## Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Gradient ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

```
#hàm để tính tích chập
def convolve(img, kernel):

    H = (kernel.shape[0] - 1) // 2
    W = (kernel.shape[1] - 1) // 2

    #loại bỏ những giá trị 0 ở viền của ảnh kết quả
    #out = np.zeros((img.shape[0] - kernel.shape[0] + 1, img.shape[1] - kernel.shape[1] + 1))
    #hoặc để padding zero ở viền ảnh gốc để đảm bảo ảnh đầu ra không bị thu nhỏ
    out = np.zeros((img.shape[0], img.shape[1]))

    #Hai vòng lặp ngoài cùng biến i cho hàng, j cho cột, thay đổi để dịch chuyển ma trận mặt nạ kernel
    for i in range(H, img.shape[0] - H):
        for j in range(W, img.shape[1] - W):
            sum = 0
            #Hai vòng lặp k, l thực hiện phép dot product giữa ma trận cửa sổ với kernel
            for k in range(-H, H + 1):
                for l in range(-W, W + 1):
                    a = img[i - k, j - l]
                    w = kernel[H + k, W + l]
                    sum += (w * a)           #g(x, y) = f(x - i, y - j)*h(i, j)
            out[i, j] = sum
    return out

def gradient_morp(image, kernel):
    dx = convolve(image, np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]))
    dy = convolve([image, np.array([[1, -2, -1], [0, 0, 0], [1, 2, 1]])])
    gradient_magnitude = np.sqrt(np.square(dx) + np.square(dy))
    edges = convolve(gradient_magnitude, kernel)

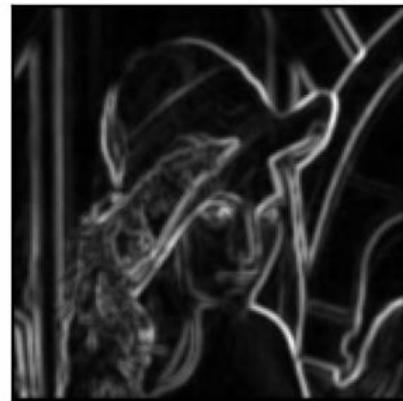
    return edges
```

Nếu sử dụng theo đoạn chương trình này, kết quả của toán tử gradient cho ra sẽ có thêm một độ mờ nhẹ.

Original



Gradient Morphology



Original



Gradient Morphology



Original



Gradient Morphology



## 6. Toán tử Granulometry

Toán tử morphology granulometry là một phương pháp trong xử lý ảnh để phân tích cấu trúc của các đối tượng trong ảnh độ xám. Phương pháp này được sử dụng để tính toán kích thước của các đối tượng trong ảnh bằng cách thay đổi kích thước của các phần tử cấu thành (như kernel hoặc structuring element) và tính toán sự thay đổi của đối tượng trong ảnh.

Đặc điểm của toán tử Granulometry bao gồm:

- Phù hợp với các ảnh chứa các đối tượng có kích thước khác nhau: Toán tử Granulometry có thể xử lý các ảnh chứa các đối tượng có kích thước khác nhau. Kỹ thuật này cung cấp thông tin về phân bố kích thước của các đối tượng trong ảnh.
- Không chỉ đơn thuần phân tích kích thước của đối tượng: Toán tử Granulometry không chỉ đơn thuần phân tích kích thước của đối tượng trong ảnh mà còn phân tích các tính chất khác như hình dạng và độ dày của các đối tượng.
- Dùng các bộ lọc khác nhau để phân tích kích thước: Toán tử Granulometry dùng các bộ lọc khác nhau để phân tích kích thước của các đối tượng trong ảnh. Các bộ lọc này thường được thiết kế dựa trên các hình dạng và kích thước của các đối tượng cần phân tích.
- Cho phép phân tích tầng và phân bố kích thước của các đối tượng: Toán tử Granulometry cho phép phân tích tầng và phân bố kích thước của các đối tượng trong ảnh. Các thông tin này có thể được sử dụng để đưa ra các quyết định trong nhiều lĩnh vực như y học, công nghệ sản xuất, hoặc khoa học địa chất.
- Áp dụng linh hoạt trên nhiều loại ảnh: Toán tử Granulometry có thể áp dụng linh hoạt trên nhiều loại ảnh như ảnh độ xám, ảnh nhị phân, ảnh màu, v.v. Nó cũng có thể được kết hợp với các kỹ thuật khác như lọc và biến đổi Fourier để tăng cường tính năng của nó trong xử lý ảnh.

Toán tử morphology granulometry tính toán tỉ lệ diện tích của các điểm ảnh được bao phủ bởi phần tử cấu thành trên tổng số điểm ảnh trong ảnh theo kích thước của phần tử cấu thành đó. Công thức chung cho toán tử morphology granulometry như sau:

$$\text{Granulometry}(k) = (B \bullet B_{-k}) - (B \circ B_{-k})$$

Trong đó:

- Granulometry(k) là tỉ lệ diện tích của các điểm ảnh được bao phủ bởi phần tử cấu thành k trên tổng số điểm ảnh trong ảnh.
- B là ảnh đầu vào.
- B<sub>-k</sub> là phần tử cấu thành có kích thước k.

