2.1 Lọc ảnh

Lọc ảnh là một thuật ngữ rộng được áp dụng cho nhiều kỹ thuật xử lý ảnh nhằm tăng cường chất lượng ảnh bằng cách loại bỏ các đặc điểm không mong muốn (ví dụ như nhiễu) hoặc cải thiện các đặc điểm cần thiết (ví dụ như tăng độ tương phản). Làm mờ, phát hiện cạnh, làm sắc nét cạnh và loại bỏ nhiễu đều là các ví dụ của việc lọc ảnh.
Lọc ảnh là một thao tác dựa trên vùng lân cận (hoặc cục bộ). Điều này có nghĩa là giá trị của điểm ảnh tại vị trí (x, y) trong ảnh đầu ra phụ thuộc vào các điểm ảnh trong vùng lân cận nhỏ xung quanh vị trí (x, y) trong ảnh đầu vào. Ví dụ, lọc ảnh sử dụng một kernel 3x3 (tương đương 2D của phản hồi xung) sẽ khiến điểm ảnh đầu ra tại vị trí (x, y) phụ thuộc vào các điểm ảnh đầu vào tại vị trí (x, y) và 8 điểm lân cận của nó, như được hiển thị trong Hình 2.

Hình 2. Ví dụ về ảnh đầu vào sẽ được lọc với kernel 3x3. Một điểm ảnh đầu ra sẽ là sự kết hợp trọng số của điểm ảnh cùng vị trí (màu vàng) và các giá trị lân cận (màu xanh). Trọng số sẽ được xác định trong kernel.
Khi điểm ảnh đầu ra chỉ phụ thuộc vào sự kết hợp tuyến tính của các điểm ảnh đầu vào, chúng ta gọi bộ lọc đó là bộ lọc tuyến tính. Ngược lại, nó được gọi là bộ lọc phi tuyến tính.

197	198	203	205	199	145	75	58	55	53
201	203	205	205	198	136	84	108	81	54
203	205	205	205	202	178	168	179	103	56
204	205	205	206	206	203	205	191	103	56
204	205	205	204	205	204	205	191	102	56
203	201	202	203	203	203	205	191	103	56
203	201	202	203	203	203	206	192	105	58
204	202	203	203	203	204	205	199	150	123
205	202	203	203	203	204	206	206	200	198
203	203	203	203	204	204	206	206	207	207

2.1.1 Phép tích chập

Bộ lọc tuyến tính được thực hiện bằng cách sử dụng phép tích chập 2D. Các thao tác như làm mờ và phát hiện đường viền được thực hiện bằng cách sử dụng phép tích chập. Hãy xem xét một kernel 3x3:

$$h = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

chúng ta sẽ sử dụng để lọc điểm ảnh trung tâm màu vàng trong Hình 2. Giá trị điểm ảnh đầu ra được tính bằng cách nhân các phần tử tương ứng của mảng màu xanh & vàng đầu vào và kernel tích chập, sau đó cộng tất cả các tích lại: $outcenter$ = 206 × (-1) + 203 × 0 + 205 × 1 + 205 × (-2) + 204 × 0 + 205 × 2 + 203 × (-1) + 203 × 0 + 205 × 1 = 1

Để lọc toàn bộ ảnh, quá trình này được lặp lại điểm ảnh theo điểm ảnh (kernel trượt qua ảnh từ trái sang phải, từ trên xuống dưới). Đối với ảnh màu, phép tích chập được thực hiện độc lập trên mỗi kênh màu. Tại các biên, phép tích chập không được xác định rõ ràng. Có một số tùy chọn để chọn:

- Bỏ qua các điểm ảnh biên: bằng cách loại bỏ các điểm ảnh biên, ảnh đầu ra sẽ nhỏ hơn một chút so với ảnh đầu vào.

- Điền thêm số 0: ảnh đầu vào được đệm bằng các số 0 tại các điểm ảnh biên để làm nó lớn hơn, sau đó thực hiện phép tích chập.

- Nhân bản biên: một tùy chọn khác là nhân bản các điểm ảnh biên của ảnh đầu vào, sau đó thực hiện phép tích chập trên ảnh lớn hơn này.

- Phản chiếu biên: tùy chọn ưu tiên là phản chiếu biên quanh vùng biên. Phản chiếu giúp đảm bảo sự chuyển tiếp mượt mà về cường độ của các điểm ảnh tại biên.

Trong OpenCV, phép tích chập được thực hiện bằng cách sử dụng hàm filter2D. Cú pháp của hàm là:

Trong OpenCV, phép tích chập được thực hiện bằng cách sử dụng hàm filter2D. Cú pháp của hàm là:

- `dst = cv2.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])`

- Với các tham số:

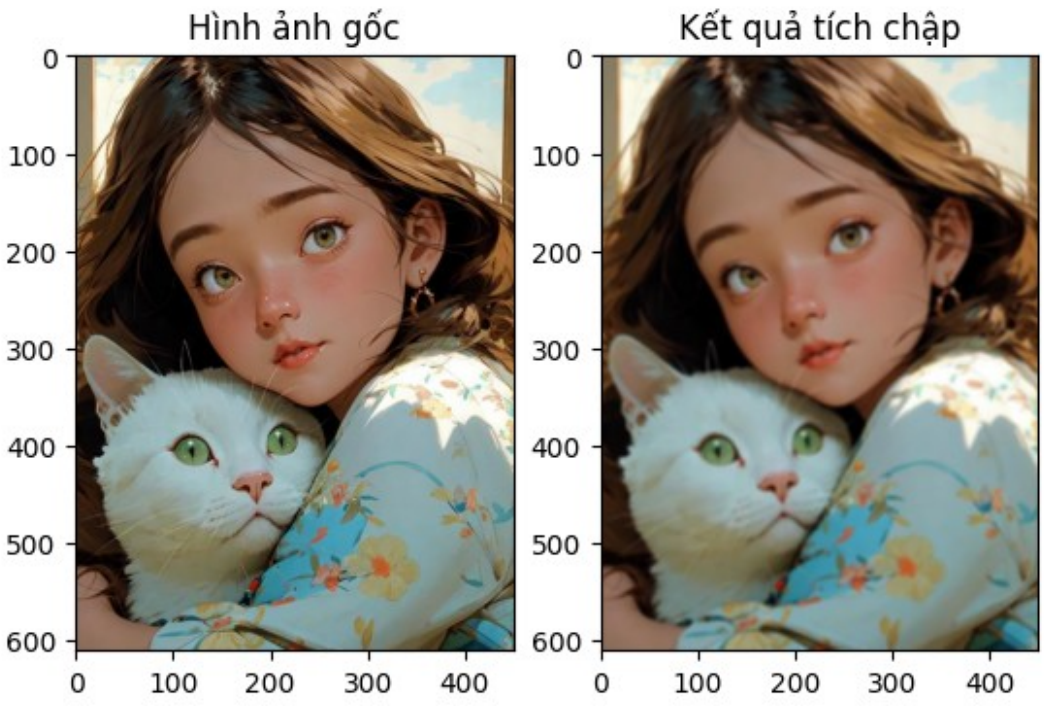
  - `src`: hình ảnh đầu vào.
  - `dst`: hình ảnh đầu ra có cùng kích thước và cùng số kênh như `src`.
  - `ddepth`: độ sâu mong muốn của hình ảnh đích.
  - `kernel`: kernel tích chập, một ma trận số thực đơn kênh; nếu bạn muốn áp dụng các kernel khác nhau cho các kênh khác nhau, hãy tách hình ảnh thành các mặt phẳng màu riêng biệt bằng cách sử - dụng `split` và xử lý chúng riêng lẻ.
  - `anchor`: điểm neo của kernel, cho biết vị trí tương đối của điểm được lọc trong kernel; điểm neo nên nằm trong kernel; giá trị mặc định `(-1, -1)` nghĩa là điểm neo ở trung tâm kernel.
  - `delta`: một giá trị tùy chọn được thêm vào các pixel đã lọc trước khi lưu chúng vào `dst`.
  - `borderType`: phương pháp ngoại suy pixel.

Các tham số tùy chọn như điểm neo (`anchor`), `delta` và `borderType` hầu như không bao giờ được thay đổi khỏi các giá trị mặc định của chúng.

Đối với hàm `cv2.filter2D` trong tài liệu OpenCV, thuật ngữ tích chập (convolution) bị sử dụng sai thay vì phép tương quan chéo (cross-correlation). Thực tế, hàm này thực hiện phép tương quan chéo, và nó chỉ giống với tích chập khi kernel bộ lọc là đối xứng!

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread(r'D:\Image-Proccessing\pratice\week_1\img\
00750.jpeg')
if image is None:  # Kiểm tra nếu tệp không tồn tại
    print("Could not open the image")

ker_size = 5
# Bộ lọc hộp: kernel 5x5 với tổng tất cả các phần tử bằng 1
kernel = np.ones((ker_size, ker_size), dtype=np.float32) / ker_size**2
print(kernel)

result = cv2.filter2D(image, -1, kernel, (-1, -1), delta=0,
borderType=cv2.BORDER_DEFAULT)

plt.figure()
plt.subplot(121)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Hình ảnh gốc")
plt.subplot(122)
plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
plt.title("Kết quả tích chập")
plt.show()
```

