

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC BÁCH KHOA**  
**KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH**



**BÁO CÁO**  
**DIGITAL IMAGE PROCESSING AND COMPUTER VISION**  
**PROJECT 2**  
**FILTER**

**GVHD:** Ths. Võ Thanh Hùng

—o0o—

**SVTH:** Nguyễn Tiến Phát - 2011797

TP. HỒ CHÍ MINH, 3/2024

# Mục lục

<b>1 Giới thiệu</b>	<b>1</b>
1.1 Khái niệm về lọc ảnh (image filtering) . . . . .	1
1.2 Phương pháp . . . . .	1
1.2.1 Low-pass filter . . . . .	1
1.2.2 High-pass filter . . . . .	1
<b>2 Hiệu thực</b>	<b>2</b>
2.1 Cơ sở lý thuyết . . . . .	2
2.1.1 Low-pass filter . . . . .	2
2.1.1.1 Average Filter . . . . .	2
2.1.1.2 Median Filter . . . . .	2
2.1.1.3 Gaussian Filter . . . . .	2
2.1.2 High-pass filter . . . . .	3
2.1.2.1 Laplacian Filter . . . . .	3
2.1.2.2 Sobel Filter . . . . .	3
2.1.2.3 Canny Filter . . . . .	4
2.2 Hiệu thực với python . . . . .	4
2.2.1 Low-pass filter . . . . .	4
2.2.1.1 Average filter . . . . .	5
2.2.1.2 Median filter . . . . .	7
2.2.1.3 Gaussian filter . . . . .	9
2.2.2 High-pass filter . . . . .	11
2.2.2.1 Laplacian filter . . . . .	11
2.2.2.2 Sobel filter . . . . .	12
2.2.2.3 Canny filter . . . . .	14
<b>3 Kết quả đạt được và tổng kết</b>	<b>17</b>
3.1 Thủ với những hình ảnh khác . . . . .	17
3.1.1 Low-pass filter . . . . .	17
3.1.1.1 Ảnh 1: . . . . .	17
3.1.1.2 Ảnh 2: . . . . .	18
3.1.2 High-pass filter . . . . .	19
3.1.2.1 Laplacian filter . . . . .	19
3.1.2.2 Sobel filter . . . . .	19
3.1.2.3 Canny filter . . . . .	20
3.2 Kết quả đạt được . . . . .	20
3.2.1 Low-pass Filters . . . . .	21
3.2.2 High-pass Filters . . . . .	21
3.3 Tổng kết . . . . .	21
3.4 Tài liệu tham khảo . . . . .	22
3.5 Phụ lục . . . . .	23

# Danh sách hình vẽ

2.1	Ảnh sau khi đã được thêm noise . . . . .	5
2.2	Average filter with Gaussian noise image . . . . .	6
2.3	Average filter with Salt and pepper noise image . . . . .	6
2.4	Average filter with Impluse noise image . . . . .	6
2.5	Average filter with Gaussian noise image . . . . .	6
2.6	Average filter with Salt and pepper noise image . . . . .	7
2.7	Average filter with Impluse noise image . . . . .	7
2.8	Median filter with Gaussian noise image . . . . .	8
2.9	Median filter with Salt and pepper noise image . . . . .	8
2.10	Median filter with Impluse noise image . . . . .	8
2.11	Median filter with Gaussian noise image . . . . .	8
2.12	Median filter with Salt and pepper noise image . . . . .	8
2.13	Median filter with Impluse noise image . . . . .	9
2.14	Gaussian filter with Gaussian noise image . . . . .	9
2.15	Gaussian filter with Salt and pepper noise image . . . . .	10
2.16	Gaussian filter with Impluse noise image . . . . .	10
2.17	Gaussian filter with Gaussian noise image . . . . .	10
2.18	Gaussian filter with Salt and pepper noise image . . . . .	10
2.19	Gaussian filter with Impluse noise image . . . . .	10
2.20	Laplacian filter by hand . . . . .	11
2.21	Laplacian filter with cv2 . . . . .	12
2.22	Sobel filter by hand . . . . .	13
2.23	Sobel filter with cv2 . . . . .	14
2.24	Canny filter by hand . . . . .	15
2.25	Canny filter with cv2 . . . . .	16
3.1	Origin noise image . . . . .	17
3.2	Average filter with kernel size = 5 . . . . .	17
3.3	Median filter with kernel size = 5 . . . . .	17
3.4	Gaussian filter with kernel size = 5 . . . . .	18
3.5	Origin noise image . . . . .	18
3.6	Average filter with kernel size = 5 . . . . .	18
3.7	Median filter with kernel size = 5 . . . . .	18
3.8	Gaussian filter with kernel size = 5 . . . . .	18
3.9	Laplacian filter . . . . .	19
3.10	Laplacian filter . . . . .	19
3.11	Sobel filter . . . . .	19
3.12	Sobel filter . . . . .	20
3.13	Canny filter . . . . .	20
3.14	Canny filter . . . . .	20

# Danh sách bảng

# Chương 1

## Giới thiệu

### 1.1 Khái niệm về lọc ảnh (image filtering)

Lọc ảnh là một quá trình quan trọng trong xử lý ảnh. Lọc ảnh có thể có một hay nhiều tác dụng bao gồm: cải thiện chất lượng hình ảnh bằng cách loại bỏ nhiễu, làm mịn, tìm biên đối tượng.

Cơ bản, lọc ảnh dựa trên nguyên tắc nhân ma trận ảnh với ma trận lọc (hay còn gọi là Kernel). Quá trình này giống việc trượt Kernel qua ảnh và tính toán kết quả cho điểm ảnh trung tâm dựa trên các giá trị xung quanh nó. Có hai phép lọc chính là tương quan (correlation) và tích chập (convolution). Chúng khá giống nhau, điểm khác biệt chính là ma trận lọc được xoay 180 độ trong phép tích chập. Với mỗi phép lọc, ta có những ma trận lọc (Kernel) khác nhau tùy thuộc vào mục đích của bộ lọc đó. Không có quy định cụ thể của việc xác định kích thước của bộ lọc, nhưng thường dùng một ma trận số lẻ, ví dụ 3x3, 5x5. Một lưu ý nữa cũng khá quan trọng, đó là tổng các phần tử trong Kernel thường bằng 1 để đảm bảo độ sáng của ảnh sau khi lọc không thay đổi độ sáng so với ảnh gốc.

### 1.2 Phương pháp

Ở bài tập lớn lần này, chúng ta sẽ thực hiện 2 nhóm bài toán lớn của lọc ảnh bao gồm low-pass filter và high-pass filter

#### 1.2.1 Low-pass filter

Low-pass filter là một loại bộ lọc được sử dụng để loại bỏ các thành phần cao tần từ một tín hiệu, giữ lại các thành phần thấp tần. Trong xử lý ảnh, low-pass filter thường được sử dụng để làm mịn hình ảnh và loại bỏ nhiễu.

**Ứng dụng:** Low-pass filter được sử dụng để làm mịn hình ảnh, giúp giảm nhiễu và tạo ra các phiên bản mờ của hình ảnh.

**Phương pháp thực hiện:** Để thực hiện low-pass filter, chúng ta sử dụng một ma trận lọc có trọng số thấp, giảm các biến thiên nhanh chóng trong ảnh và làm mịn các vùng đồng nhất. Các phương pháp có thể kể đến như Average, Median, Gaussian.

#### 1.2.2 High-pass filter

High-pass filter là một loại bộ lọc được sử dụng để tạo ra các phiên bản tăng cường của hình ảnh bằng cách loại bỏ các thành phần thấp tần và tăng cường các biến thiên nhanh chóng trong ảnh.

**Ứng dụng:** High-pass filter được sử dụng để làm nổi bật các đặc điểm cạnh và chi tiết trong hình ảnh, giúp trong việc phát hiện biên của các đối tượng.

**Phương pháp thực hiện:** Để thực hiện high-pass filter, chúng ta sử dụng một ma trận lọc có trọng số cao, làm nổi bật các biến thiên nhanh chóng trong ảnh và giảm mờ các vùng đồng nhất. Các phương pháp có thể kể đến như Sobel, Canny hay Laplacian.