- • là toán tử closing và • là toán tử opening trong morphology.

```
def main():
    img = Read_Img()
    exe = Choose_Kernel()

    # Thực hiện toán tử opening
    opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, Kernel(exe))

    # Thực hiện toán tử closing
    closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, Kernel(exe))

    # Tính toán tử Granulometry bằng cách trừ ảnh đóng và ảnh mở
    granulometry = closing - opening

    # Hiển thị ảnh kết quả
    plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(1,2,2), plt.title("Granulometry"), plt.imshow(granulometry, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker

Original



Granulometry



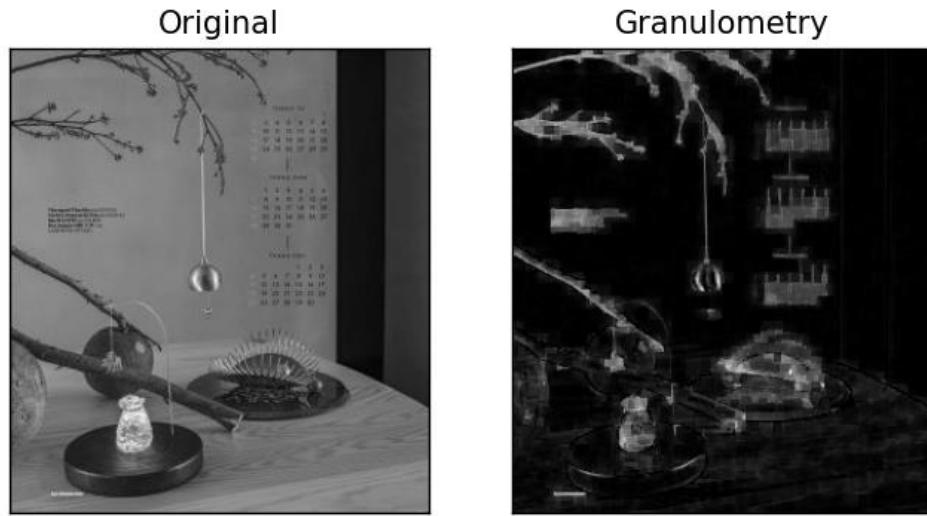
Original



Granulometry



Dùng mặt nạ kernel



Tương tự như vậy, thực hiện khi không sử dụng các hàm thư viện

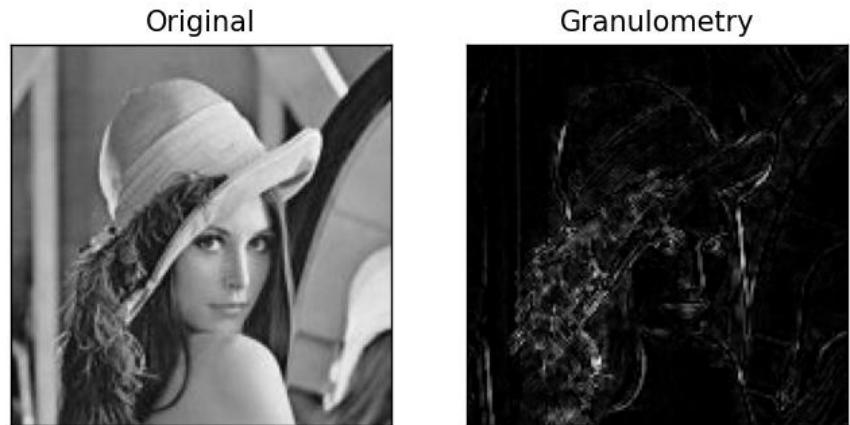
```
from opening import opening
from closing import closing

def granulometry_cvt(image, kernel):
    # Thực hiện toán tử opening
    opening_img = opening(image, kernel)
    # Thực hiện toán tử closing
    clos_img = closing(image, kernel)

    return clos_img - opening_img
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1



## Dùng mặt nạ kernel\_4

Nhìn chung, vẫn chưa thực hiện được chính xác toán tử Granulometry, tuy vậy có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Granulometry ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## 7. Toán tử Reconstruction

Một cách đơn giản, toán tử morphology reconstruction trong ảnh độ xám có thể được mô tả như sau: giả sử chúng ta có một ảnh độ xám và một cấu trúc hình học như một hình vuông hoặc một đường thẳng. Chúng ta muốn tìm tất cả các pixel trong ảnh độ xám mà nằm trong

cấu trúc đó. Toán tử morphology reconstruction sẽ thực hiện việc này bằng cách khóp cấu trúc với ảnh độ xám và sau đó lặp lại quá trình này để loại bỏ tất cả các pixel không thuộc cấu trúc.

Toán tử này có các đặc điểm sau:

- Toán tử reconstruction được sử dụng để tái tạo một tín hiệu ban đầu từ các dữ liệu mẫu hay dữ liệu bị nén.
- Toán tử này là một toán tử phi tuyến, có nghĩa là nó không thể được biểu diễn dưới dạng ma trận. Thay vào đó, nó được mô tả bằng một công thức toán học phức tạp.
- Toán tử reconstruction thường được sử dụng trong các ứng dụng như phục hồi ảnh hoặc giảm thiểu nhiễu trong tín hiệu.
- Để áp dụng toán tử này, cần phải sử dụng các phương pháp tính toán số để giải quyết công thức toán học phức tạp.
- Khi sử dụng toán tử reconstruction, cần phải đánh giá chất lượng của tín hiệu tái tạo bằng các chỉ số định lượng như sai số tương đối hay độ tương đồng giữa tín hiệu ban đầu và tín hiệu tái tạo.

Quá trình này có thể được tóm tắt thành các bước sau:

- Tạo một hình ảnh ban đầu (ban đầu là ảnh đầu vào).
- Áp dụng một toán tử morphological như mở hoặc đóng vào ảnh ban đầu.
- Lặp lại quá trình 2 cho đến khi tất cả các pixel trong ảnh đó đều thuộc cấu trúc hình học đã cho.
- Kết quả cuối cùng sẽ là ảnh đã được tái tạo, chỉ chứa các pixel nằm trong cấu trúc đã cho.

```
def main():
    img = Read_Img()
    exe = Choose_Kernel()

    recon = np.copy(img)
    prev = np.zeros_like(img)

    # Lặp lại quá trình tái tạo cho đến khi không còn thay đổi
    while not np.array_equal(prev, recon):
        prev = np.copy(recon)
        # Áp dụng toán dilation giữa G và B
        dilation = cv2.dilate(recon, Kernel(exe), iterations = 1 )
        # Áp dụng toán erosion giữa kết quả và ảnh đầu vào f
        recon = cv2.bitwise_and(dilation, img)

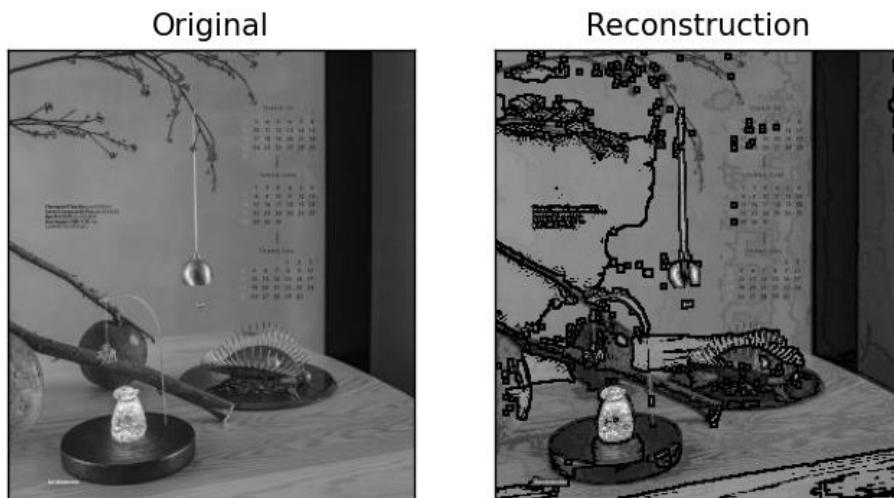
    # Hiển thị ảnh kết quả
    plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(1,2,2), plt.title("Reconstruction"), plt.imshow(recon, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Để thực hiện toán tử reconstruction trên ảnh xám mà không sử dụng thư viện cv2, chúng ta có thể làm theo các bước sau đây:

- Xác định kernel, kích thước của kernel sẽ phù hợp với tình huống của ảnh.
- Thực hiện phép toán dilation ban đầu trên ảnh đầu vào bằng cách dùng kernel.
- Thực hiện phép toán erosion bằng cách sử dụng kernel trên kết quả dilation ở bước trước.
- So sánh kết quả với dilation ban đầu. Nếu chúng không khác nhau thì kết quả đã đạt được, kết thúc thuật toán.
- Nếu chúng khác nhau, gán kết quả hiện tại cho dilation và quay lại bước 4.
- Trả về kết quả tái tạo hình dáng đã loại bỏ các đối tượng không mong muốn.