```
[[0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]]
```



Tham số thứ hai của hàm `cv2.filter2D` (độ sâu - `depth`) được đặt là `-1`, nghĩa là độ sâu bit của hình ảnh đầu ra sẽ giống với hình ảnh đầu vào. Vì vậy, nếu hình ảnh đầu vào thuộc kiểu `uint8`, thì hình ảnh đầu ra cũng sẽ có cùng kiểu dữ liệu.

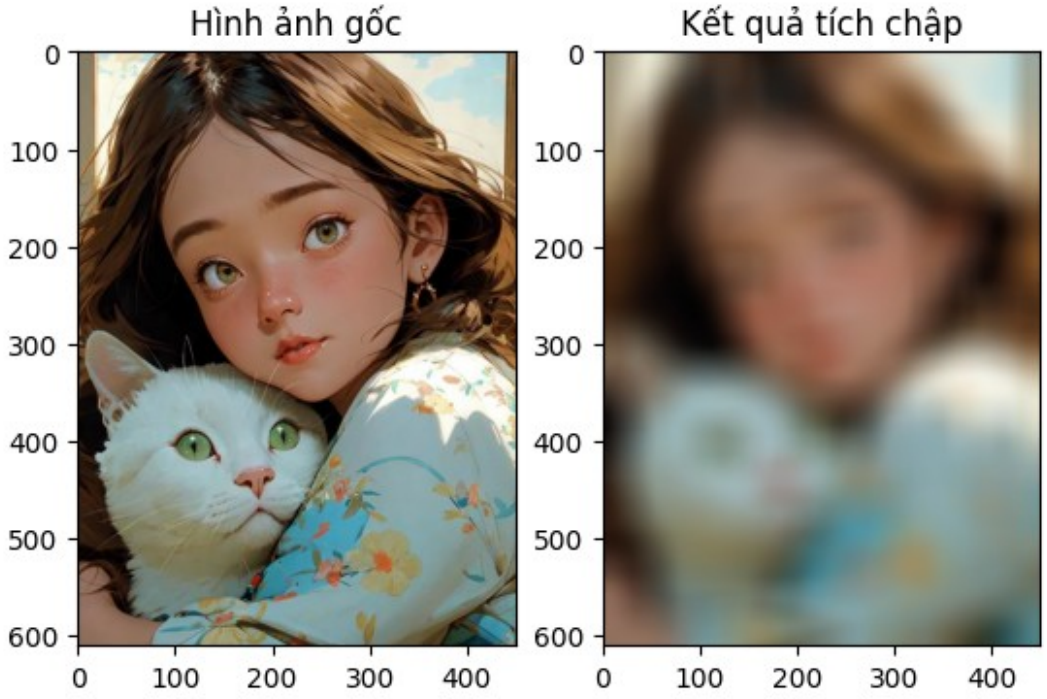
Bài tập 2.1: Thay đổi kernel đã sử dụng trong ví dụ trước và phân tích kết quả

Thử sử dụng một kernel mà không cần chuẩn hóa các giá trị bằng số lượng phần tử trong kernel. Sự thay đổi này ảnh hưởng như thế nào đến độ sáng của hình ảnh đầu ra so với hình ảnh đầu vào?

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread(r'D:\Image-Proccessing\pratice\week_1\img\
00750.jpeg')
if image is None:  # Kiểm tra nếu tệp không tồn tại
    print("Could not open the image")

ker_size = 50
# Bộ lọc hộp: kernel 5x5 với tổng tất cả các phần tử bằng 1
kernel = np.ones((ker_size, ker_size), dtype=np.float32) / ker_size**2
print(kernel)

result = cv2.filter2D(image, -1, kernel, (-1, -1), delta=0,
borderType=cv2.BORDER_DEFAULT)
```

```
plt.figure()
plt.subplot(121)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Hình ảnh gốc")
plt.subplot(122)
plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
plt.title("Kết quả tích chập")
plt.show()
```

```
[[0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]
 [0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]
 [0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]
 ...
 [0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]
 [0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]
 [0.0004 0.0004 0.0004 ... 0.0004 0.0004 0.0004]]
```



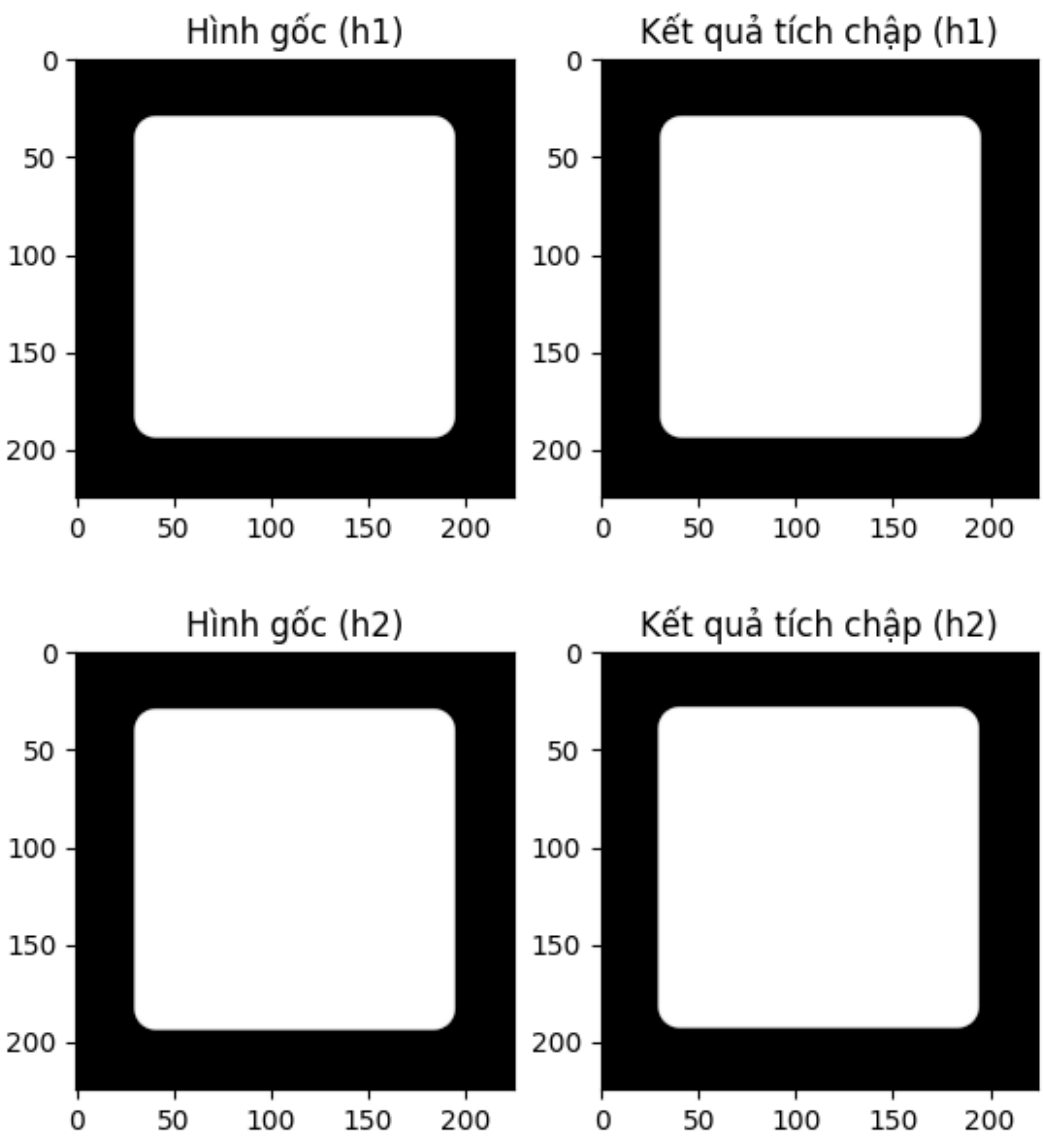
Phân tích kết quả:

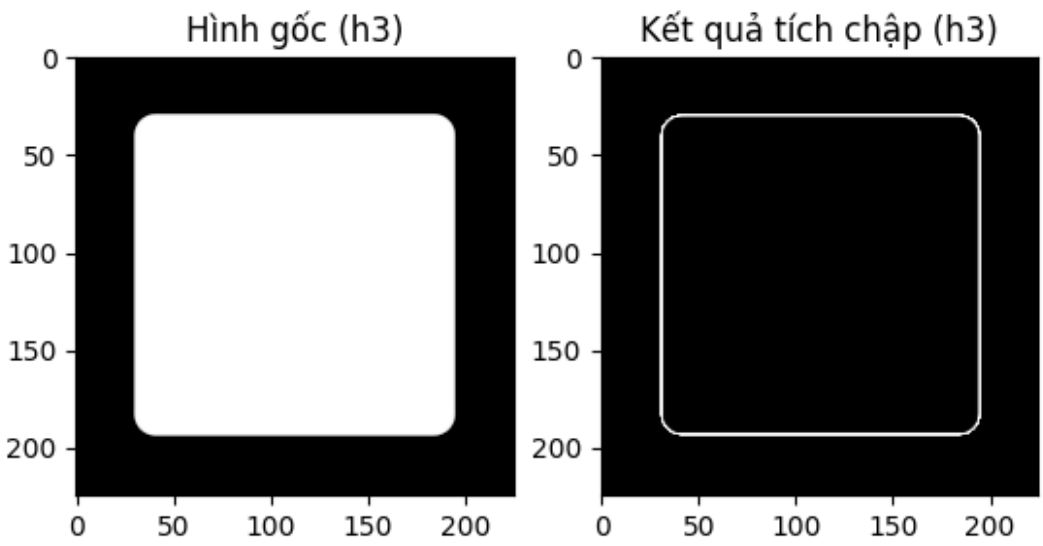
- Khi không chuẩn hóa các giá trị trong kernel, các giá trị pixel được nhân với các phần tử của kernel và sau đó được cộng lại, dẫn đến tăng độ sáng tổng thể của hình ảnh đầu ra. Điều này là do các giá trị trong kernel chưa được chia cho số lượng phần tử của nó, làm cho giá trị trung bình của các pixel cao hơn nhiều so với giá trị đầu vào.
- So sánh với hình ảnh gốc: Hình ảnh đầu ra có thể xuất hiện sáng hơn hoặc thậm chí bị "cháy sáng" (overexposed) ở một số vùng nếu các giá trị pixel vượt quá phạm vi dữ liệu (thường là từ 0 đến 255 cho hình ảnh loại `uint8`).

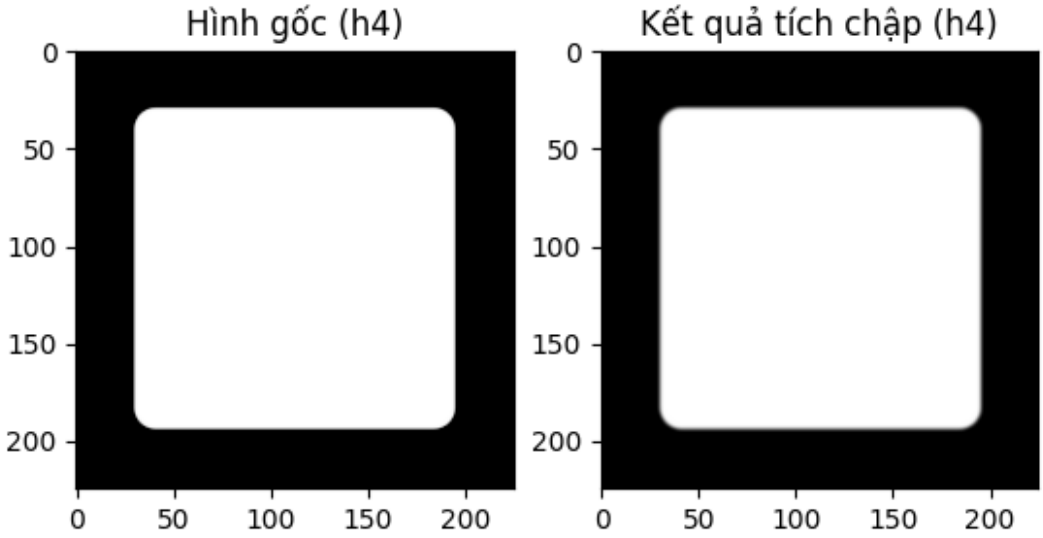
- Bài học rút ra: Việc chuẩn hóa kernel giúp duy trì độ sáng tương đối của hình ảnh, trong khi việc không chuẩn hóa có thể làm thay đổi độ sáng và độ tương phản của kết quả đầu ra.