# Chương 2

## Hiện thực

### 2.1 Cơ sở lý thuyết

#### 2.1.1 Low-pass filter

##### 2.1.1.1 Average Filter

Average filter là một dạng đơn giản của low-pass filter, nó sử dụng một ma trận lọc có tất cả các giá trị bằng nhau và tổng các giá trị trong ma trận lọc là 1. Phương pháp này làm mịn hình ảnh bằng cách thay thế mỗi điểm ảnh bằng giá trị trung bình của các điểm ảnh lân cận trong hình chữ nhật hoặc hình vuông.

Công thức toán học cho average filter có thể được biểu diễn như sau:

$$\text{Output}(x,y) = \frac{1}{n \times m} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \text{Input}(x+i, y+j)$$

Trong đó:

- Output( $x,y$ ) là giá trị của điểm ảnh mới sau khi áp dụng average filter tại tọa độ ( $x,y$ ) trong ảnh kết quả.
- Input( $x+i, y+j$ ) là giá trị của điểm ảnh tại tọa độ ( $x+i, y+j$ ) trong ảnh gốc.
- $n$  và  $m$  là kích thước của ma trận lọc,  $n \times m$  là tổng số điểm ảnh trong ma trận lọc.

##### 2.1.1.2 Median Filter

Median filter là một phương pháp xử lý ảnh được sử dụng để loại bỏ nhiễu trong hình ảnh bằng cách lấy giá trị trung vị của tất cả các điểm ảnh trong vùng kernel và thay thế phần tử trung tâm bằng giá trị trung vị này. Phương pháp này rất hiệu quả đối với nhiễu loại muối và tiêu (salt and pepper) trong một hình ảnh. Điều thú vị là trong các bộ lọc trên, phần tử trung tâm là một giá trị mới được tính toán có thể là một giá trị pixel trong ảnh hoặc một giá trị mới. Nhưng trong median blurring, phần tử trung tâm luôn được thay thế bằng một giá trị pixel trong ảnh. Điều này giúp giảm nhiễu một cách hiệu quả. Kích thước kernel của median filter nên là một số nguyên dương lẻ.

Công thức toán học cho median filter như sau:

$$\text{Output}(x,y) = \text{Median} \left\{ \text{Input}(x+i, y+j) \mid i \in [-\frac{n}{2}, \frac{n}{2}], j \in [-\frac{m}{2}, \frac{m}{2}] \right\}$$

Trong đó:

- Output( $x,y$ ) là giá trị của điểm ảnh mới sau khi áp dụng median filter tại tọa độ ( $x,y$ ) trong ảnh kết quả.
- Input( $x+i, y+j$ ) là giá trị của điểm ảnh tại tọa độ ( $x+i, y+j$ ) trong ảnh gốc

##### 2.1.1.3 Gaussian Filter

Gaussian filter là một phương pháp lọc ảnh được sử dụng rộng rãi trong xử lý ảnh để làm mịn và giảm nhiễu. Phương pháp này dựa trên việc áp dụng một ma trận lọc dựa trên hàm Gaussian vào ảnh gốc. Các giá trị trong ma trận lọc được tính toán từ hàm Gaussian, với trọng số giảm dần khi cách điểm trung tâm xa đi. Kết quả của Gaussian filter thường là một ảnh mịn hơn và ít nhiễu hơn so với ảnh gốc.

Công thức toán học cho Gaussian filter như sau:



$$\text{Output}(x, y) = \frac{1}{\sqrt{2\pi}\sigma} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \text{Input}(x+i, y+j) \cdot \exp\left(-\frac{i^2 + j^2}{2\sigma^2}\right)$$

Trong đó:

- Output( $x, y$ ) là giá trị của điểm ảnh mới sau khi áp dụng Gaussian filter tại tọa độ ( $x, y$ ) trong ảnh kết quả.
- Input( $x+i, y+j$ ) là giá trị của điểm ảnh tại tọa độ ( $x+i, y+j$ ) trong ảnh gốc.
- $n$  và  $m$  là kích thước của ma trận lọc.
- $\sigma$  là tham số đặc trưng của hàm Gaussian, quyết định mức độ mịn của ảnh kết quả.

## 2.1.2 High-pass filter

### 2.1.2.1 Laplacian Filter

Laplacian filter là một phương pháp xử lý ảnh được sử dụng để làm nổi bật các biên cạnh và chi tiết trong hình ảnh bằng cách phát hiện các điểm thay đổi nhanh chóng trong độ sáng. Phương pháp này thường được sử dụng sau khi đã làm mịn hình ảnh bằng low-pass filter để tạo ra hiệu ứng như làm nổi bật các đặc điểm cạnh.

Công thức toán học cho Laplacian filter có thể được biểu diễn như sau:

$$\text{Output}(x, y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \text{Input}(x+i, y+j) \cdot \text{Kernel}(i, j)$$

Trong đó:

- Output( $x, y$ ) là giá trị của điểm ảnh mới sau khi áp dụng Laplacian filter tại tọa độ ( $x, y$ ) trong ảnh kết quả.
- Input( $x+i, y+j$ ) là giá trị của điểm ảnh tại tọa độ ( $x+i, y+j$ ) trong ảnh gốc.
- Kernel( $i, j$ ) là một ma trận lọc Laplacian được sử dụng để xác định cách điểm ảnh mới được tính toán. Ma trận lọc này thường là một ma trận có trung tâm âm và các giá trị xung quanh dương để phát hiện các biên cạnh.

$$\text{Kernel} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

### 2.1.2.2 Sobel Filter

Sobel filter là một phương pháp xử lý ảnh được sử dụng để phát hiện biên cạnh trong hình ảnh bằng cách áp dụng các bộ lọc gradient theo các hướng khác nhau. Cụ thể, Sobel filter sử dụng hai ma trận lọc 3x3 để tính đạo hàm bậc nhất theo hướng ngang và dọc của mỗi điểm ảnh trong ảnh.

Công thức toán học cho Sobel filter có thể được biểu diễn như sau:

$$\text{Output}_x(x, y) = \sum_{i=0}^2 \sum_{j=0}^2 \text{Input}(x+i-1, y+j-1) \cdot \text{Kernel}_x(i, j)$$

$$\text{Output}_y(x, y) = \sum_{i=0}^2 \sum_{j=0}^2 \text{Input}(x+i-1, y+j-1) \cdot \text{Kernel}_y(i, j)$$

Trong đó:

- Output<sub>x</sub>( $x, y$ ) là giá trị của gradient theo hướng ngang tại điểm ( $x, y$ ) trong ảnh kết quả.
- Output<sub>y</sub>( $x, y$ ) là giá trị của gradient theo hướng dọc tại điểm ( $x, y$ ) trong ảnh kết quả.
- Input( $x+i-1, y+j-1$ ) là giá trị của điểm ảnh tại vị trí ( $x+i-1, y+j-1$ ) trong ảnh gốc.
- Kernel<sub>x</sub> và Kernel<sub>y</sub> là hai ma trận lọc Sobel được sử dụng để tính toán gradient theo hai hướng. Các ma trận lọc này có giá trị khác nhau để phát hiện biên cạnh theo hướng ngang và dọc.

Cụ thể, hai ma trận lọc Sobel thường được định nghĩa như sau:

$$\text{Kernel}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\text{Kernel}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Cả hai ma trận lọc này đều có kết cấu simetric để phát hiện các biên cạnh trong ảnh.

### 2.1.2.3 Canny Filter

Canny filter là một trong những phương pháp phát hiện biên cạnh hiệu quả nhất trong xử lý ảnh. Phương pháp này được đề xuất bởi John Canny vào năm 1986 và bao gồm một loạt các bước như làm mịn ảnh, phát hiện độ dốc, làm sạch biên cạnh và thực hiện hysteresis thresholding để loại bỏ các biên cạnh giả mạo.