```
def reconstruction(image, kernel):
    # Khởi tạo kết quả ban đầu là hình ảnh đầu vào
    result = image.copy()

    # Thực hiện dilation ban đầu
    dilation = np.zeros_like(result)
    dilation = convolution_dilation(kernel, result)

    # Lặp lại quá trình thực hiện dilation và erosion cho đến khi không còn thay đổi nữa nữa
    while not np.array_equal(result, dilation):
        result = dilation.copy()
        dilation = convolution_dilation(kernel, result)
        dilation = erosion(dilation, kernel)

    return result
```

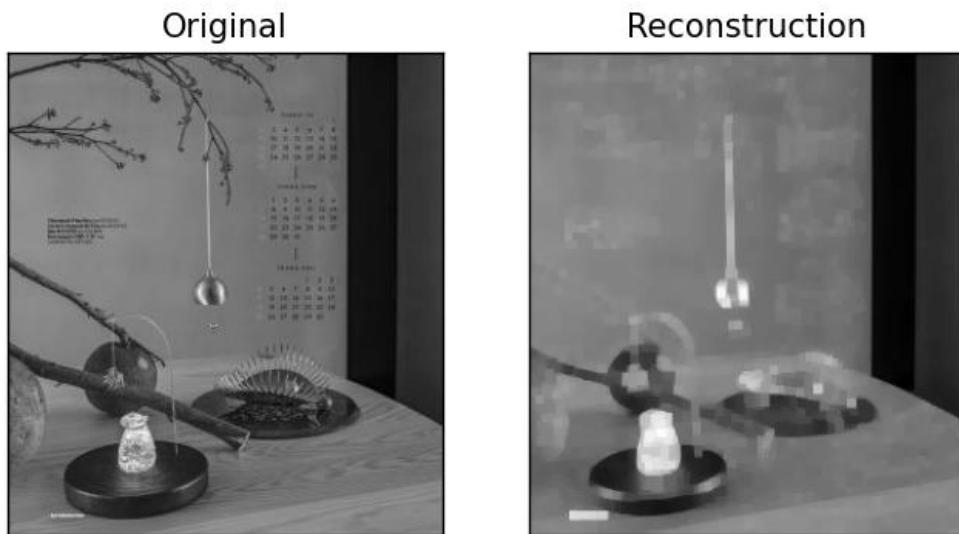
Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1





Dùng mặt nạ kernel\_4



Nhìn chung, vẫn chưa thực hiện được chính xác toán tử Reconstruction, tuy vậy vẫn chưa có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Reconstruction ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## 8. Toán tử Smoothing (làm trơn)

Toán tử Smooth Morphology trên ảnh độ xám là một phương pháp xử lý ảnh để giảm nhiễu và làm mịn bì mặt ảnh. Nó được sử dụng để cải thiện chất lượng của ảnh và giúp cho các phép nhận dạng và phân loại hình ảnh được chính xác hơn.

### **Định nghĩa**

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

Đặc điểm của toán tử làm trơn bao gồm:

- Làm giảm độ sắc nét của các cạnh trong ảnh, làm cho chúng trở nên mờ hơn.
- Làm giảm các biến động nhỏ trong ảnh, giúp làm giảm nhiễu và tăng độ tương phản.
- Thường được thực hiện bằng cách sử dụng các bộ lọc như bộ lọc Gauss hoặc bộ lọc trung bình để tính toán giá trị trung bình của các pixel xung quanh một pixel cụ thể.
- Toán tử làm trơn có thể được sử dụng để chuẩn bị ảnh cho các phép nhận dạng hoặc phân loại hình ảnh.
- Toán tử làm trơn có thể được kết hợp với các phép xử lý ảnh khác như phép lọc và phép biến đổi để tăng cường tính năng của ảnh.
- Toán tử làm trơn có thể được sử dụng trong nhiều lĩnh vực, bao gồm xử lý ảnh y tế, xử lý ảnh kỹ thuật số và xử lý ảnh nghệ thuật.

Để thực hiện toán tử Smooth Morphology trên ảnh bằng thư viện OpenCV, bạn có thể thực hiện theo các bước sau:

- Đọc ảnh đầu vào bằng hàm `cv2.imread()` và chuyển sang định dạng ảnh độ xám nếu cần thiết bằng hàm `cv2.cvtColor()`.
- Tạo kernel (còn gọi là cửa sổ) bằng hàm `cv2.getStructuringElement()` với các tham số kích thước kernel và hình dạng (thường là hình vuông hoặc hình tròn).

- Thực hiện phép biến đổi Smooth Morphology bằng hàm cv2.morphologyEx() với các tham số ảnh đầu vào, loại phép biến đổi (thường là Erosion hoặc Dilatation), kernel và số lần lặp lại.