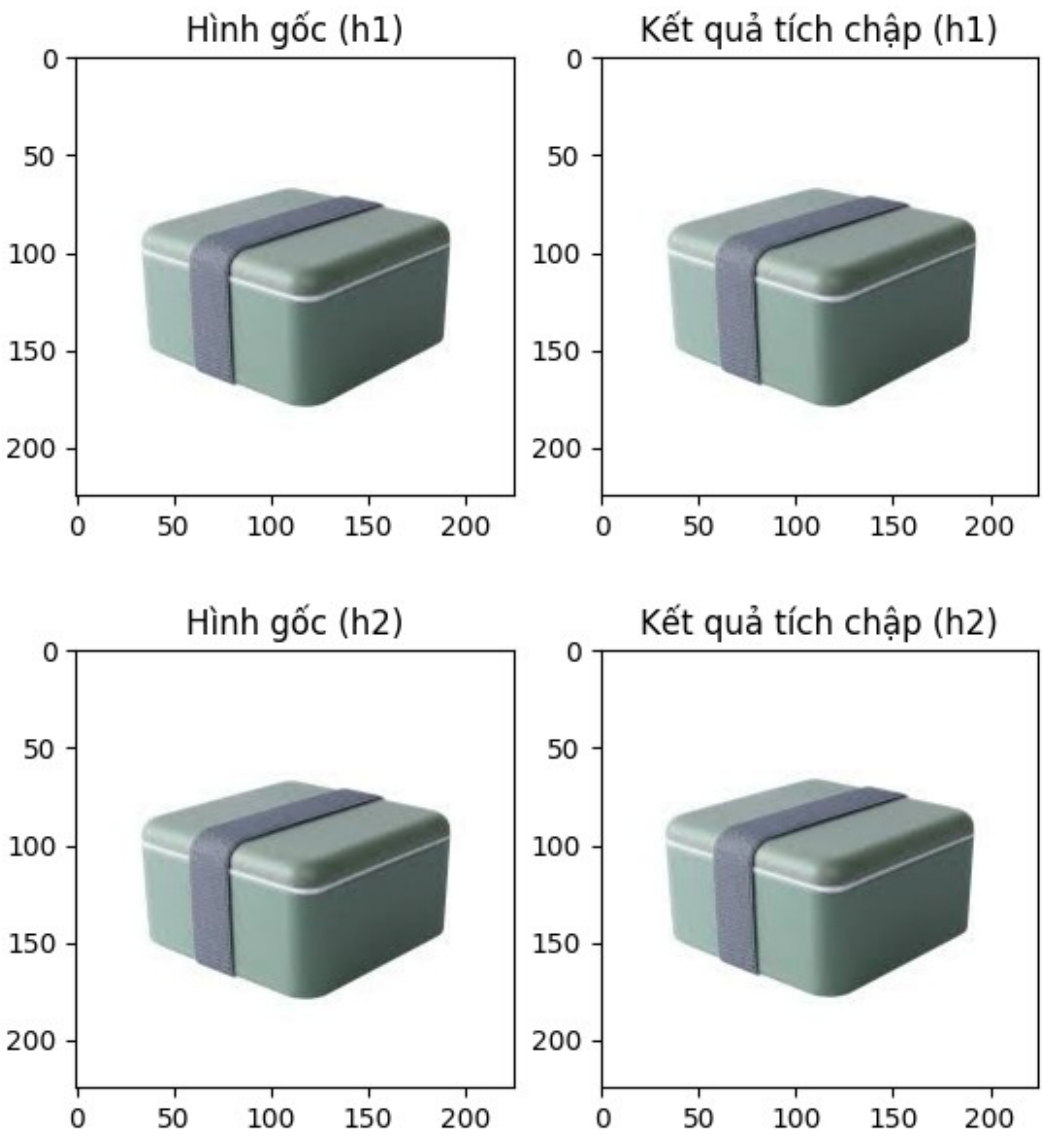
Bài tập 2.2: Thử nghiệm các kernel sau trên hình ảnh đầu vào "white_square.jpg" và "hop.jpg"

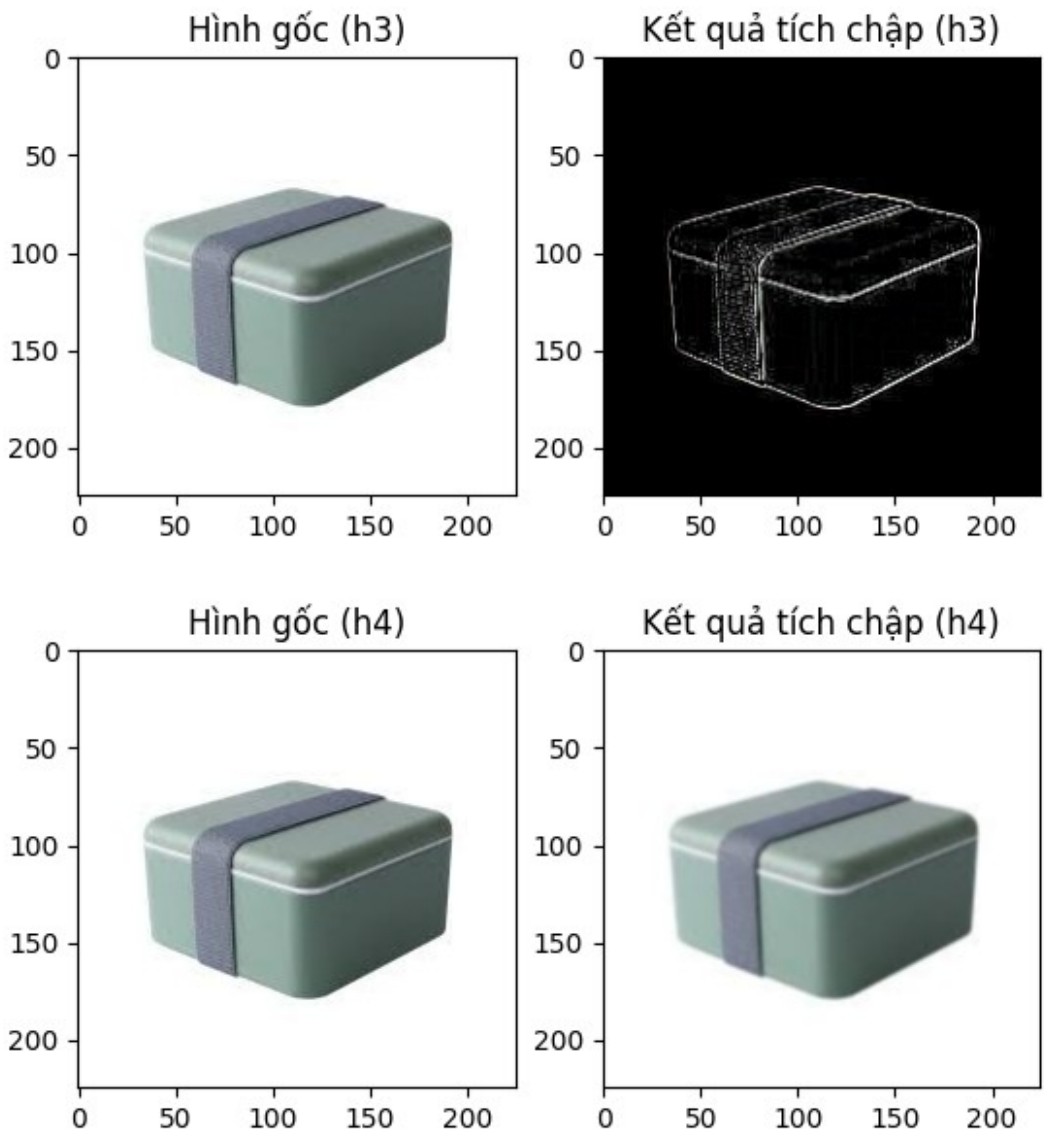
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Đọc hình ảnh đầu vào
image1 = cv2.imread(r'white_square.jpg')
image2 = cv2.imread(r'hop.jpg')

# Kiểm tra nếu hình ảnh không tồn tại
if image1 is None or image2 is None:
    print("Could not open one of the images")

# Định nghĩa các kernel
kernels = {
    "h1": np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]],
dtype=np.float32),
    "h2": np.array([[0, 0, 0], [0, 0, 0], [0, 0, 1]],
dtype=np.float32),
    "h3": np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]],
dtype=np.float32),
    "h4": (1/16) * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]],
dtype=np.float32)
}

# Hàm để áp dụng kernel và hiển thị kết quả
def apply_kernel_and_show(image, kernel, kernel_name):
    result = cv2.filter2D(image, -1, kernel)
    plt.figure()
    plt.subplot(121)
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.title(f"Hình gốc ({kernel_name})")
    plt.subplot(122)
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.title(f"Kết quả tích chập ({kernel_name})")
    plt.show()

# Áp dụng các kernel lên hình ảnh "white_square.jpg"
for kernel_name, kernel in kernels.items():
    apply_kernel_and_show(image1, kernel, kernel_name)

# Áp dụng các kernel lên hình ảnh "hop.jpg"
for kernel_name, kernel in kernels.items():
    apply_kernel_and_show(image2, kernel, kernel_name)
```


Hình gốc (h1)
Kết quả tích chập (h1)
Hình gốc (h2)
Kết quả tích chập (h2)
0
50
100
150
200


Hình gốc (h3)
Kết quả tích chập (h3)
0
50
100
150
200


Hình gốc (h4)
Kết quả tích chập (h4)
0
50
100
150
200


Hình gốc (h1)
Kết quả tích chập (h1)
Hình gốc (h2)
Kết quả tích chập (h2)
0
50
100
150
200



Mô tả hiệu ứng của từng kernel:

1. h1 Tác dụng: Kernel này chỉ sao chép pixel trung tâm mà không ảnh hưởng đến các pixel xung quanh. Do đó, hình ảnh đầu ra sẽ giống hệt với hình ảnh đầu vào.
2. h2 Tác dụng: Kernel này chỉ lấy giá trị từ góc dưới bên phải của vùng 3x3 và bỏ qua các phần tử khác. Điều này sẽ tạo ra một hình ảnh bị dịch chuyển, vì nó chỉ lấy giá trị từ một phần cụ thể của vùng ảnh đầu vào.
3. h3 Tác dụng: Đây là kernel phát hiện biên (edge detection). Nó làm nổi bật các biên giữa các vùng có cường độ khác nhau trong hình ảnh và loại bỏ các phần có cường độ đồng nhất. Hình ảnh đầu ra sẽ là các cạnh sáng trên nền tối, làm nổi bật các biên.
4. h4 Tác dụng: Đây là một kernel làm mịn (Gaussian blur). Nó làm giảm nhiễu và làm mờ hình ảnh đầu vào bằng cách lấy trung bình có trọng số của các pixel xung quanh. Hình ảnh đầu ra sẽ có hiệu ứng làm mờ nhẹ, giúp làm mịn các chi tiết.

2.1.2 Box Blur

Box blur là một phép lọc hình ảnh phổ biến để làm mờ, trong đó mỗi pixel đầu ra được tính bằng giá trị trung bình của các pixel xung quanh nó trong một vùng hình chữ nhật, còn được gọi là kernel. Kernel cho Box blur có thể là một ma trận mà tất cả các phần tử của nó có giá trị bằng nhau và tổng các phần tử của kernel được chuẩn hóa sao cho tổng các giá trị của kernel bằng 1.

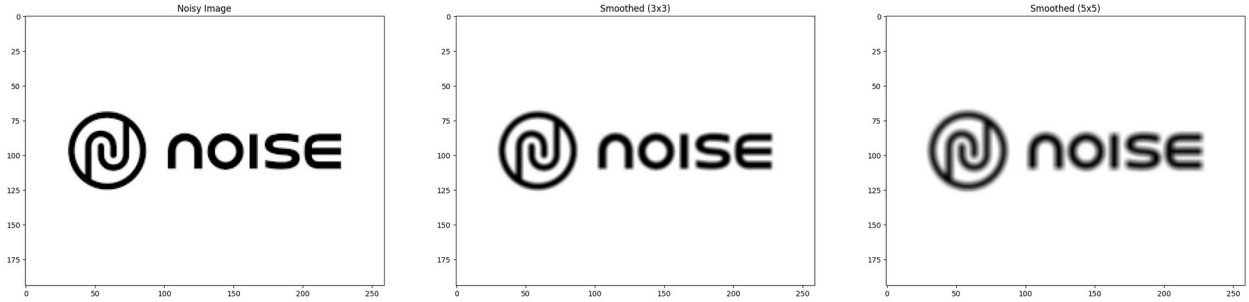
Hàm cv2.blur() trong OpenCV được sử dụng để áp dụng phép làm mờ hình ảnh (Box Blur). Cú pháp của hàm là:

```
dst = cv2.blur(src, ksize[, dst[, anchor[, borderType]]])
```

Thông số:

- `src`: Hình ảnh đầu vào. Hình ảnh có thể có bất kỳ số kênh nào (ví dụ: hình ảnh màu RGB có 3 kênh), và mỗi kênh được xử lý độc lập. Tuy nhiên, độ sâu của hình ảnh cần phải là một trong các kiểu sau: `CV_8U`, `CV_16U`, `CV_16S`, `CV_32F`, hoặc `CV_64F`.
- `dst`: Hình ảnh đầu ra có cùng kích thước và kiểu dữ liệu như hình ảnh đầu vào `src`.
- `ksize`: Kích thước kernel làm mờ, xác định kích thước của vùng mà giá trị pixel được tính trung bình.
- `anchor`: Điểm neo xác định vị trí tương đối của pixel lọc trong kernel. Giá trị mặc định `Point(-1, -1)`có nghĩa là điểm neo nằm ở trung tâm kernel.
- `borderType`: Phương thức ngoại suy pixel khi điểm lọc nằm ngoài ranh giới của hình ảnh. Tùy chọn này xác định cách xử lý các giá trị pixel ngoài vùng ảnh.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread(r'logo_noise2.jpg')

dst1 = cv2.blur(img, (3, 3), (-1, -1))
dst2 = cv2.blur(img, (5, 5), (-1, -1))

plt.figure(figsize=(30, 10))
plt.subplot(131); plt.imshow(img[..., ::-1]); plt.title('Noisy Image')
plt.subplot(132); plt.imshow(dst1[..., ::-1]); plt.title('Smoothed
(3x3)')
plt.subplot(133); plt.imshow(dst2[..., ::-1]); plt.title('Smoothed
(5x5)')
plt.show()
```



Bài tập 2.3: Mô tả các phép làm mịn

Sau khi thực hiện các phép làm mịn trên hình ảnh, bạn cần phân tích kết quả dựa trên hai tiêu chí chính:

1. Độ mịn của nền logo: So sánh mức độ mịn của nền sau khi áp dụng các phép làm mịn.
2. Thay đổi của các đường viền (contours): Nhận xét về sự thay đổi của các đường viền của đối tượng trong hình ảnh khi áp dụng kernel làm mờ.

Phân tích:

1. Độ mịn của nền logo:

- Sau khi áp dụng các phép làm mờ (Box Blur hoặc Gaussian Blur), nền của logo có xu hướng trở nên mịn hơn do các chi tiết nhỏ và nhiễu (noise) trong ảnh bị làm mờ. Đặc biệt, các điểm sáng/tối đột ngột được làm phẳng, dẫn đến sự chuyển tiếp giữa các pixel mượt mà hơn.
- Với Box Blur, vì tất cả các pixel trong vùng kernel đều có trọng số bằng nhau, nó thường làm cho hình ảnh bị mờ đều trên toàn bộ khu vực.
- Gaussian Blur (hoặc bất kỳ phép làm mịn tương tự) sẽ làm mờ nền mạnh hơn nhưng với mức độ khác nhau, với các pixel gần trung tâm kernel có trọng số lớn hơn, do đó hình ảnh sẽ có sự làm mờ tự nhiên hơn so với Box Blur.

1. Thay đổi của các đường viền (contours):

- Khi áp dụng các phép làm mờ, các đường viền xung quanh các đối tượng trong hình ảnh trở nên ít rõ nét hơn. Các cạnh sắc nét ban đầu sẽ trở nên mềm mại và có thể bị làm nhòe, làm mất đi sự tương phản giữa đối tượng và nền.
- Với Box Blur, do kernel áp dụng trọng số đều lên các pixel, các đường viền của đối tượng có thể trở nên không sắc nét và bị nhòe. Điều này có thể làm các đối tượng có hình dạng trở nên ít rõ ràng hơn.
- Gaussian Blur sẽ có hiệu ứng tương tự nhưng có thể giữ lại một số chi tiết sắc nét hơn ở trung tâm của đối tượng, vì nó ưu tiên các giá trị pixel gần trung tâm của vùng kernel hơn là vùng ngoại biên.

Tóm tắt:

- Nền logo: Sẽ trông mịn hơn do phép làm mờ làm giảm nhiễu và chi tiết nhỏ.
- Các đường viền: Sẽ bị mờ đi, khiến cho các đường viền sắc nét ban đầu trở nên mềm mại và không còn rõ ràng như trước khi làm mờ.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread(r'logo_noise2.jpg')

dst1 = cv2.GaussianBlur(img, (3, 3), 0)

plt.figure(figsize=(30, 10))
plt.subplot(131); plt.imshow(img[..., ::-1]); plt.title('Noisy Image')
plt.subplot(132); plt.imshow(dst1[..., ::-1]); plt.title('Smoothed
(3x3)')
plt.show()
```

2.1.3 Gaussian filtering in OpenCV

Kernel hộp (Box kernel) đã được giải thích trước đó phân bổ trọng số đều cho tất cả các pixel trong vùng lân cận. Ngược lại, kernel Gaussian Blur phân bổ trọng số cho các pixel lân cận dựa trên khoảng cách của pixel đó so với pixel trung tâm. Hình dạng của đường cong này được điều chỉnh bởi một tham số duy nhất gọi là σ (sigma), tham số này kiểm soát độ nhọn của đặc tính "đỉnh đồi" của kernel Gaussian.

$$\left(G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \right)$$

Một giá trị **σ** lớn hơn tạo ra một kernel làm mờ mạnh hơn. Kernel Gaussian 5x5 với **σ = 1** được cho bởi:

$$\frac{1}{337}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 20 & 33 & 20 & 4 \\ 7 & 33 & 55 & 33 & 7 \\ 4 & 20 & 33 & 20 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Rõ ràng là pixel ở giữa nhận được trọng số lớn nhất, trong khi các pixel càng xa trung tâm thì trọng số càng nhỏ hơn.

Một hình ảnh được làm mờ bằng cách sử dụng kernel Gaussian trông ít bị mờ hơn so với kernel hộp có cùng kích thước. Một lượng nhỏ làm mờ Gaussian thường được sử dụng để loại bỏ nhiễu từ hình ảnh. Nó cũng được áp dụng cho hình ảnh trước khi thực hiện các phép lọc nhạy cảm với nhiễu. Ví dụ, kernel Sobel được sử dụng để tính toán đạo hàm của một hình ảnh là sự kết hợp giữa kernel Gaussian và kernel sai phân hữu hạn.

Hàm OpenCV thực hiện việc lọc Gaussian là `GaussianBlur` với cú pháp như sau:

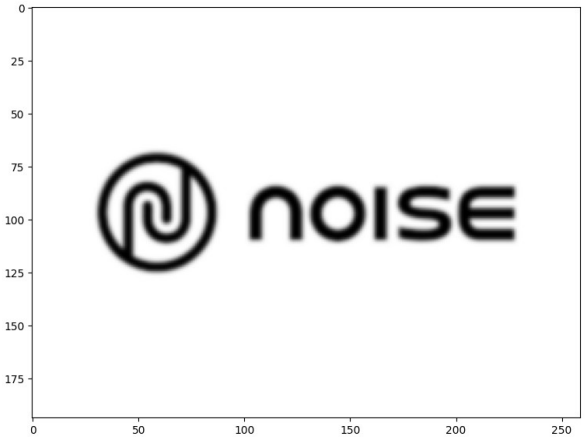
```
dst = cv2.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]])
```

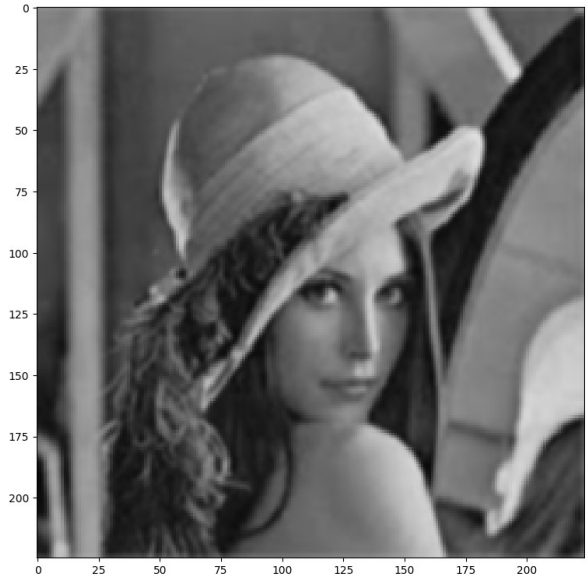
- **`src`**: đại diện cho hình ảnh đầu vào; hình ảnh có thể có bất kỳ số lượng kênh nào và được xử lý độc lập, nhưng độ sâu phải là `CV_8U`, `CV_16U`, `CV_16S`, `CV_32F` hoặc `CV_64F`.
- **`dst`**: là hình ảnh đầu ra có cùng kích thước và loại dữ liệu như `src`.
- **`ksize`**: kích thước kernel Gaussian. `ksize.width` và `ksize.height` có thể khác nhau nhưng cả hai đều phải là số dương và lẻ. Hoặc, chúng có thể là số không và sẽ được tính toán từ `sigma`.
- **`sigmaX`**: độ lệch chuẩn của kernel Gaussian theo hướng X.
- **`sigmaY`**: độ lệch chuẩn của kernel Gaussian theo hướng Y; nếu `sigmaY` bằng 0, nó sẽ được đặt bằng `sigmaX`; nếu cả hai giá trị `sigma` đều bằng 0, chúng sẽ được tính toán từ `ksize.width` và `ksize.height` tương ứng; để kiểm soát hoàn toàn kết quả bất kể các thay đổi có thể xảy ra trong tương lai về tất cả các ý nghĩa này, khuyến nghị nên chỉ định cả `ksize`, `sigmaX` và `sigmaY`.
- **`borderType`**: phương pháp ngoại suy pixel.

Bài tập 2.4: Thêm các dòng mã bị thiếu, được chỉ định bởi các bình luận. Mô tả cả hai phép làm mịn bằng cách so sánh các hình ảnh cuối cùng: Nền logo trông mịn hơn bao nhiêu? Các đường viền trong hình ảnh thay đổi như thế nào với kernel làm mờ? Thử lọc Gaussian cho các hình ảnh thử nghiệm “logo_noise2.jpg” và “lina_noise.jpg” và xác định kích thước kernel tốt nhất.