Canny filter có thể được mô tả như sau:

- Làm mịn ảnh:** Sử dụng một bộ lọc Gaussian để làm mịn ảnh và loại bỏ nhiễu.
- Phát hiện độ dốc:** Tính toán gradient của ảnh bằng cách sử dụng bộ lọc Sobel hoặc Prewitt.
- Làm sạch biên cạnh:** Sử dụng kỹ thuật non-maximum suppression để duy trì chỉ các điểm cực đại trên biên cạnh.
- Hysteresis thresholding:** Áp dụng ngưỡng hai cấp độ để xác định biên cạnh và loại bỏ các biên cạnh yếu không liên quan.

Canny filter tạo ra các biên cạnh rõ ràng và chính xác trong hình ảnh và thường được sử dụng trong nhiều ứng dụng thị giác máy tính và xử lý ảnh.

## 2.2 Hiện thực với python

Toàn bộ source code có thể tìm thấy tại đây: [Google colab](#)

### 2.2.1 Low-pass filter

Trước khi thực hiện Low-pass filter, vì chức năng chính của nó là cải thiện nhiễu của ảnh, nên ta sẽ tiến hành thêm một vài loại vào ảnh gốc, ví dụ như salt and pepper, gaussian and impulse.

```
import cv2
import numpy as np
import albumentations as albu

def add_gaussian_noise(image, mean, std=25):
    transform = albu.GaussNoise(var_limit=(1000,10000),mean=100,p=1)
    noisy_image = transform(image=image)['image']
    return noisy_image

def add_salt_and_pepper_noise(img, amount):
    thresh = amount
    numRows, numCols, _ = img.shape
    noise = np.random.rand(numRows,numCols)
    # Salt noise
    salt = noise < thresh
    image[salt] = 255
    # Pepper noise
    pepper = noise > 1 - thresh
    image[pepper] = 0
    return image

def add_impulse_noise(img, amount):
```

```
# Generate random noise
thresh = amount
numRows, numCols, _ = img.shape
noise = np.random.rand(numRows,numCols)
# pepper noise
im = noise < thresh
image[im] = 255
return image
```



Hình 2.1: Ảnh sau khi đã được thêm noise

### 2.2.1.1 Average filter

Hiện thực với công thức từ lý thuyết:

```
def average_blur(image, kernel_size):
    # Calculate the padding size
    pad_size = kernel_size // 2

    # Pad the image
    padded_image = np.pad(image, pad_size, mode='edge')

    # Initialize the blurred image
    blurred_image = np.zeros_like(image)

    # Apply average blur
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded_image[i:i+kernel_size, j:j+kernel_size]
            average_value = np.mean(window)
            blurred_image[i, j] = average_value
    return blurred_image
```

```
def average.blur_color(image, kernel_size):
    # Initialize the blurred image
    blurred_image = np.zeros_like(image)

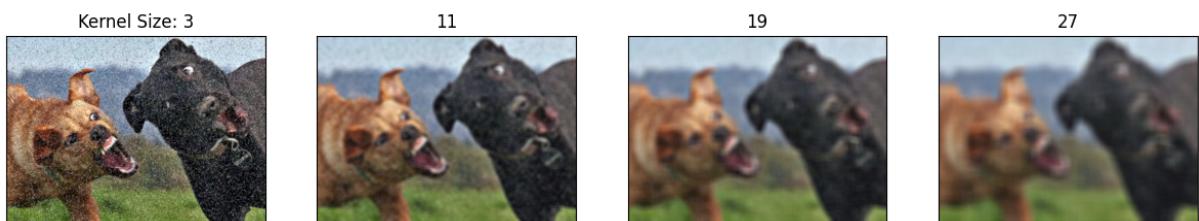
    # Apply average blur to each color channel separately
    for i in range(image.shape[2]):
        blurred_image[:, :, i] = average.blur(image[:, :, i], kernel_size)

    return blurred_image
```

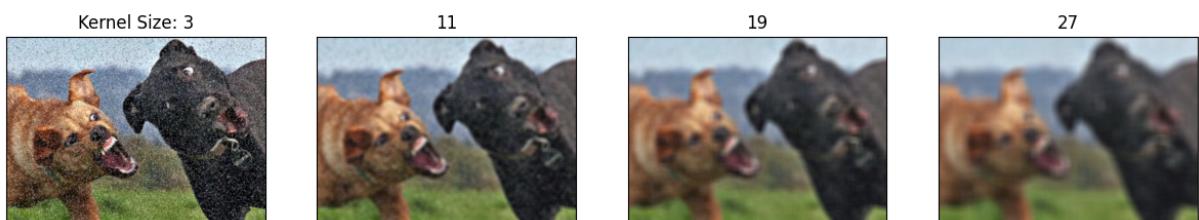
Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



**Hình 2.2:** Average filter with Gaussian noise image



**Hình 2.3:** Average filter with Salt and pepper noise image



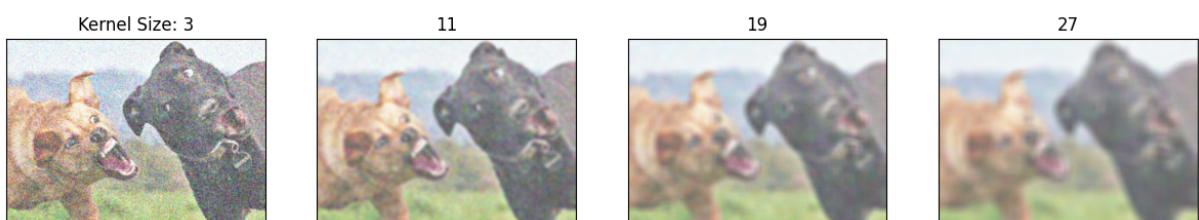
**Hình 2.4:** Average filter with Impulse noise image

Hiện thực bằng hàm có sẵn trong thư viện cv2

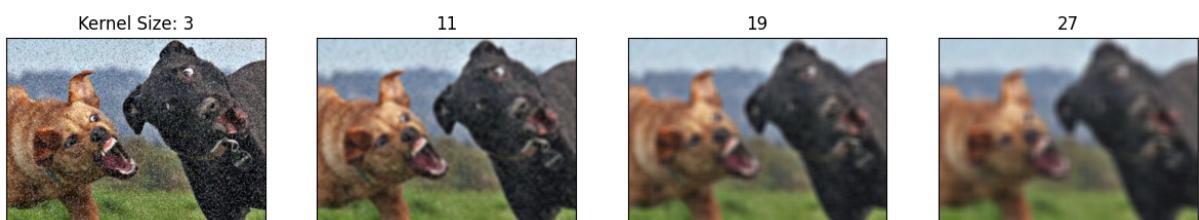
```
def average.blur_color_cv2(image, kernel_size):
    blurred_image = cv2.blur(image, (kernel_size, kernel_size))

    return blurred_image
```

Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



**Hình 2.5:** Average filter with Gaussian noise image



Hình 2.6: Average filter with Salt and pepper noise image



Hình 2.7: Average filter with Impulse noise image

### 2.2.1.2 Median filter

Hiện thực với công thức từ lý thuyết:

```
def median_blur(image, kernel_size):
    # Calculate the padding size
    pad_size = kernel_size // 2

    # Pad the image
    padded_image = np.pad(image, pad_size, mode='edge')

    # Initialize the blurred image
    blurred_image = np.zeros_like(image)

    # Apply median blur
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Extract the kernel window
            window = padded_image[i:i+kernel_size, j:j+kernel_size]
            # Calculate the median value
            median_value = np.median(window)
            # Set the corresponding pixel in the blurred image
            blurred_image[i, j] = median_value

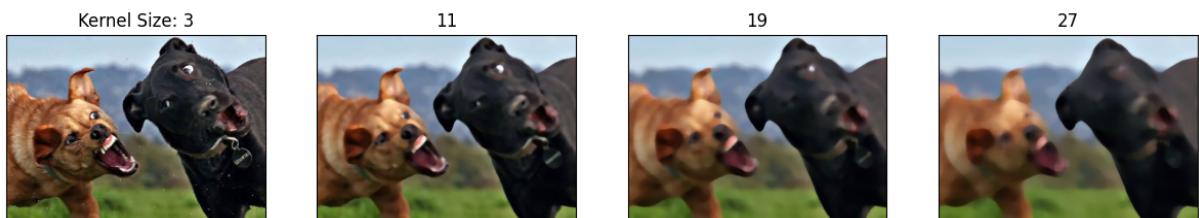
    return blurred_image
def median_blur_color(image, kernel_size):
    blurred_image = np.zeros_like(image)

    # Apply median blur to each color channel separately
    for i in range(image.shape[2]):
        blurred_image[:, :, i] = median_blur(image[:, :, i], kernel_size)
    return blurred_image
```