```
img = Read_Img()
exe = Choose_Kernel()

# Thực hiện toán tử opening
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, Kernel(exe))

# Perform Grayscale smoothing
smooth = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, Kernel(exe))    #thực hiện tiếp với ảnh sau khi biến đổi với toán tử opening theo công thức

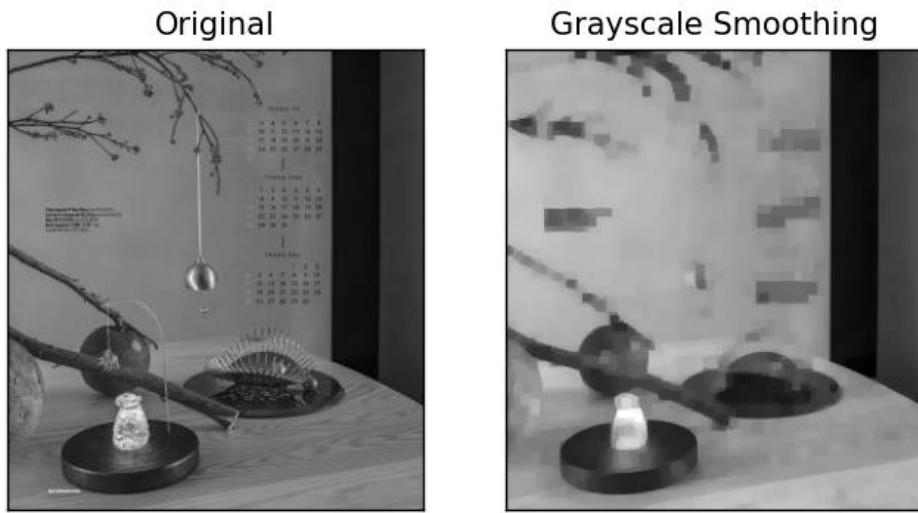
# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"),plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Grayscale Smoothing"),plt.imshow(smooth, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



## Dùng mặt nạ kernel



Các bước thực hiện toán tử Smooth Morphology khi không sử dụng OpenCV có thể được thực hiện như sau:

- Chuẩn bị ảnh đầu vào: Đầu tiên, bạn cần chuẩn bị ảnh đầu vào để áp dụng toán tử Smooth Morphology. Ảnh có thể là ảnh độ xám hoặc ảnh màu.
- Tạo kernel: Sau đó, bạn cần tạo một kernel để áp dụng các phép biến đổi Smooth Morphology. Kernel là một ma trận có kích thước cụ thể và được sử dụng để xác định vùng lân cận của mỗi pixel trong ảnh. Kernel có thể được tạo bằng cách sử dụng các hàm như Gaussian hoặc Box.
- Sau đó thực hiện theo định nghĩa, xử lý các toán tử opening và closing

```
def smooth_cvt(image, kernel):
    # Thực hiện toán tử opening
    opening_img = opening(image, kernel)

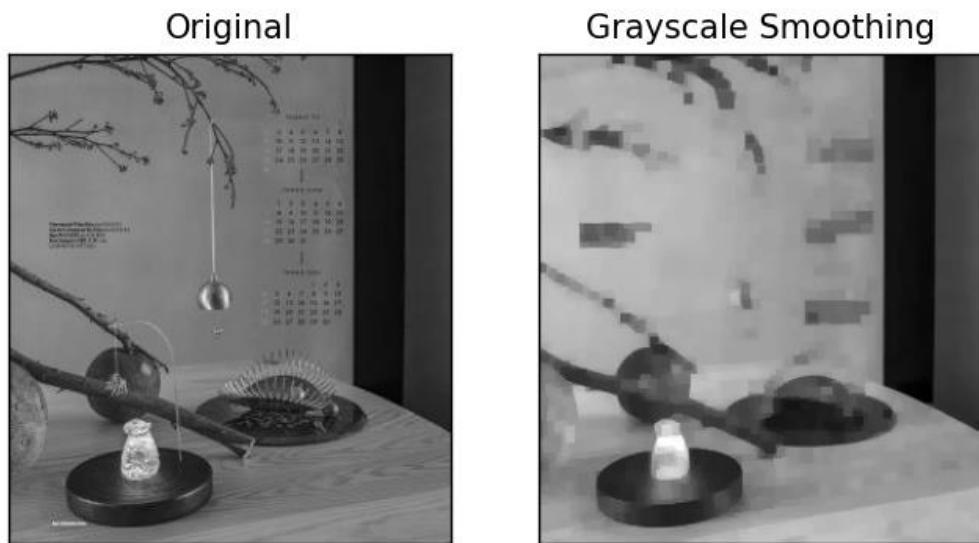
    return closing(np.array(opening_img), kernel)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

## Dùng mặt nạ kernel\_1



Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Smoothing ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## 9. Toán tử Textual Segmentation

Toán tử Textual segmentation morphology là một phép biến đổi hình thái học được áp dụng trên ảnh độ xám để phân đoạn văn bản và các ký tự trong ảnh.

Phép biến đổi này thường được sử dụng để loại bỏ các chi tiết không cần thiết trong ảnh, tạo ra các vùng phân tách rõ ràng giữa các ký tự và đối tượng trong ảnh, giúp cho việc xử lý văn bản và nhận dạng ký tự trở nên dễ dàng hơn.

### Định nghĩa

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

Các đặc điểm chính của toán tử Textual segmentation morphology trên ảnh độ xám bao gồm:

- Toán tử này được áp dụng để phân đoạn văn bản và các ký tự trong ảnh độ xám, tạo ra các vùng phân tách rõ ràng giữa các đối tượng trong ảnh.
- Các toán tử Textual segmentation morphology thường được sử dụng để loại bỏ các chi tiết không cần thiết trong ảnh, tạo ra các đối tượng văn bản và ký tự có độ tương phản và độ sắc nét tốt hơn.
- Các toán tử này thường được áp dụng sau khi ảnh đã được xử lý bằng các phép biến đổi khác như làm mịn, cân bằng sáng tối, hoặc phân đoạn ngưỡng.

Để thực hiện toán tử Textual segmentation morphology trên ảnh độ xám bằng OpenCV, bạn có thể thực hiện các bước sau:

- Đọc ảnh độ xám và chuyển đổi sang ảnh nhị phân nếu cần thiết bằng các phép biến đổi như làm mịn (blurring), phân đoạn ngưỡng (thresholding)...
- Chọn và khởi tạo các toán tử morphology thích hợp, ví dụ như toán tử mở rộng (dilation), toán tử thu nhỏ (erosion), toán tử mở rộng đóng (closing), toán tử thu nhỏ mở (opening)...

- Áp dụng toán tử morphology đã chọn và khởi tạo lên ảnh.
- Thực hiện các phép biến đổi như trong định nghĩa