```
import cv2
import numpy as np

# Đọc ảnh đầu vào
image_logo = cv2.imread(r'logo_noise2.jpg')
image_lina = cv2.imread(r'lina_noise.jpg')

# Kích thước kernel Gaussian
ksize = (5, 5)  # Chọn kích thước kernel tùy ý, có thể thay đổi để thử nghiệm

# Áp dụng lọc Gaussian
image_logo_gaussian = cv2.GaussianBlur(image_logo, ksize, sigmaX=1)
image_lina_gaussian = cv2.GaussianBlur(image_lina, ksize, sigmaX=1)
```

```
# Hiển thị hình ảnh
plt.figure(figsize=(20, 10))
plt.subplot(121)
plt.imshow(image_logo)
plt.imshow(image_logo_gaussian)
plt.subplot(122)

plt.imshow(image_lina)
plt.imshow(image_lina_gaussian)
plt.show()
```





2.1.4 Bộ lọc trung vị (Median filter)

Bộ lọc làm mờ trung vị là một kỹ thuật lọc phi tuyến tính, chủ yếu được sử dụng để loại bỏ nhiễu "muối và tiêu" khỏi hình ảnh. Trong hình ảnh màu, nhiễu muối và tiêu có thể xuất hiện dưới dạng các đốm màu ngẫu nhiên nhỏ. Bộ lọc trung vị trong OpenCV yêu cầu kernel có hình vuông và độ dài cạnh của hình vuông phải là một số lẻ. Bộ lọc làm mờ trung vị thay thế giá trị của pixel trung tâm bằng giá trị trung vị của tất cả các pixel trong vùng kernel. Giá trị trung vị được tính bằng cách sắp xếp dữ liệu theo thứ tự tăng dần và lấy giá trị ở giữa làm ước lượng.

Hãy xem xét một vùng 3x3 từ một hình ảnh bị ảnh hưởng bởi nhiễu muối và tiêu:

$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 255 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

Dễ dàng nhận thấy rằng giá trị của pixel trung tâm cao hơn nhiều so với các pixel xung quanh, có thể là do ảnh hưởng của nhiễu. Nếu chúng ta sử dụng bộ lọc Box Blur để làm mờ nhiễu này, giá trị của pixel trung tâm sau khi lọc sẽ là:

$$\frac{60+62+59+61+255+65+65+60+63}{9}=83.3$$

Do đó, vùng sau khi lọc bằng Box Blur sẽ là:

$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 83 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

Kết quả này rõ ràng là đã có cải thiện, nhưng giá trị trung tâm vẫn cao hơn so với các pixel lân cận. Sử dụng bộ lọc trung vị, các pixel đã sắp xếp theo thứ tự tăng dần sẽ là:

$$[59\,60\,60\,61\,62\,63\,65\,65\,255]$$

Và giá trị ở giữa danh sách đã sắp xếp là 62. Vì vậy, sau khi lọc pixel trung tâm, vùng sẽ trở thành:

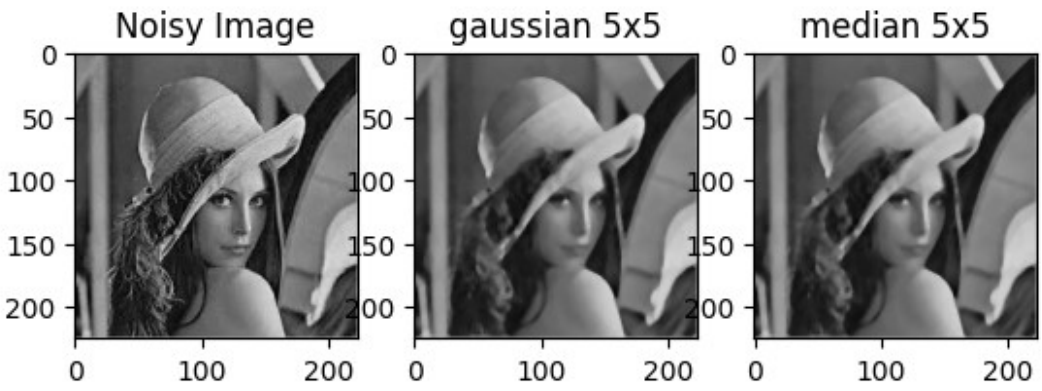
$$\begin{bmatrix} 60 & 62 & 59 \\ 61 & 62 & 65 \\ 65 & 60 & 63 \end{bmatrix}$$

Quá trình này được lặp lại theo cùng một cách cho tất cả các pixel trong hình ảnh đầu vào.

Bài tập 2.5:

Thêm các dòng mã bị thiếu, được chỉ định bởi các bình luận. Loại kết cấu nào bị ảnh hưởng nhiều nhất bởi nhiễu muối và tiêu? Bộ lọc làm mịn nào hoạt động tốt hơn ở các cạnh? Để bảo toàn các cạnh, nên sử dụng kernel lớn hay nhỏ hơn? Bộ lọc nào loại bỏ nhiễu hiệu quả nhất? Hãy thử nghiệm với các kích thước kernel khác!

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread(r'lina_noise.jpg')
# Apply gaussian blur with kernel 5x5 and sigmaX=0, sigmaY=0; output
image dst1
kernelSize = 5
# Performing Median Blurring with kernelSize=5 and store it in numpy
array
dst1 = cv2.medianBlur(img, kernelSize,0)
dst2 = cv2.medianBlur(img, kernelSize, 0)
plt.figure()
plt.subplot(131);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(132);plt.imshow(dst1[...,::-1]);plt.title("gaussian 5x5")
plt.subplot(133);plt.imshow(dst2[...,::-1]);plt.title("median 5x5")
plt.show()
```



2.1.5 Bộ lọc Bilateral

Bộ lọc Bilateral là một kỹ thuật làm mịn phi tuyến tính khác, giúp bảo toàn các đường biên và giảm nhiễu. Hầu hết các bộ lọc làm mịn (ví dụ như Gaussian hay Box filter) có một tham số gọi là σ_s (chữ **s** trong chỉ số dưới là viết tắt của "spatial" - không gian), tham số này xác định mức độ làm mịn. Thường thì giá trị này liên quan chặt chẽ đến kích thước kernel. Một bộ lọc làm mịn thông thường thay thế giá trị cường độ của một pixel bằng tổng có trọng số của các pixel lân cận. Khu vực lân cận càng lớn, hình ảnh sau khi lọc sẽ trông càng mịn hơn. Kích thước của khu vực lân cận tỷ lệ thuận với tham số σ_s.

Trong các bộ lọc bảo toàn đường biên, có 2 mục tiêu cạnh tranh:

- Làm mịn hình ảnh.
- Không làm mịn các đường biên hoặc ranh giới màu.

Hãy xem xét một khối hình ảnh 3x3 với các giá trị sau:

$$\begin{bmatrix} 60 & 200 & 239 \\ 61 & 220 & 235 \\ 65 & 210 & 233 \end{bmatrix}$$

Dễ dàng nhận thấy rằng các giá trị trong cột bên trái thấp hơn nhiều so với các giá trị ở cột giữa và cột bên phải. Khối này là một phần của đường biên dọc. Pixel trung tâm được lọc trong trường hợp này dựa trên chỉ các cột trung tâm và bên phải, vì vậy đường biên được giữ lại và không bị làm mờ. Trong bộ lọc Bilateral, khi tính toán sự đóng góp của bất kỳ pixel nào vào đầu ra cuối cùng, chúng ta ưu tiên các pixel có cường độ gần với pixel trung tâm cao hơn so với các pixel có cường độ rất khác biệt. Trọng số đó là một hàm Gaussian đã được điều chỉnh:

$$G_{\sigma_r}\left(I_c - I_n\right)$$

Trong đó, sự chênh lệch cường độ giữa pixel trung tâm I_s và pixel lân cận I_n được kiểm soát bởi tham số σ_r (chữ **r** viết tắt của "range" - dải màu).

Ngoài ra, giống như bộ lọc Gaussian, chúng ta cũng muốn ưu tiên các pixel gần với pixel trung tâm hơn là các pixel xa hơn, do đó trọng số cũng sẽ phụ thuộc vào $\|c - n\|$. Trọng số trong trường hợp này sẽ là một hàm Gaussian:

$$G_{\sigma_s}\left(\|c - n\|\right)$$

Kết hợp cả hai, bộ lọc Bilateral sẽ cho ra giá trị sau tại pixel trung tâm **c**:

$$O_c = \frac{1}{W_c} \sum G_{\sigma_s}(\|c - n\|) G_{\sigma_r}(I_c - I_n)$$

Trong đó, W_s là một hằng số chuẩn hóa, G_s là kernel Gaussian không gian, G_r là kernel Gaussian màu/dải màu, **c** là vị trí của pixel trung tâm, **n** là vị trí của pixel lân cận, I_s là cường độ tại pixel trung tâm và I_n là cường độ tại vị trí pixel lân cận **n**.

Nếu các pixel lân cận là các đường biên, sự khác biệt về cường độ ($I_s - I_n$) sẽ cao hơn. Vì Gaussian là một hàm giảm, G_r sẽ có trọng số thấp hơn cho các giá trị lớn. Do đó, hiệu ứng làm mịn sẽ ít hơn đối với các pixel như vậy, giúp bảo toàn các đường biên.

Có 2 tham số quan trọng trong bộ lọc Bilateral: σ_s và σ_r. σ_s kiểm soát mức độ làm mịn không gian, và σ_r kiểm soát mức độ trung bình của các màu không giống nhau trong khu vực lân cận. Giá trị σ_r càng lớn, vùng có màu sắc đồng nhất càng lớn.