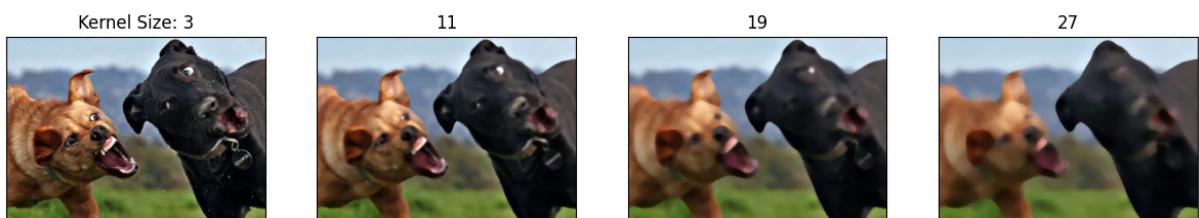
Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



**Hình 2.8:** Median filter with Gaussian noise image



**Hình 2.9:** Median filter with Salt and pepper noise image



**Hình 2.10:** Median filter with Impulse noise image

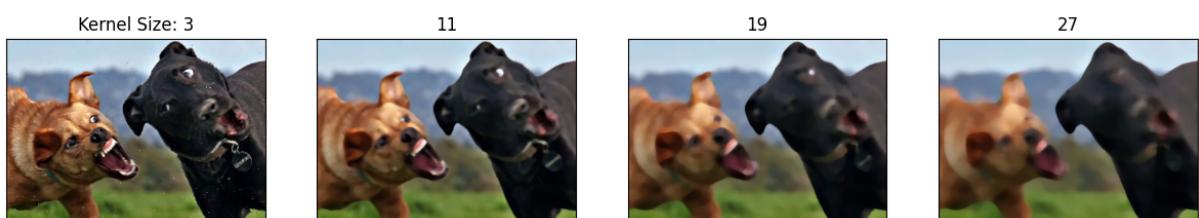
Hiện thực bằng hàm có sẵn trong thư viện cv2

```
def median_blur_color_cv2(image, kernel_size):
    blurred_image = cv2.medianBlur(image, kernel_size)
    return blurred_image
```

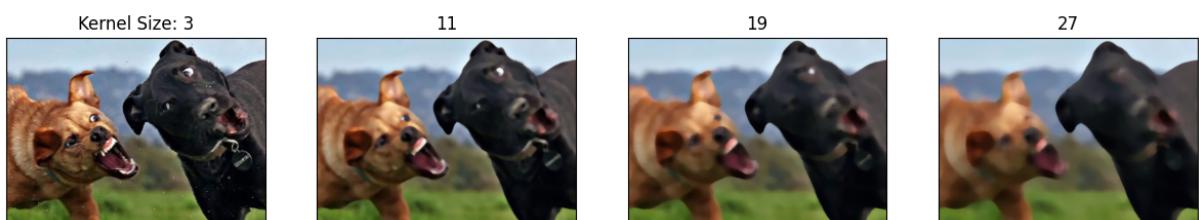
Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



**Hình 2.11:** Median filter with Gaussian noise image



**Hình 2.12:** Median filter with Salt and pepper noise image



Hình 2.13: Median filter with Impulse noise image

### 2.2.1.3 Gaussian filter

Hiện thực với công thức từ lý thuyết:

```
def median_blur(image, kernel_size):
    # Calculate the padding size
    pad_size = kernel_size // 2

    # Pad the image
    padded_image = np.pad(image, pad_size, mode='edge')

    # Initialize the blurred image
    blurred_image = np.zeros_like(image)

    # Apply median blur
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Extract the kernel window
            window = padded_image[i:i+kernel_size, j:j+kernel_size]
            # Calculate the median value
            median_value = np.median(window)
            # Set the corresponding pixel in the blurred image
            blurred_image[i, j] = median_value

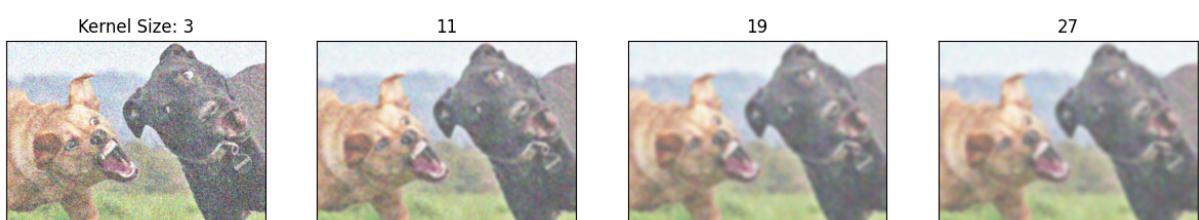
    return blurred_image

def median_blur_color(image, kernel_size):
    blurred_image = np.zeros_like(image)

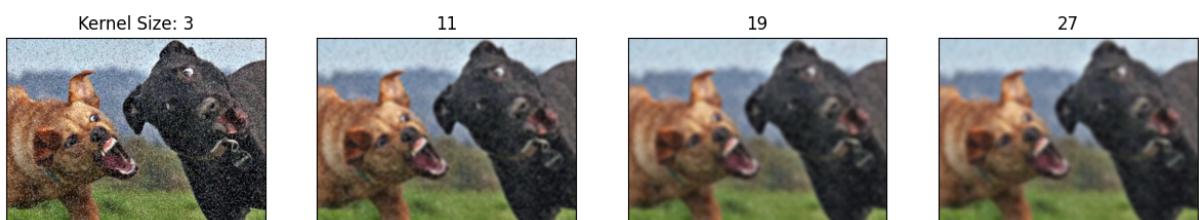
    # Apply median blur to each color channel separately
    for i in range(image.shape[2]):
        blurred_image[:, :, i] = median_blur(image[:, :, i], kernel_size)

    return blurred_image
```

Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



Hình 2.14: Gaussian filter with Gaussian noise image



Hình 2.15: Gaussian filter with Salt and pepper noise image

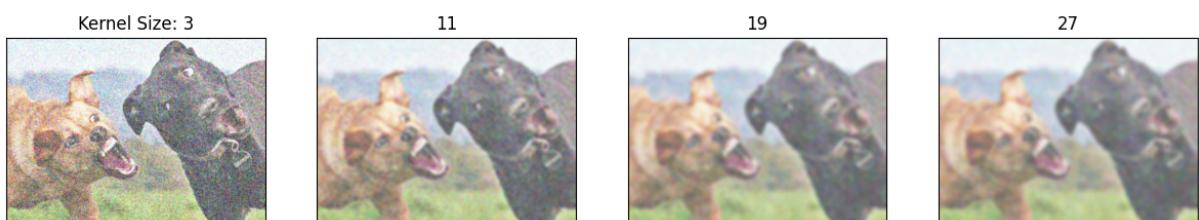


Hình 2.16: Gaussian filter with Impluse noise image

Hiện thực bằng hàm có sẵn trong thư viện cv2

```
def gaussian_blur(image, kernel_size):
    return cv2.GaussianBlur(image, (kernel_size, kernel_size), 5)
```

Kết quả với ba loại nhiễu khi áp dụng với kích thước kernel tăng dần.