```



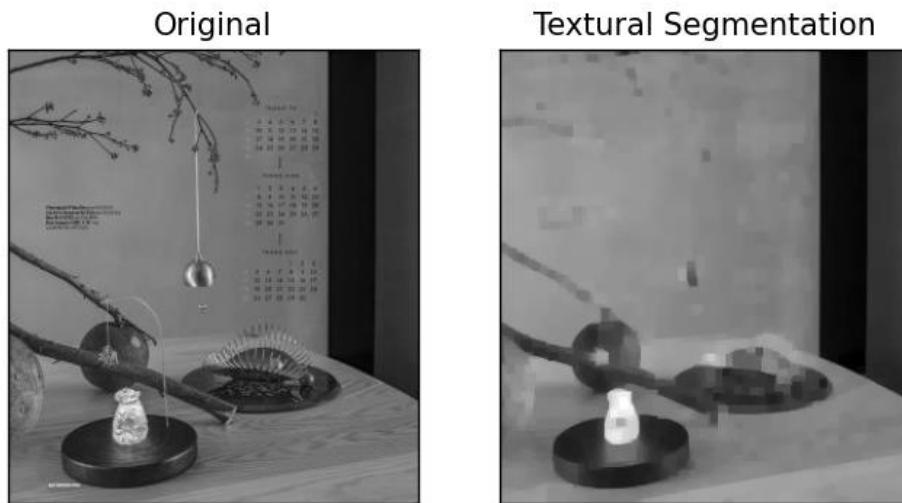
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker



Dùng mặt nạ kerl



Để thực hiện toán tử Textual segmentation morphology trên ảnh độ xám khi không sử dụng OpenCV, bạn có thể thực hiện các bước sau:

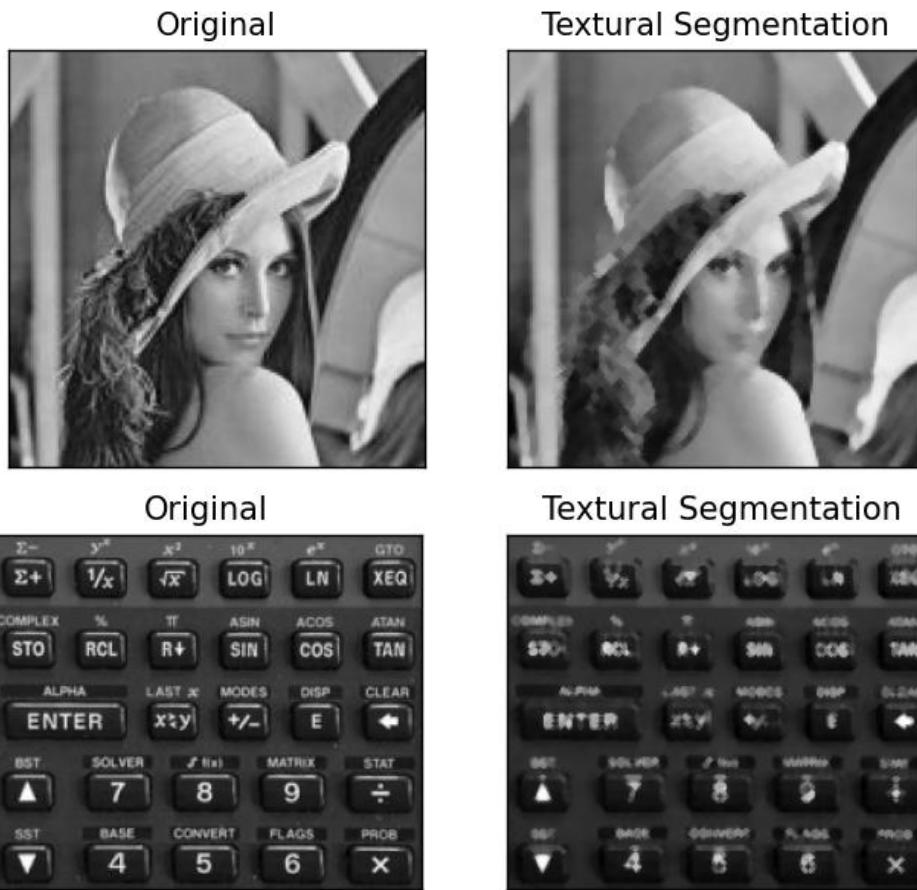
- Chuyển ảnh mà bạn muốn xử lý sang ảnh độ xám nếu ảnh ban đầu là ảnh màu.
- Áp dụng phép biến đổi ngưỡng (thresholding) để chuyển ảnh thành ảnh nhị phân, với giá trị ngưỡng được chọn sao cho tách được ký tự và đối tượng văn bản.
- Áp dụng các toán tử morphology như toán tử mở rộng (dilation) và thu nhỏ (erosion) để loại bỏ các chi tiết không cần thiết và tạo ra các vùng phân tách rõ ràng giữa các ký tự và đối tượng trong ảnh.
- Nếu cần thiết, bạn có thể thực hiện các bước tiền xử lý khác như loại bỏ nhiễu, phân đoạn vùng quan tâm, hoặc sử dụng các phép biến đổi khác để tăng độ tương phản và độ sắc nét của các ký tự và đối tượng văn bản.
- Cuối cùng, bạn có thể thực hiện các bước nhận dạng ký tự và văn bản để trích xuất thông tin từ ảnh đã được phân đoạn.

```
def textural(image, kernel):
    # Thực hiện toán tử closing
    closing_img = closing(image, kernel)

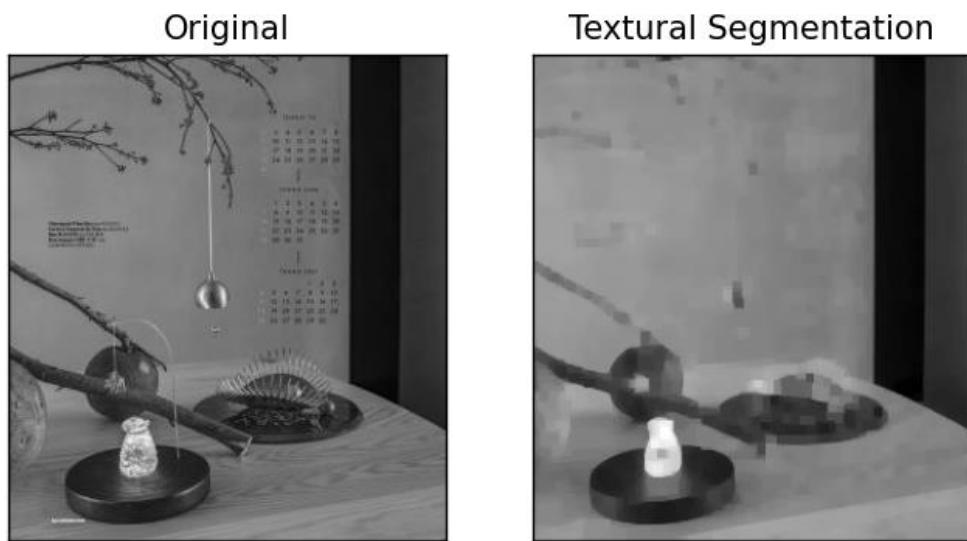
    return opening(np.array(closing_img), kernel)
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ kernel\_1



Dùng mặt nạ kernel\_4



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Textual Segmentation ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không

sử dụng hàm thư viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## 10. Toán tử Top Hat

Toán tử Top Hat Morphology trên ảnh độ xám là một kỹ thuật xử lý ảnh để tìm kiếm các cấu trúc đặc biệt trong ảnh, chẳng hạn như cạnh hoặc đối tượng nhỏ. Toán tử Top Hat Morphology thực hiện bằng cách so sánh ảnh gốc với một mẫu được gọi là phép biến đổi Top Hat. Khi được áp dụng cho ảnh độ xám, phép biến đổi Top Hat tìm kiếm các cấu trúc có kích thước nhỏ hơn kích thước của phép biến đổi.

### Định nghĩa

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

Một số đặc điểm của toán tử Top Hat Morphology trên ảnh độ xám:

- Tìm kiếm các cấu trúc nhỏ: Toán tử Top Hat Morphology tìm kiếm các cấu trúc có kích thước nhỏ hơn kích thước của phép biến đổi Top Hat. Điều này cho phép tìm kiếm các đối tượng nhỏ hơn trong ảnh.
- Giảm nhiễu: Toán tử Top Hat Morphology có thể giảm nhiễu trong ảnh độ xám bằng cách lọc ảnh độ xám bằng phép nhân Morphological Opening trước khi áp dụng phép biến đổi Top Hat.
- Dễ sử dụng: Toán tử Top Hat Morphology dễ sử dụng và được tính toán nhanh chóng, cho phép xử lý ảnh nhanh chóng và hiệu quả.
- Có thể được kết hợp với các phép biến đổi khác: Toán tử Top Hat Morphology có thể được kết hợp với các phép biến đổi khác để tăng độ chính xác và cải thiện kết quả, chẳng hạn như toán tử Erosion hoặc Dilatation.

Trong OpenCV, toán tử Top Hat Morphology trên ảnh độ xám có thể được thực hiện bằng cách sử dụng hàm cv2.morphologyEx() với tham số cv2.MORPH\_TOPHAT.

`cv2.morphologyEx(img, cv2.MORPH_TOPHAT, Kernel)`

Cụ thể, hàm cv2.morphologyEx() được sử dụng để thực hiện các phép biến đổi hình thái trên ảnh đầu vào img. Tham số thứ hai là kiểu phép biến đổi hình thái, trong trường hợp này là cv2.MORPH\_TOPHAT để thực hiện toán tử Top Hat Morphology. Tham số thứ ba là kernel được sử dụng trong phép biến đổi, được xác định bởi biến Kernel đã được tạo trước đó.