Hàm OpenCV để thực hiện lọc Bilateral là:

```
dst = cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace[, dst[, borderType]])
```

Các tham số:

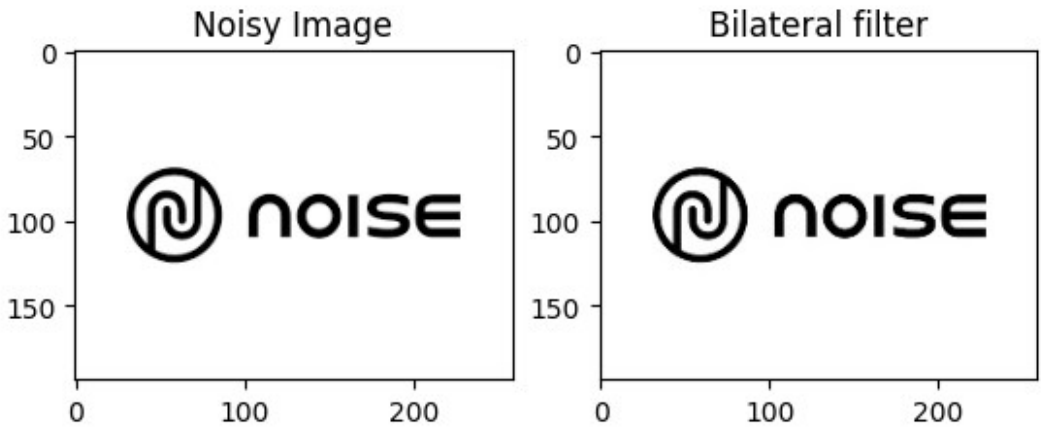
- **`src`**: Hình ảnh đầu vào – hình ảnh có thể là loại 8-bit hoặc điểm nổi, với 1 kênh hoặc 3 kênh.
- **`dst`**: Hình ảnh đầu ra có cùng kích thước và loại dữ liệu như `src`.
- **`d`**: Đường kính của mỗi khu vực lân cận được sử dụng trong quá trình lọc. Nếu giá trị này không dương, nó sẽ được tính từ `sigmaSpace`.
- **`sigmaColor`**: Sigma của bộ lọc trong không gian màu. Giá trị lớn hơn có nghĩa là các màu xa hơn trong khu vực lân cận pixel sẽ được trộn lẫn với nhau, dẫn đến các vùng màu đồng nhất lớn hơn.
- **`sigmaSpace`**: Sigma của bộ lọc trong không gian tọa độ. Giá trị lớn hơn có nghĩa là các pixel xa hơn sẽ ảnh hưởng lẫn nhau miễn là màu sắc của chúng đủ gần nhau.

Bài tập 2.6:

Thêm các dòng mã bị thiếu, được chỉ định bởi các bình luận. Đọc nhiều hình ảnh bị ảnh hưởng bởi các loại nhiễu khác nhau: nhiễu muối và tiêu có trong “logo_noise.jpg” và “lina_noise.jpg”, trong khi nhiễu Gaussian có trong “logo_noise2.jpg”. Thử nghiệm với các giá trị khác nhau của kích thước kernel và sigma để xác định loại lọc nào: trung vị hay Bilateral là tốt nhất trong từng trường hợp, tùy thuộc vào loại nhiễu. Hiển thị kết quả so sánh và nhận xét về lựa chọn tham số tốt nhất cho từng bộ lọc.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread(r'logo_noise2.jpg')
# diameter of the pixel neighbourhood used during filtering
dia=15
```

```
# Larger the value the distant colours will be mixed together
# to produce areas of semi equal colors
sigmaColor=80
# Larger the value more the influence of the farther placed pixels
# as long as their colors are close enough
sigmaSpace=80
#Apply bilateralFilter on img with parameters dia, sigmaColor and
sigmaSpace
dst1 = cv2.bilateralFilter(img, dia, sigmaColor, sigmaSpace)
plt.figure()
plt.subplot(121);plt.imshow(img[...,::-1]);plt.title("Noisy Image")
plt.subplot(122);plt.imshow(dst1[...,::-1]);plt.title("Bilateral
filter")
plt.show()
```



Khi quyết định sử dụng bộ lọc nào, cần lưu ý các điểm sau:

- **Lọc trung vị** là cách tốt nhất để làm mịn hình ảnh bị nhiễu dạng muối và tiêu (các giá trị đột ngột cao/thấp trong khu vực lân cận của một pixel).
- **Lọc Gaussian** có thể được sử dụng nếu có nhiễu Gaussian mức thấp.
- **Lọc Bilateral** nên được sử dụng nếu có mức độ nhiễu Gaussian cao và bạn muốn bảo toàn các đường biên trong khi làm mờ các khu vực khác.
- Về **tốc độ thực thi**, lọc Gaussian là nhanh nhất và lọc Bilateral là chậm nhất.

2.2 Gradients của hình ảnh

Gradient của hình ảnh là sự thay đổi về cường độ (hoặc màu sắc) theo hướng trong một hình ảnh. Chúng ta cần các gradient để nghiên cứu các bộ lọc làm thay đổi sự đột ngột của cường độ. Sau đây là một số ví dụ từ Hình 3 với các điểm được đánh dấu bằng dấu chấm đỏ:

- **Pixel A và B** và khu vực lân cận của chúng có cùng màu, không có thay đổi về cường độ, vì vậy gradient tại những điểm này là 0.
- Tại **Pixel C**, có sự thay đổi đột ngột về cường độ từ trái sang phải, nên pixel C có gradient dương cao theo hướng x. Tương tự, **Pixel D** có gradient dương cao theo hướng y.

- Tại **Pixel E**, cường độ thay đổi từ trắng sang đen, do đó gradient là âm theo hướng x. Tương tự, **Pixel F** có gradient âm theo hướng y.
- **Pixel G và I** có các gradient lần lượt theo hướng x và y, nhưng khá nhẹ. Độ lớn của gradient không còn tối đa nữa, vì sự chuyển tiếp chỉ từ xám đậm sang xám nhạt (không phải từ đen sang trắng như trước).
- **Pixel H** có gradient theo cả hai hướng, đây là trường hợp xảy ra với hầu hết các pixel trong hình ảnh tự nhiên.

Hình 3: Gradients của hình ảnh

Trong hình ảnh màu, các gradient có thể được tính cho từng kênh màu riêng biệt (một pixel sẽ có 6 giá trị, đại diện cho 3 gradient màu theo hướng x và y). Trong nhiều ứng dụng, các gradient màu không cần thiết, nên thông thường hình ảnh màu được chuyển đổi thành ảnh xám và các gradient cường độ được tính toán thay thế.

Chúng ta ký hiệu gradient theo hướng x là $\mathbf{I_x}$ và gradient theo hướng y là $\mathbf{I_Y}$. Gradient của một pixel có thể được xem như một vector với các thành phần x và y. Độ lớn của gradient của một pixel được cho bởi:

$$G = \sqrt{I_x^2 + I_y^2}$$

Trong khi đó, hướng của gradient là:

$$\theta = \arctan\left(\frac{I_y}{I_x}\right)$$

Độ lớn và hướng của gradient được tính tại mỗi pixel trong hình ảnh. $\mathbf{I_x}$ và $\mathbf{I_Y}$ là các hình ảnh được tạo ra bằng cách áp dụng bộ lọc Sobel cho các gradient theo hướng X và Y của hình ảnh.

2.2.1 Bộ lọc Prewitt

Để tính giá trị Gradient theo hướng X của một pixel $\mathbf{I_x}$, chúng ta cần tìm sự chênh lệch về cường độ ở phía bên phải và bên trái của vị trí pixel hiện tại. Bộ lọc Prewitt sau đây dùng để lấy gradient theo hướng X:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Đối với gradient theo hướng Y, kernel Prewitt thứ hai là:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Sử dụng bộ lọc Prewitt để tính toán các gradient $\mathbf{I_x}$ và $\mathbf{I_Y}$ của hình ảnh, chúng ta thực sự đang tính các đạo hàm theo hướng x và y của hình ảnh.

2.2.2 Bộ lọc Sobel

Việc tính toán gradient sẽ chính xác hơn và ít nhiễu hơn nếu hình ảnh được làm mờ Gaussian nhẹ trước khi áp dụng bộ lọc gradient. Bộ lọc Sobel thực hiện làm mịn Gaussian một cách ngầm định. Bộ lọc Sobel cho gradient theo hướng X và Y như sau:

Gradient theo hướng X:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Gradient theo hướng Y:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Hàm OpenCV để thực hiện Sobel filtering là:

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

Thông số:

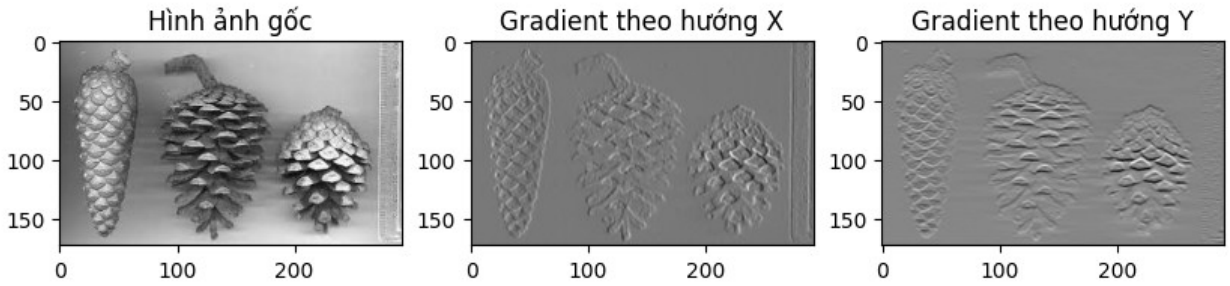
- **`src`**: Hình ảnh đầu vào.
- **`dst`**: Hình ảnh đầu ra có cùng kích thước và số lượng kênh như hình ảnh đầu vào.
- **`ddepth`**: Độ sâu của hình ảnh đầu ra, với hình ảnh đầu vào 8-bit sẽ dẫn đến các đạo hàm bị cắt.
- **`dx`**: Bậc đạo hàm theo hướng x.
- **`dy`**: Bậc đạo hàm theo hướng y.
- **`ksize`**: Kích thước của kernel Sobel mở rộng, phải là 1, 3, 5 hoặc 7.
- **`scale`**: Hệ số tỷ lệ tùy chọn cho các giá trị đạo hàm đã tính toán.
- **`delta`**: Giá trị delta tùy chọn được thêm vào kết quả trước khi lưu vào `dst`.
- **`borderType`**: Phương pháp ngoại suy pixel.

Bài tập 2.7:

- Đọc hình ảnh "Halep.jpg" ở chế độ grayscale.
- Sử dụng hàm Sobel để tính gradient theo hướng X và Y. Đặt độ sâu của hình ảnh đầu ra thành `CV_32F` vì các gradient có thể có giá trị âm.
- Sử dụng hàm `cv2.normalize` để chuẩn hóa các gradient, sao cho tất cả các giá trị pixel nằm trong khoảng từ 0 đến 1.
- Hiển thị cả hai hình ảnh gradient. Liệu chúng có thể được sử dụng như các bộ phát hiện cạnh không?