Hình 2.17: Gaussian filter with Gaussian noise image



Hình 2.18: Gaussian filter with Salt and pepper noise image



Hình 2.19: Gaussian filter with Impluse noise image

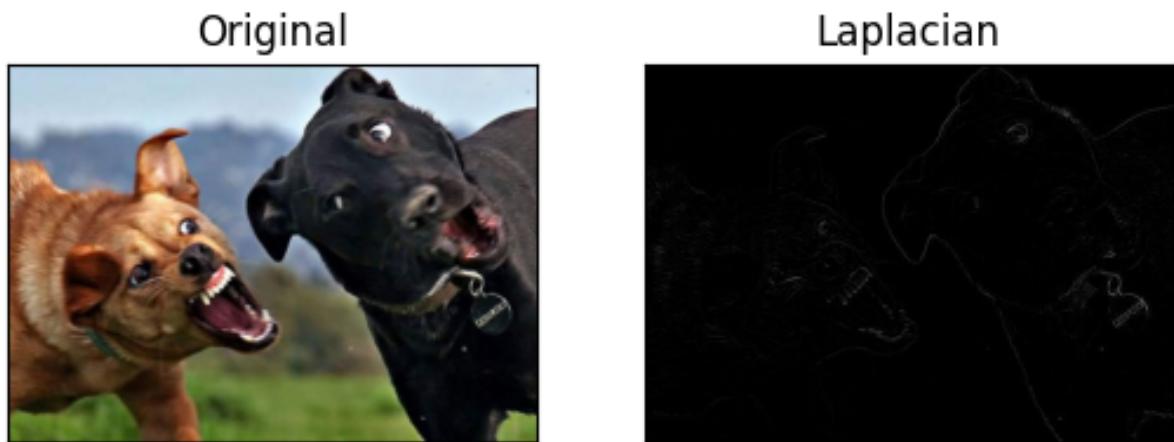
## 2.2.2 High-pass filter

### 2.2.2.1 Laplacian filter

Hiện thực với công thức từ lý thuyết:

```
def laplacian_filter_build(image):
    # Define the Laplacian kernel
    laplacian_kernel = np.array([[0, -1, 0],
                                [-1, 4, -1],
                                [0, -1, 0]])
    filtered_channels = [cv2.filter2D(channel, -1, laplacian_kernel) for
                         channel in cv2.split(image)]
    filtered_image = cv2.merge(filtered_channels)
    return filtered_image
```

Kết quả:



**Hình 2.20:** Laplacian filter by hand

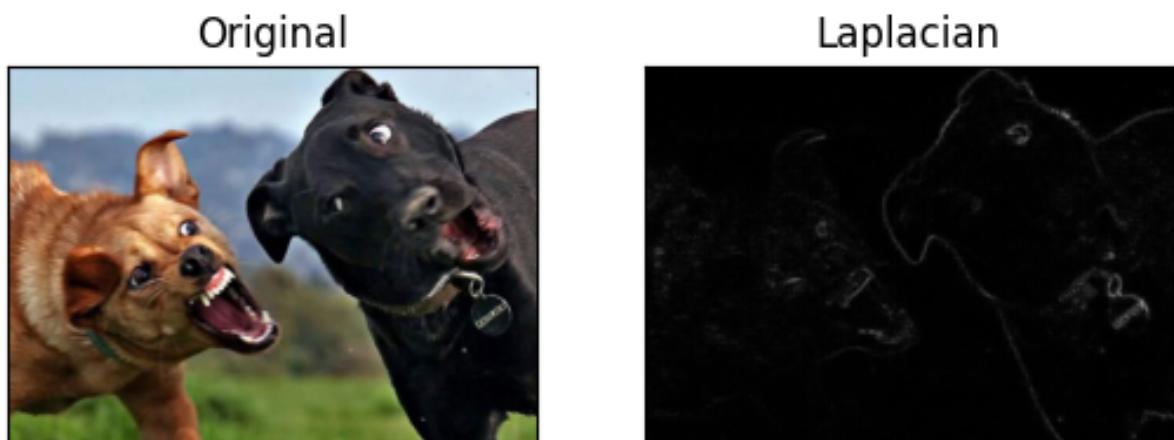
Hiện thực bằng hàm có sẵn trong thư viện cv2

```
def laplacian_filter(image):
    # Convert the image to grayscale if it's in color
    if len(image.shape) == 3:
        grayscale_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        grayscale_image = image

    # Apply Laplacian filter using cv2.Laplacian()
    filtered_image = cv2.Laplacian(grayscale_image, cv2.CV_64F)

    # Convert the output back to uint8 and scale it to [0, 255]
    filtered_image = cv2.convertScaleAbs(filtered_image)
    return filtered_image
```

Kết quả:



Hình 2.21: Laplacian filter with cv2

### 2.2.2.2 Sobel filter

Hiện thực với công thức từ lý thuyết:

```
def sobel_filter(image):
    sobel_x = np.array([[-1, 0, 1],
                      [-2, 0, 2],
                      [-1, 0, 1]])

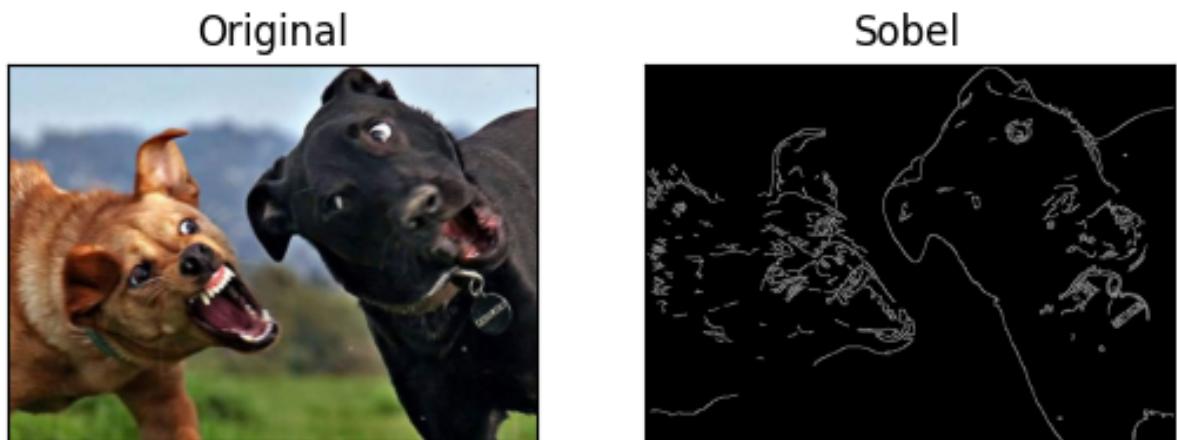
    sobel_y = np.array([[-1, -2, -1],
                      [0, 0, 0],
                      [1, 2, 1]])

    # Apply Sobel filter to compute gradient magnitudes in horizontal and \
    vertical directions
    gradient_x = cv2.filter2D(image, -1, sobel_x)
    gradient_y = cv2.filter2D(image, -1, sobel_y)

    # Compute the gradient magnitude
    gradient_magnitude = np.sqrt(np.square(gradient_x) + np.square(gradient_y))
    gradient_magnitude = gradient_magnitude.astype(np.float32)
    gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0,
                                       255, cv2.NORM_MINMAX)

    return filtered_image
```

Kết quả:



Hình 2.22: Sobel filter by hand

Hiện thực bằng hàm có sẵn trong thư viện cv2

```
def sobel_filter_cv2(image):
    if image is None or image.size == 0:
        raise ValueError("Input image is empty or invalid.")

    if len(image.shape) != 2:
        raise ValueError("Input image must be grayscale.")

    # Apply Sobel filter to compute gradient magnitudes in horizontal and \
    vertical directions
    gradient_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    gradient_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

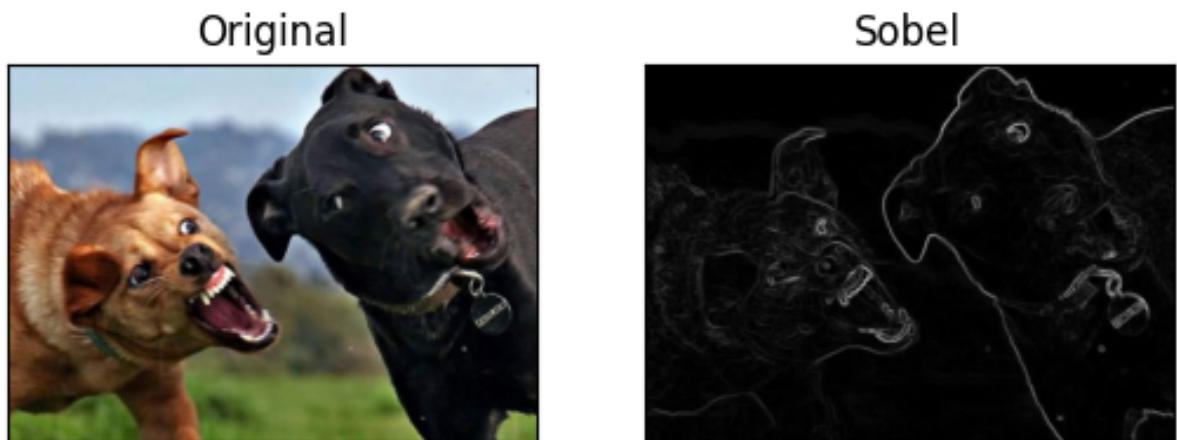
    # Compute the gradient magnitude
    gradient_magnitude = cv2.magnitude(gradient_x, gradient_y)

    # Normalize the gradient magnitude to [0, 255]
    gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0, 255,
                                       cv2.NORM_MINMAX)

    # Convert the data type to uint8
    filtered_image = np.uint8(gradient_magnitude)

    return filtered_image
```

Kết quả:



Hình 2.23: Sobel filter with cv2

### 2.2.2.3 Canny filter

Hiện thực với công thức từ lý thuyết:

```
def canny_edge_detector(image, min_threshold, max_threshold):
    # Apply Gaussian blur to the image to reduce noise
    blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

    # Compute gradients using the Sobel operator
    gradient_x = cv2.Sobel(blurred_image, cv2.CV_64F, 1, 0, ksize=3)
    gradient_y = cv2.Sobel(blurred_image, cv2.CV_64F, 0, 1, ksize=3)

    # Compute gradient magnitude and direction
    gradient_magnitude = cv2.magnitude(gradient_x, gradient_y)
    gradient_direction = cv2.phase(gradient_x, gradient_y)

    # Perform non-maximum suppression to thin edges
    gradient_magnitude_suppressed = cv2.dilate(cv2.erode(gradient_magnitude, None),
                                                None)

    # Apply double thresholding to identify potential edges
    edges = np.zeros_like(gradient_magnitude_suppressed)
    edges[(gradient_magnitude_suppressed >= min_threshold) &
          (gradient_magnitude_suppressed <= max_threshold)] = 255

    return edges

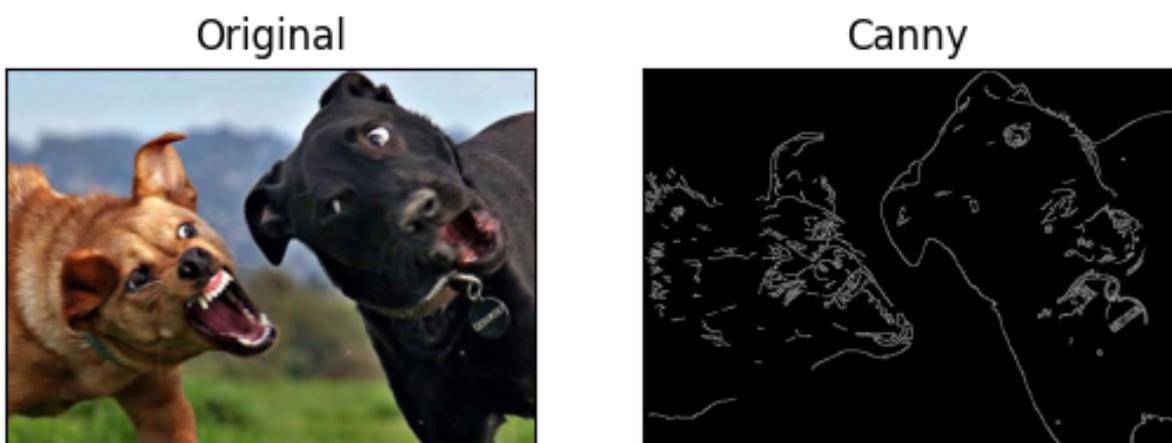
# Read the input image
image = np.array(img_list[1])
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Define the threshold values for Canny edge detector
min_threshold = 100
```

```
max_threshold = 200

# Apply Canny edge detector to the grayscale image
filtered_image = canny_edge_detector(gray_image, min_threshold, max_threshold)
```

Kết quả:



**Hình 2.24:** Canny filter by hand

Hiện thực bằng hàm có sẵn trong thư viện cv2

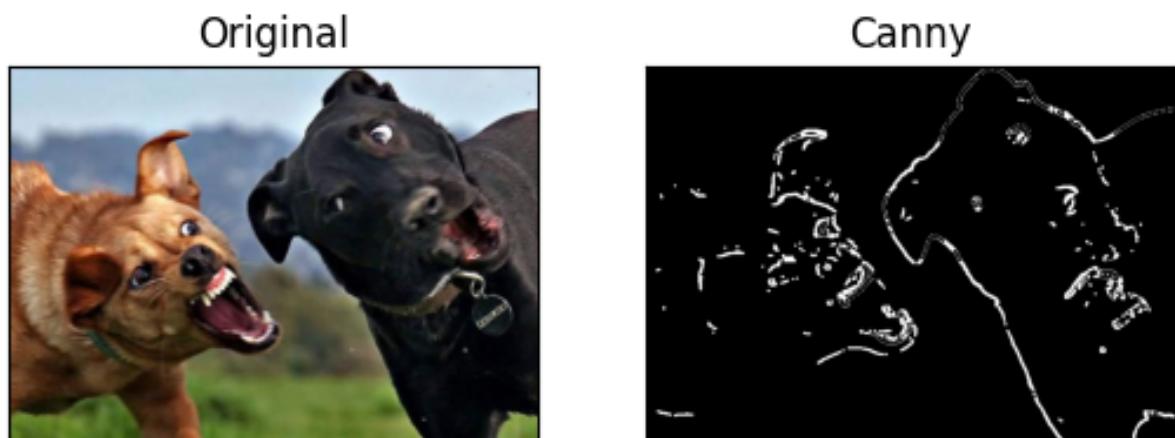
```
def canny_edge_detector(image, min_threshold, max_threshold):
    edges = cv2.Canny(image, min_threshold, max_threshold)
    return edges

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Define the threshold values for Canny edge detector
min_threshold = 100
max_threshold = 200

# Apply Canny edge detector to the grayscale image
filtered_image = canny_edge_detector(gray_image, min_threshold, max_threshold)
```

Kết quả:



Hình 2.25: Canny filter with cv2

## Chương 3

# Kết quả đạt được và tổng kết

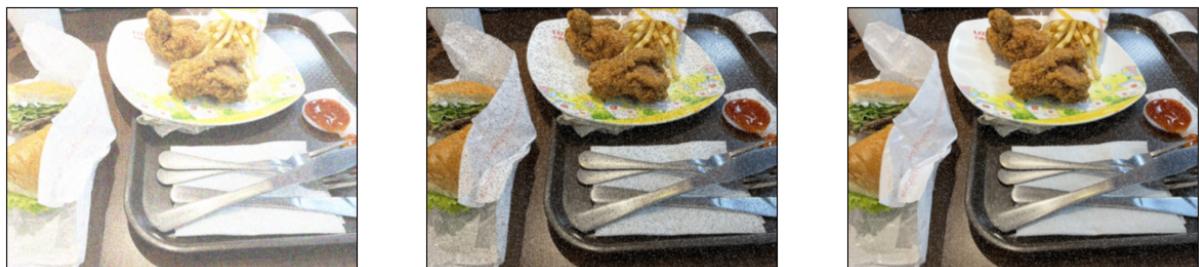
### 3.1 Thủ với những hình ảnh khác

#### 3.1.1 Low-pass filter

##### 3.1.1.1 Ảnh 1:



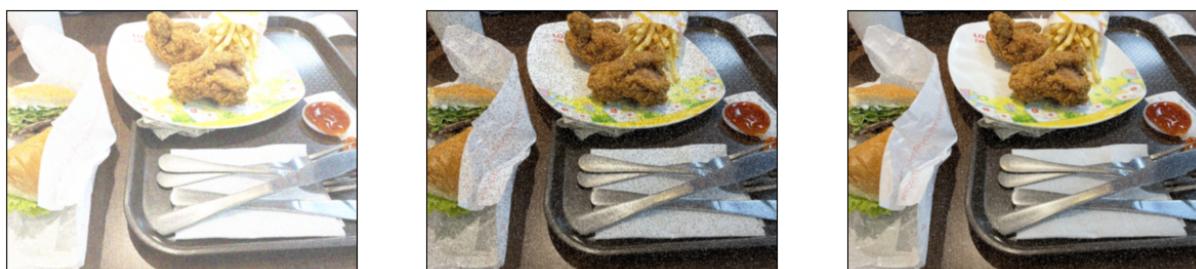
Hình 3.1: Origin noise image



Hình 3.2: Average filter with kernel size = 5



Hình 3.3: Median filter with kernel size = 5



Hình 3.4: Gaussian filter with kernel size = 5

### 3.1.1.2 Ảnh 2:



Hình 3.5: Origin noise image



Hình 3.6: Average filter with kernel size = 5



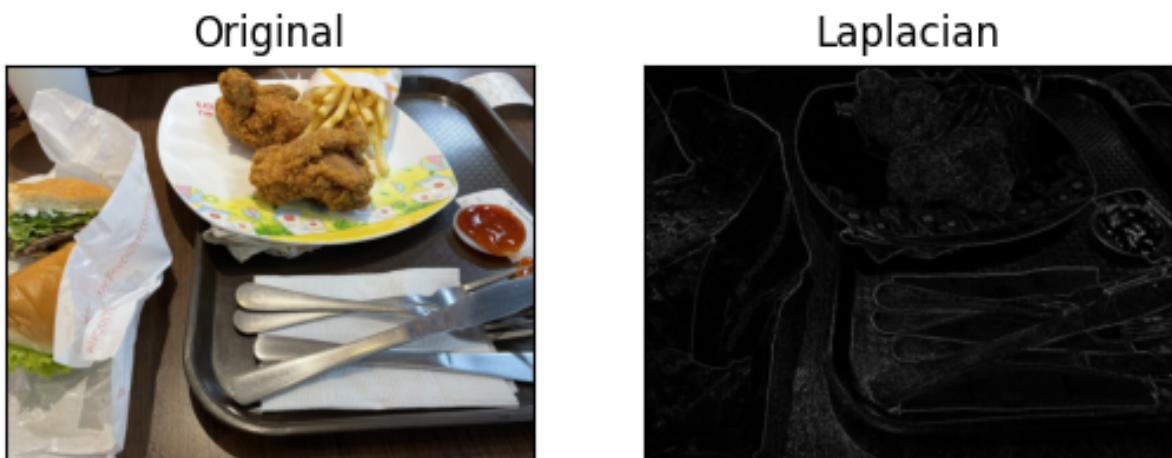
Hình 3.7: Median filter with kernel size = 5



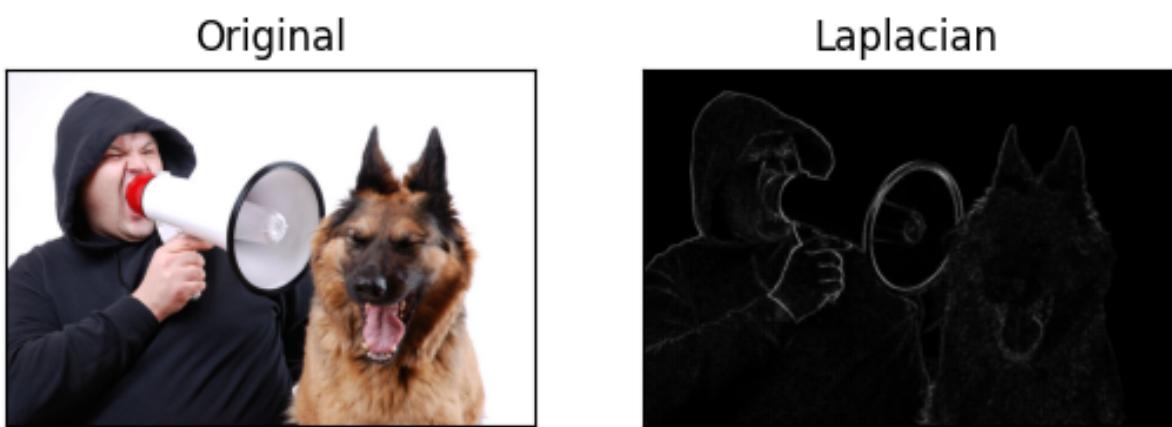
Hình 3.8: Gaussian filter with kernel size = 5