```
img = Read_Img()
exe = Choose_Kernel()

#thực hiện toán tử Top-hat transformation
tophat = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, Kernel(exe))

# Hiển thị ảnh kết quả
plt.subplot(1,2,1), plt.title("Original"), plt.imshow(img, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.title("Top Hat"), plt.imshow(tophat, cmap = "gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

Kết quả khi sử dụng hàm thư viện có sẵn (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

Dùng mặt nạ ker

Original



Top Hat



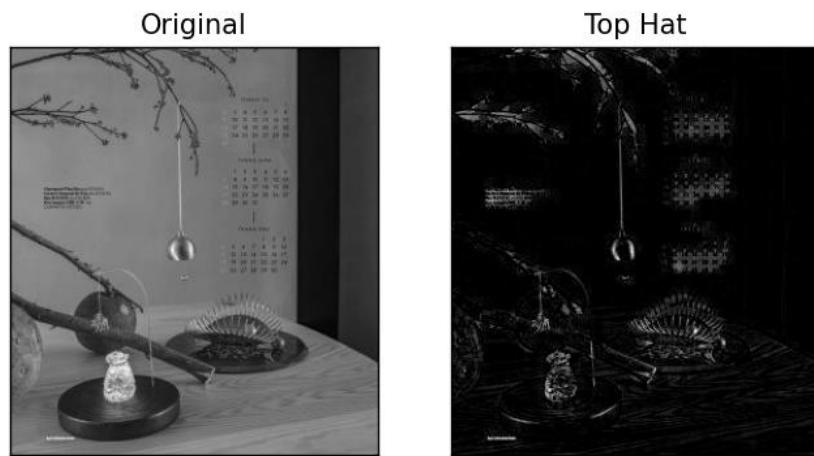
Original



Top Hat



### Dùng mặt nạ kernel



Để thực hiện toán tử Top Hat Morphology trên ảnh độ xám mà không sử dụng OpenCV, ta có thể thực hiện các bước sau:

- Đọc ảnh độ xám gốc.
- Tạo một mặt nạ (kernel) kích thước NxN, với N là số lẻ. Mặt nạ này được sử dụng để thực hiện phép biến đổi Morphological Opening.
- Thực hiện phép biến đổi Morphological Opening trên ảnh độ xám gốc bằng cách lọc ảnh với mặt nạ vừa tạo.
- Trừ ảnh độ xám gốc với ảnh được lọc bằng phép nhân Morphological Opening. Kết quả được gọi là phép biến đổi Top Hat sau khi biến đổi theo định nghĩa lúc đầu.

```
def Top_Hat(image, kernel):
    # Thực hiện toán tử opening
    opening_img = opening(image, kernel)

    return image - opening_img
```

Kết quả khi không sử dụng hàm thư viện (sử dụng các hình ảnh *Lenna.jpg*, *decor.jpg*, *inp.png*)

### Dùng mặt nạ kernel\_1

Original



Top Hat



Original



Top Hat

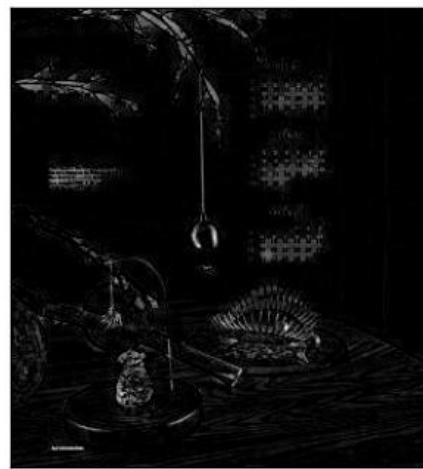


Dùng mặt nạ kernel\_4

Original



Top Hat



Nhìn chung, có điểm tương đồng về ảnh kết quả giữa hai thuật toán của toán tử Top Hat ở các mặt nạ con (kernel) giống nhau khi sử dụng hàm thư viện và khi không sử dụng hàm thư

viện. Nhưng về tốc độ xử lý, tốc độ thực hiện để trả về kết quả thì khi sử dụng hàm thư viện có tốc độ nhanh hơn hẳn nhưng không đáng kể.