```
import cv2
import numpy as np
```

```python
import matplotlib.pyplot as plt

# Đọc ảnh Halep.jpg dưới dạng ảnh xám
image = cv2.imread(r'Halep.jpg', cv2.IMREAD_GRAYSCALE)

# Tính gradient theo hướng X và Y sử dụng Sobel
grad_x = cv2.Sobel(image, cv2.CV_32F, 1, 0, ksize=3)
grad_y = cv2.Sobel(image, cv2.CV_32F, 0, 1, ksize=3)

# Chuẩn hóa các gradient để giá trị pixel nằm giữa 0 và 1
grad_x = cv2.normalize(grad_x, None, 0, 1, cv2.NORM_MINMAX)
grad_y = cv2.normalize(grad_y, None, 0, 1, cv2.NORM_MINMAX)

# Hiển thị hình ảnh gốc và các gradient
plt.figure(figsize=(10, 5))

plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Hình ảnh gốc')

plt.subplot(1, 3, 2)
plt.imshow(grad_x, cmap='gray')
plt.title('Gradient theo hướng X')

plt.subplot(1, 3, 3)
plt.imshow(grad_y, cmap='gray')
plt.title('Gradient theo hướng Y')

plt.show()
```



Nhận xét:

- Các gradient theo hướng X và Y có thể được sử dụng làm bộ phát hiện cạnh (edge detectors) vì chúng giúp phát hiện các sự thay đổi mạnh về cường độ giữa các pixel, tương ứng với các cạnh trong hình ảnh.
- **Gradient theo hướng X** sẽ phát hiện các cạnh dọc, trong khi **gradient theo hướng Y** sẽ phát hiện các cạnh ngang.

2.2.3 Bộ lọc Laplacian

Bộ lọc Laplacian dựa trên đạo hàm bậc hai và cũng được sử dụng để nhận diện các cạnh. Về mặt toán học, toán tử Laplacian hoặc bộ lọc Laplacian được cho bởi:

$$L(x,y)=\frac{\partial^2 I}{\partial x^2}+\frac{\partial^2 I}{\partial y^2}$$

Kernel tương ứng cho tích chập là:

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

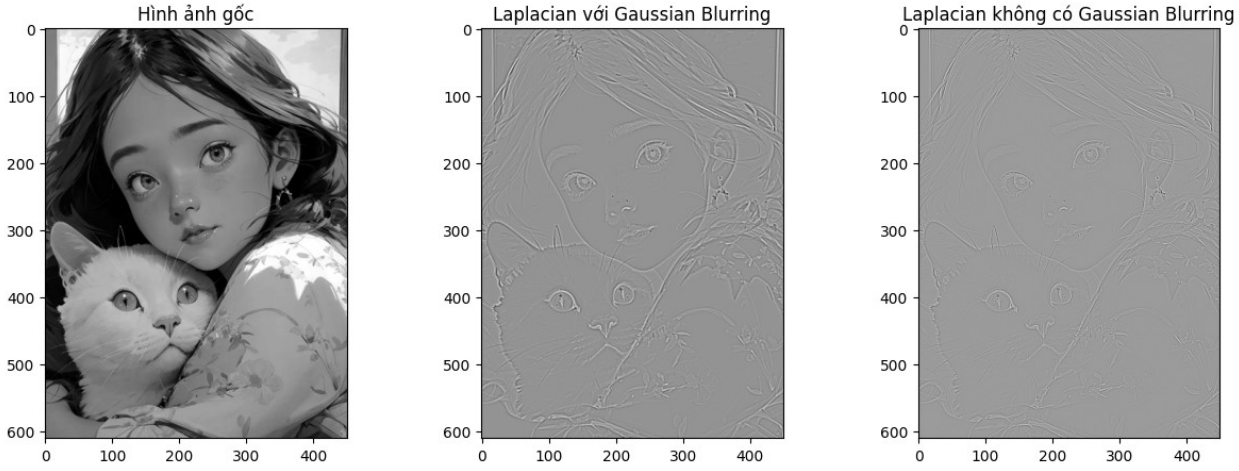
Bộ lọc Laplacian rất nhạy cảm với nhiễu, do đó quan trọng là phải làm mịn hình ảnh trước khi áp dụng bộ lọc này. Hàm tương ứng trong OpenCV là `cv2.Laplacian`.

Bài tập 2.8:

1. Đọc ảnh "Hummingbird.jpg" ở chế độ grayscale.
2. Áp dụng Gaussian Blurring bằng cách sử dụng `cv2.GaussianBlur` với kích thước kernel 3x3 pixel.
3. Áp dụng bộ lọc Laplacian lên hình ảnh đã làm mờ trước đó. Đặt độ sâu của hình ảnh đầu ra thành `CV_32F` vì gradient có thể có giá trị âm.
4. Sử dụng hàm `cv2.normalize` để chuẩn hóa gradient, sao cho tất cả các giá trị pixel nằm trong khoảng từ 0 đến 1.
5. Hiển thị hình ảnh đã chuẩn hóa cuối cùng. Lợi ích của bộ phát hiện cạnh Laplacian so với bộ lọc Sobel là gì?
6. Thử áp dụng bộ lọc Laplacian mà không làm mờ Gaussian. So sánh hai hình ảnh đã lọc!

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Đọc ảnh "Hummingbird.jpg" ở chế độ grayscale
image = cv2.imread(r'D:\Image-Proccessing\pratice\week_1\img\
00750.jpeg', cv2.IMREAD_GRAYSCALE)

# Áp dụng Gaussian Blurring với kernel 3x3
image_blur = cv2.GaussianBlur(image, (3, 3), 0)

# Áp dụng bộ lọc Laplacian lên ảnh đã làm mờ
laplacian_blurred = cv2.Laplacian(image_blur, cv2.CV_32F, ksize=3)

# Chuẩn hóa kết quả Laplacian sau khi làm mờ
laplacian_blurred_norm = cv2.normalize(laplacian_blurred, None, 0, 1,
cv2.NORM_MINMAX)

# Áp dụng bộ lọc Laplacian lên ảnh gốc mà không làm mờ
```

```
laplacian_no_blur = cv2.Laplacian(image, cv2.CV_32F, ksize=3)

# Chuẩn hóa kết quả Laplacian không làm mờ
laplacian_no_blur_norm = cv2.normalize(laplacian_no_blur, None, 0, 1,
cv2.NORM_MINMAX)

# Hiển thị hình ảnh gốc, ảnh Laplacian với Gaussian Blurring và
không có Gaussian Blurring
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Hình ảnh gốc')

plt.subplot(1, 3, 2)
plt.imshow(laplacian_blurred_norm, cmap='gray')
plt.title('Laplacian với Gaussian Blurring')

plt.subplot(1, 3, 3)
plt.imshow(laplacian_no_blur_norm, cmap='gray')
plt.title('Laplacian không có Gaussian Blurring')

plt.show()
```



Giải thích:

1. **Lợi ích của bộ phát hiện cạnh Laplacian so với bộ lọc Sobel**:
   - **Laplacian**: Là một toán tử tính đạo hàm bậc hai, nó phát hiện các thay đổi cường độ mạnh và được sử dụng để phát hiện các cạnh trong cả hai hướng (X và Y) cùng một lúc. Điều này giúp việc phát hiện cạnh nhanh chóng hơn mà không cần tính riêng biệt các gradient theo X và Y.
   - **Sobel**: Bộ lọc Sobel tính toán các gradient riêng biệt theo hướng X và Y, vì vậy Sobel phù hợp hơn trong việc phát hiện các cạnh dọc và ngang riêng lẻ. Điều này cho phép kiểm soát tốt hơn nhưng yêu cầu nhiều bước tính toán.
2. **So sánh giữa Laplacian có Gaussian Blurring và không có Gaussian Blurring**:

- **Với Gaussian Blurring**: Hình ảnh sẽ mượt mà hơn, giúp giảm nhiễu và các cạnh phát hiện sẽ rõ ràng hơn.
- **Không có Gaussian Blurring**: Nhiễu trong hình ảnh có thể được coi là cạnh, dẫn đến kết quả nhiều cạnh giả hơn và các cạnh bị nhòe.

2.3 Làm sắc nét hình ảnh

Làm sắc nét hình ảnh nhằm mục đích tăng cường các đường viền và làm nổi bật kết cấu bên dưới. Ý tưởng này không phải mới, thực ra nó được lấy từ một kỹ thuật cũ gọi là **unsharp masking**. Các bước sau đây là cần thiết để đạt được hiệu ứng **unsharp masking**:

- **Bước 1**: Làm mờ hình ảnh đầu vào để làm mịn kết cấu. Hình ảnh bị làm mờ chứa các thông tin tần số thấp từ hình ảnh gốc. Gọi I là hình ảnh đầu vào ban đầu và I^b là hình ảnh bị làm mờ.
- **Bước 2**: Trừ hình ảnh bị làm mờ khỏi hình ảnh gốc, I - I^b, để lấy được thông tin tần số cao từ hình ảnh gốc.
- **Bước 3**: Thêm lại thông tin tần số cao vào hình ảnh gốc và kiểm soát lượng kết cấu tần số cao được thêm vào bằng cách sử dụng tham số α. Hình ảnh cuối cùng được làm sắc nét là:

$$I_s = I + \alpha(I - I^b)$$

Hiện tại, việc làm sắc nét được thực hiện bằng cách sử dụng một kernel làm sắc nét đơn giản, mô phỏng hành vi trên. Hình ảnh đầu vào được tích chập với kernel sau:

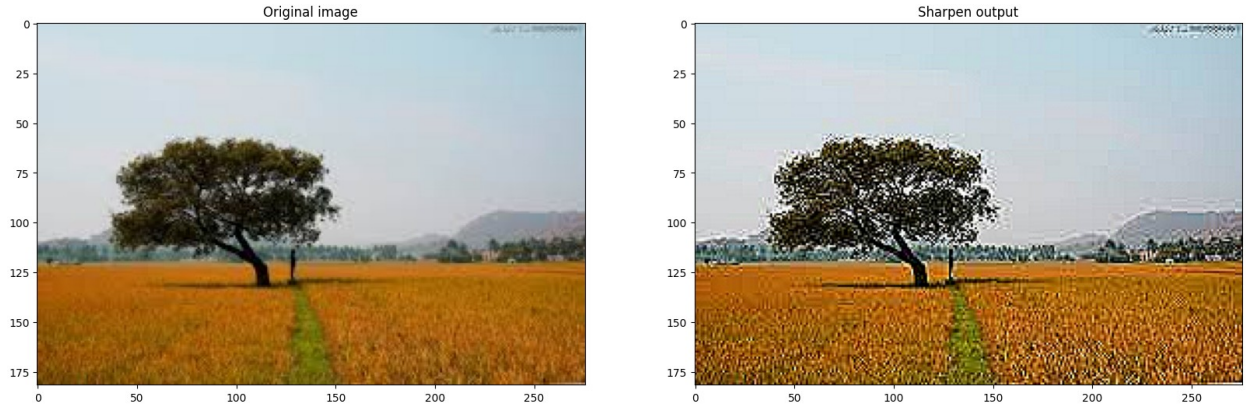
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Kernel trên được tạo ra bằng cách sử dụng α **= 1** và mô phỏng I **-** I^b bằng kernel Laplacian:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