### 3.1.2 High-pass filter

#### 3.1.2.1 Laplacian filter

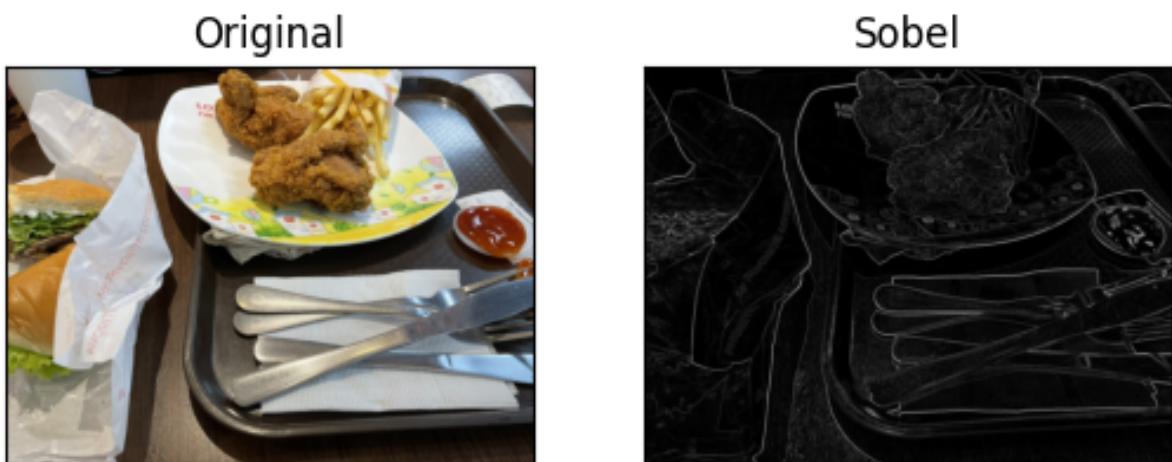


Hình 3.9: Laplacian filter

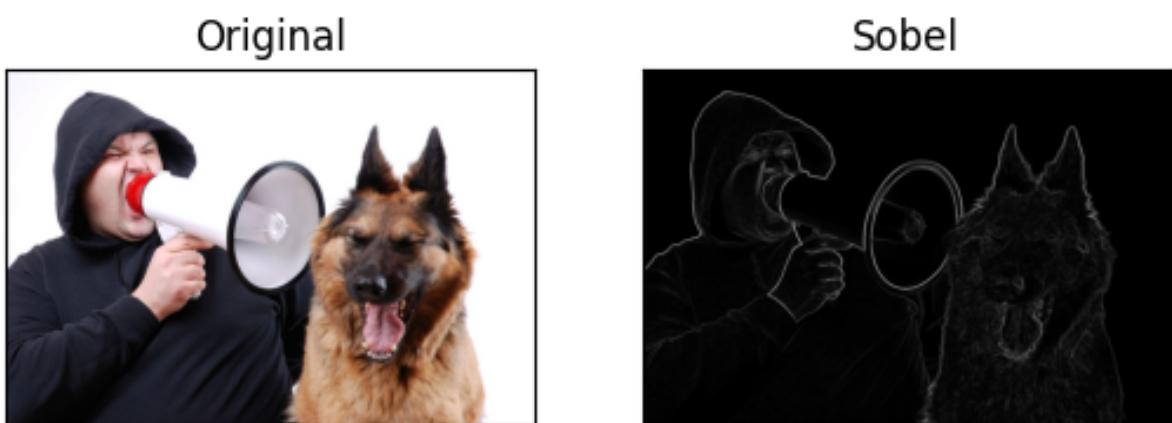


Hình 3.10: Laplacian filter

#### 3.1.2.2 Sobel filter

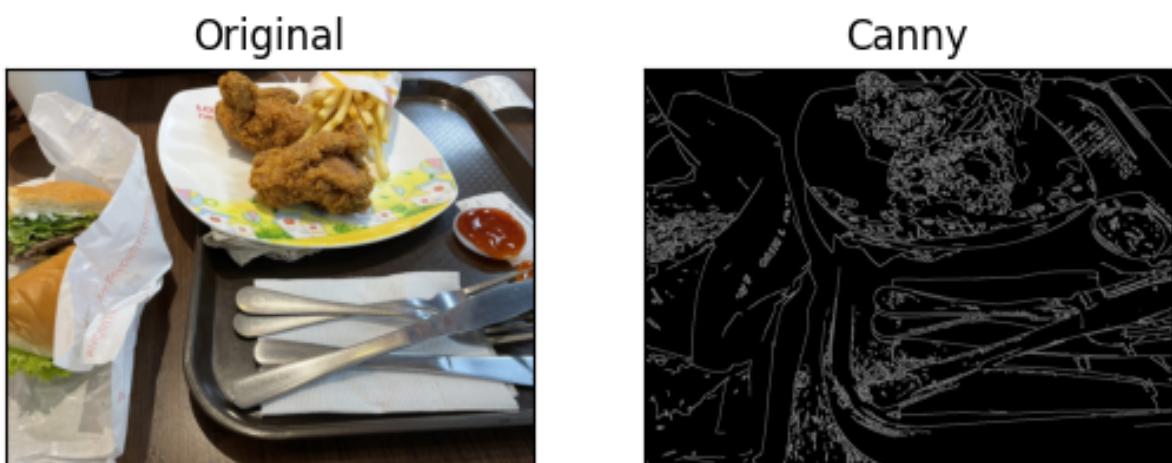


Hình 3.11: Sobel filter

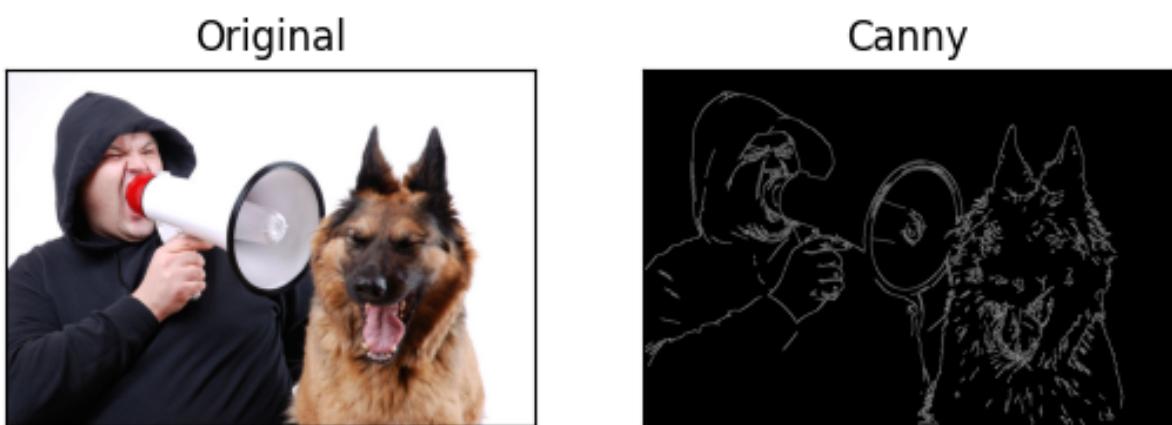


Hình 3.12: Sobel filter

### 3.1.2.3 Canny filter



Hình 3.13: Canny filter



Hình 3.14: Canny filter

## 3.2 Kết quả đạt được

Vậy là chúng ta đã có một cái nhìn tổng thể và hiểu sâu sắc hơn cách hoạt động cũng như kết quả trả về của từng loại filter. Sau đây là phần kết quả đạt được và nhận xét sau khi thực hiện các low-pass filter (average, median, gaussian) và high-pass filter (laplacian, sobel, canny)



### 3.2.1 Low-pass Filters

#### 1. Average Filter:

Khi áp dụng Average Filter, chúng ta đã quan sát thấy rằng ảnh đã được làm mịn một cách đồng đều và các điểm nhiễu đã được giảm bớt. Tuy nhiên, phương pháp này không giữ được các chi tiết cạnh trong ảnh.

#### 2. Median Filter:

Median Filter đã cho hiệu quả tốt trong việc loại bỏ nhiễu loại muối và tiêu từ ảnh, đồng thời giữ nguyên các chi tiết cạnh và các đặc điểm quan trọng khác.

#### 3. Gaussian Filter:

Gaussian Filter làm mịn ảnh một cách tự nhiên hơn so với average filter, giữ được chi tiết cạnh trong ảnh và làm giảm nhiễu một cách hiệu quả.

### 3.2.2 High-pass Filters

#### 1. Laplacian Filter:

Kết quả của Laplacian Filter đã phát hiện được các biên cạnh và chi tiết trong ảnh một cách nhanh chóng. Tuy nhiên, một số nhiễu cũng được tạo ra trong quá trình này.

#### 2. Sobel Filter:

Sobel Filter đã hiệu quả trong việc phát hiện biên cạnh và có thể được điều chỉnh để phát hiện theo các hướng khác nhau, giúp tạo ra các kết quả linh hoạt.

#### 3. Canny Filter:

Canny Filter đã tạo ra các biên cạnh sắc nét và chính xác, loại bỏ nhiễu và giả mạo cạnh một cách hiệu quả.

## 3.3 Tổng kết

Sau bài tập lớn lần này, chúng ta đã học được cách sử dụng kernel trong việc lọc ảnh thông qua các công thức liên quan cũng như cách thức mà việc trượt kernel có thể biến đổi từ bức ảnh gốc thành bức ảnh đã được lọc. Qua đó, chúng ta có thể rút ra được một số bài học:

- Việc sử dụng các low-pass filter làm mịn ảnh giúp loại bỏ nhiễu và làm tăng chất lượng hình ảnh. Tuy nhiên khi tăng kích thước kernel lên quá lớn, ảnh sẽ bị mờ đi đáng kể.
- Các high-pass filter giúp phát hiện các biên cạnh và chi tiết trong ảnh một cách hiệu quả.
- Tùy thuộc vào nhu cầu và đặc điểm của ảnh, có thể lựa chọn các filter phù hợp để đạt được kết quả mong muốn.
- Việc tự xây dựng kernel cho các filter cơ bản giúp hiểu rõ hơn về nguyên lý hoạt động của chúng.
- Sử dụng các hàm có sẵn trong thư viện OpenCV giúp tiết kiệm thời gian và tối ưu hóa hiệu suất của quá trình xử lý ảnh.
- Không có sự khác biệt nhiều trong việc tự xây dựng các kernel so với sử dụng các thư viện có sẵn trong cv2.



### 3.4 Tài liệu tham khảo

1. Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing* (4th Edition). Pearson, 2022.
2. Sonka, Milan, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision* (4th Edition). Cengage Learning, 2019.
3. Szeliski, Richard. *Computer Vision: Algorithms and Applications* (3rd Edition). Springer, 2011.
4. Szeliski, Richard. "Computer Vision: Algorithms and Applications." Springer Science & Business Media, 2010.
5. Burger, Wilhelm, and Mark J. Burge. "Digital Image Processing: An Algorithmic Approach with MATLAB." Springer Science & Business Media, 2016.
6. Gonzales, Rafael C., and Richard E. Woods. "Digital Image Processing." Pearson Education, Inc., 2018.



### 3.5 Phụ lục