## V. Thủ tạo nhiễu và lọc nhiễu (tăng cường) qua Opening và Closing

Sử dụng hàm không dùng thư viện vẫn thực hiện tốt

```
from header_bin_noisy import *
from opening import opening
from closing import closing

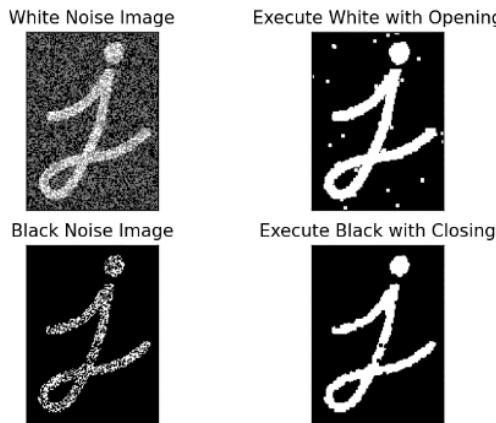
def main():
    binary_im, _ = Read_Img()
    exe = Choose_Kernel()

    binary_img = np.array(binary_im)
    #Xử lý khi ảnh bị nhiễu
    #white noise & black noise
    noise = np.random.randint(0,2, size = binary_img.shape[:2])
    white_noise_img = noise * 255 + binary_img      #chèn thêm độ nhiễu sáng vào ảnh nhị phân
    black_noise_img = noise * -255 + binary_img
    black_noise_img[black_noise_img < -245] = 0      #cho độ nhiễu trong phạm vi đối tượng

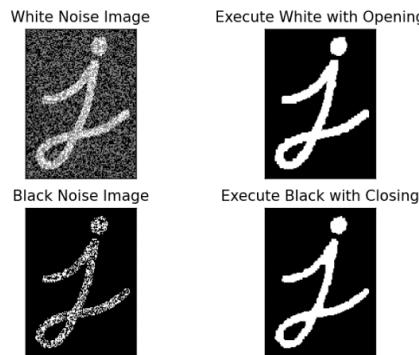
    #execute noising with morphology Open and Close
    af_noise = opening(np.array(white_noise_img.astype(np.float32)), np.array(Kernel(exe)))
    af_black_nois = closing(np.array(black_noise_img.astype(np.float32)), np.array(Kernel(exe)))

    #show results
    plt.subplot(2,2,1), plt.title("White Noise Image"),plt.imshow(white_noise_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(2,2,2), plt.title("Execute White with Opening"), plt.imshow(af_noise, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(2,2,3), plt.title("Black Noise Image"), plt.imshow(black_noise_img, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.subplot(2,2,4), plt.title("Execute Black with Closing"), plt.imshow(af_black_nois, cmap = "gray"), plt.xticks([]), plt.yticks([])
    plt.show()
```

Dùng mặt nạ kernel\_2

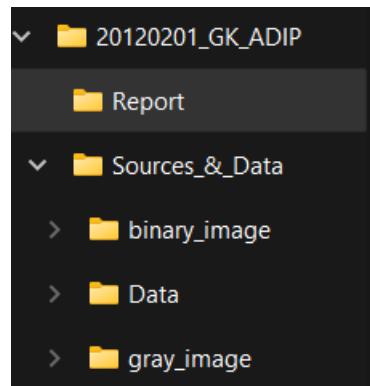


Dùng mặt nạ kernel\_4 lọc tốt nhất



## VI. Hướng dẫn chương trình

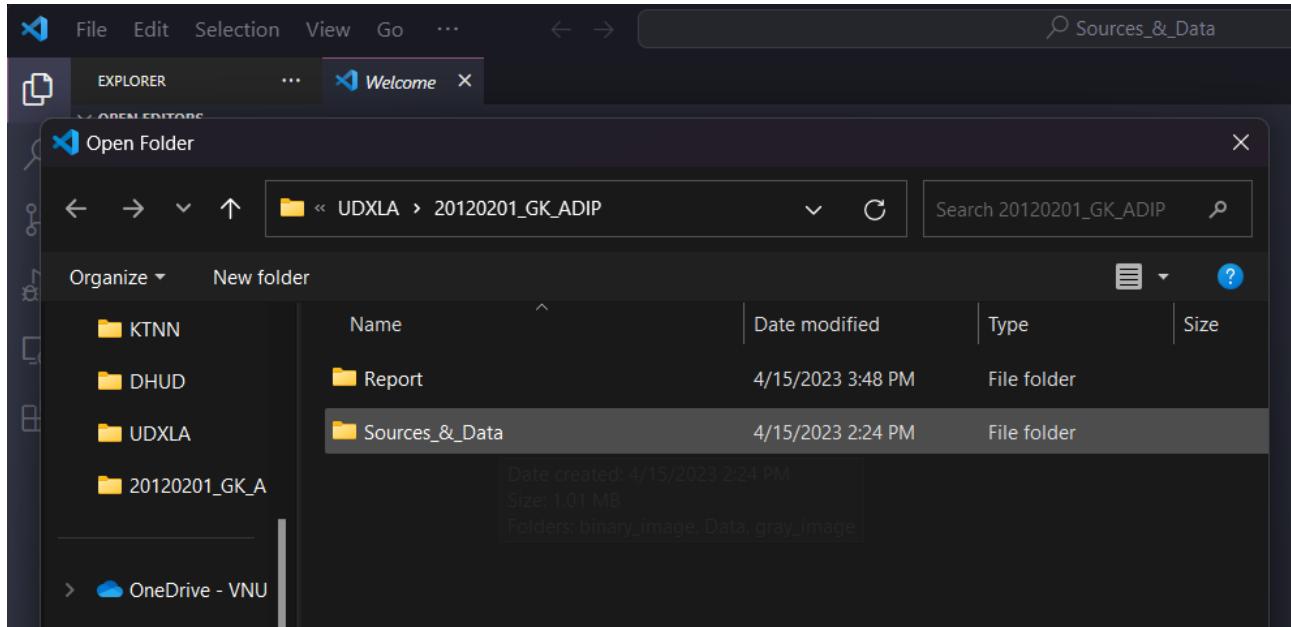
### 1. Sơ đồ tổ chức thư mục



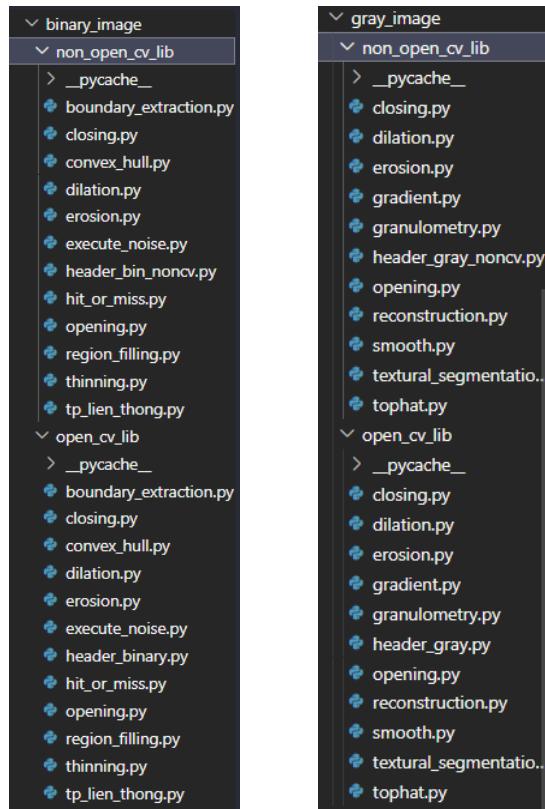
Trong đó, folder Report sẽ chứa các file báo cáo gồm có file .doc và file .pdf , folder Sources\_&\_Data chứa các đoạn mã nguồn của chương trình và dữ liệu hình ảnh mà chương trình cần dùng.

### 2. Cách khởi chạy chương trình

Dùng phần mềm VSCode để thực thi chương trình, chọn File và Open Folder

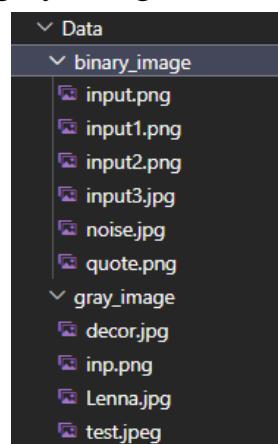


Chọn chương trình cần chạy trong các folder binary\_image và gray\_image



A screenshot of the Visual Studio Code interface. The top navigation bar shows tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL, with TERMINAL being the active tab. To the right of the terminal are icons for Python, a plus sign, a file, a microphone, and other settings. The main area is a terminal window displaying a command-line session. The command `PS D:\Bai\_hoc\_moingay\UDXLA\20120201\_GK\_ADIP\Sources\_&\_Data> & C:/Users/thong/AppData/Local/Programs/Python/Python310/python.exe "d:/Bai\_hoc\_moingay/UDXLA/20120201\_GK\_ADIP/Sources\_&\_Data/binary\_image/open\_cv\_lib/dilation.py" Enter your file picture name (file from Data/binary\_image):` is shown, along with a cursor at the end of the line.

Nhập tên của tập tin hình ảnh cần xử lý theo đúng mục của folder Data đã được cấu hình sẵn. Chạy chương trình ở folder binary\_image thì sử dụng hình ảnh trong Data/binary\_image, tương tự với gray\_image



Giả sử nhập sai tập tin hình ảnh đầu vào

```
IP/Sources_&_Data/binary_image/open_cv_lib/dilation.py"
Enter your file picture name (file from Data/binary_image): Lenna.jpg
Tập tin không tồn tại trong thư mục hiện tại
Enter your file picture name (file from Data/binary_image):
```

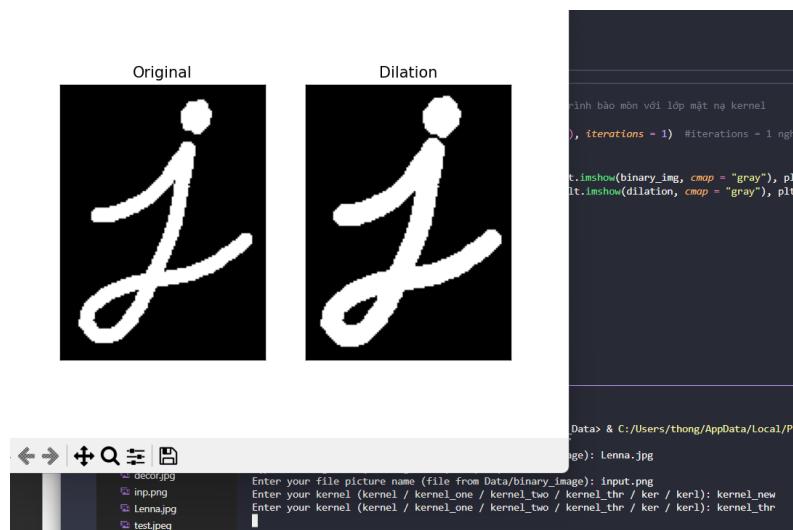
Khi nhập đúng, chương trình tiếp tục cho nhập mặt nạ kernel cho việc xử lý toán tử morphology

```
Enter your file picture name (file from Data/binary_image): input.png
Enter your kernel (kernel / kernel_one / kernel_two / kernel_thr / ker / kerl):
```

Nếu nhập không đúng các kernel được đề xuất sẵn này, chương trình sẽ bắt nhập lại đến khi đúng

```
Enter your kernel (kernel / kernel_one / kernel_two / kernel_thr / ker / kerl): kernel_new
Enter your kernel (kernel / kernel_one / kernel_two / kernel_thr / ker / kerl):
```

Khi nhập đúng, chương trình trả ra kết quả



Và cứ thế, xử lý lần lượt các tập tin chương trình mong muốn được xử lý trong folder binary\_image và gray\_image

---

## TÀI LIỆU THAM KHẢO

---

### Danh mục tài liệu tham khảo:

- [1] [https://docs.opencv.org/4.x/d9/d61/tutorial\\_py\\_morphological\\_ops.html](https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html)
- [2] <https://scikit-image.org/docs/stable/api/skimage.morphology.html>
- [3] <https://www.amazon.com/Digital-Image-Processing-Rafael-Gonzalez/dp/013168728X>
- [4] <https://www.amazon.com/Mathematical-Morphology-Applications-Image-Processing/dp/1447113902>
- [5] <https://www.amazon.com/Handbook-Mathematical-Morphology-Jean-Serra/dp/146137908X>