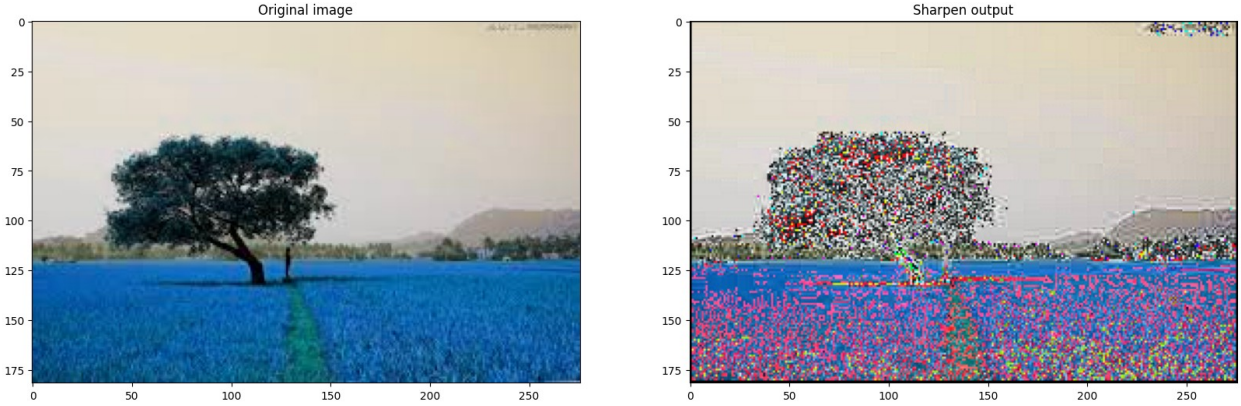
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread(r"tree1.jpg")
# Sharpen kernel
sharpen = np.array(( [0, -1, 0],
 [-1, 5, -1],
 [0, -1, 0]), dtype="int")
sharpImg = cv2.filter2D(img, -1, sharpen)
plt.figure(figsize=[20,10])
plt.subplot(121);plt.imshow(img[...,::-1]);plt.title("Original image")
plt.subplot(122);plt.imshow(sharpImg[...,::-1]);plt.title("Sharpen
```

```
output")
plt.show()
```




```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread(r"tree1.jpg")
img = np.array(img)

# Sharpen kernel
sharpen = np.array(( [0, -1, 0],
 [-1, 5, -1],
 [0, -1, 0]), dtype="int")
# Hàm tích chập (convolution)
def apply_kernel(img, kernel):
    # Lấy kích thước ảnh và kernel
    img_height, img_width, num_channels = img.shape
    kernel_height, kernel_width = kernel.shape

    # Tạo một ảnh rỗng để chứa kết quả
    output_img = np.zeros_like(img)

    # Duyệt qua từng điểm ảnh, bỏ qua viền (vì kernel không thể
áp dụng lên viền)
    for h in range(1, img_height - 1):
        for w in range(1, img_width - 1):
            for c in range(num_channels):  # Áp dụng trên từng kênh
màu (RGB)
                # Lấy vùng ảnh con tương ứng với kernel
                region = img[h-1:h+2, w-1:w+2, c]

                # Tính tích chập giữa vùng con và kernel
                output_img[h, w, c] = np.sum(region * kernel)

    # Đảm bảo giá trị điểm ảnh nằm trong khoảng [0, 255]
    output_img = np.clip(output_img, 0, 255)
```

```
    return output_img.astype(np.uint8)

# Áp dụng kernel làm sắ́c nét
sharp_img = apply_kernel(img, sharpen)

# Hiể̂n thị ả̂nh gốc và ả̂nh sau khi làm sắ́c nét
plt.figure(figsize=[20,10])
plt.subplot(121); plt.imshow(img); plt.title("Original image")
plt.subplot(122); plt.imshow(sharp_img); plt.title("Sharpen output")
plt.show()
```



Bài tập 2.9: Hiệu ứng làm sắc nét hình ảnh

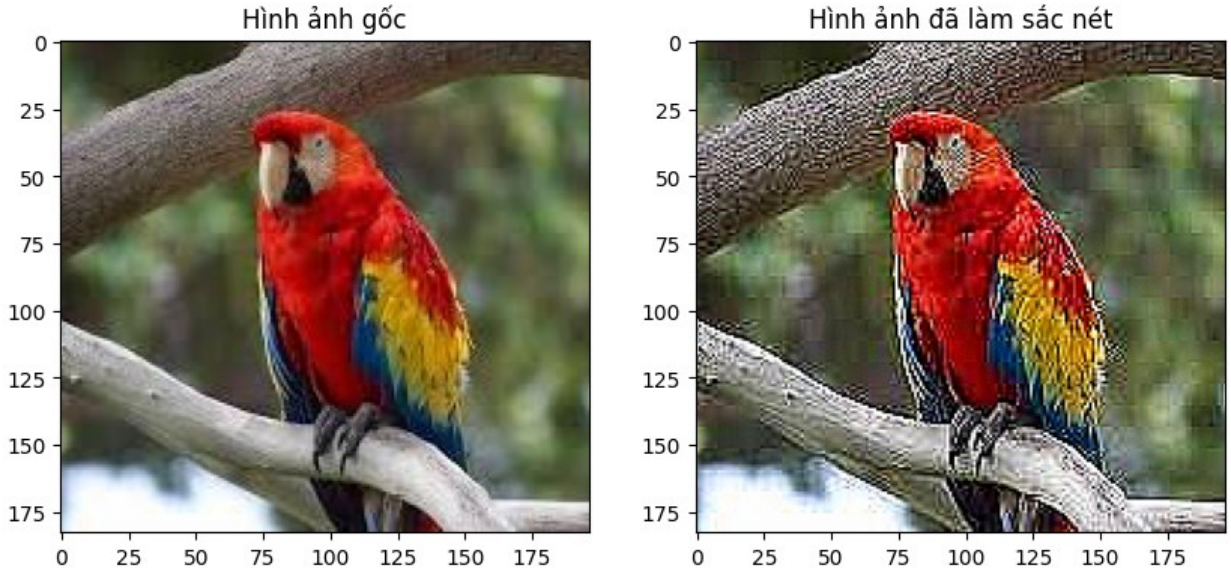
- **Các thành phần kết cấu nào được làm nổi bật bởi hiệu ứng làm sắc nét?**
  Hiệu ứng làm sắc nét chủ yếu nhấn mạnh vào các cạnh và các vùng có sự thay đổi đột ngột về cường độ, làm cho các chi tiết nhỏ và đường biên của đối tượng trong hình ảnh trở nên rõ ràng hơn. Các vùng kết cấu như lông chim, lá cây hoặc các bề mặt có nhiều thay đổi về cường độ sẽ được làm nổi bật nhiều nhất.

- **Thử nghiệm với các hình ảnh khác, chẳng hạn như "parrots.jpg".**
  Sau khi áp dụng bộ lọc làm sắc nét lên hình ảnh "parrots.jpg", ta có thể nhận thấy các chi tiết như lông chim, kết cấu của bề mặt cây, và các chi tiết nhỏ trong hình ảnh được làm nổi bật hơn. Bộ lọc làm sắc nét sẽ làm rõ các đường viền của các chi tiết nhỏ này, giúp chúng dễ thấy hơn so với trước khi áp dụng bộ lọc.

- **Mô tả loại kết cấu nào trở nên nổi bật sau khi áp dụng bộ lọc làm sắc nét.**
  Các loại kết cấu có sự thay đổi nhanh chóng về cường độ hoặc màu sắc, chẳng hạn như cạnh của lông chim, các chi tiết trên bề mặt cây, hoặc các đường viền giữa các vùng màu sắc khác nhau, sẽ trở nên nổi bật hơn sau khi áp dụng bộ lọc làm sắc nét. Những vùng có bề mặt phẳng hoặc ít thay đổi về cường độ sẽ ít bị ảnh hưởng hơn.

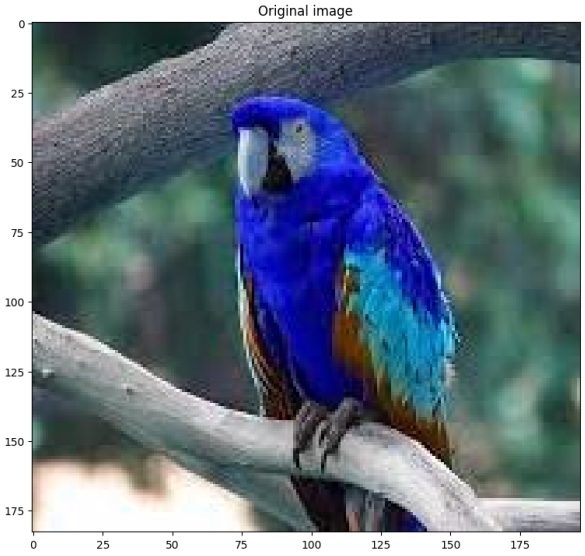
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Đọc ả̂nh "parrots.jpg"
```

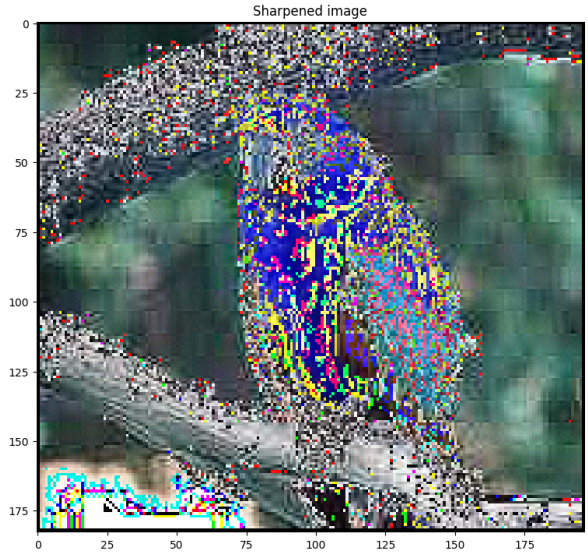
```
image = cv2.imread(r'parrots.jpg')

# Kernel làm sắ́c nét
sharpening_kernel = np.array([[0, -1, 0],
                              [-1, 5, -1],
                              [0, -1, 0]])

# Áp dụng bộ lọc làm sắ́c nét
sharpened_image = cv2.filter2D(image, -1, sharpening_kernel)

# Hiển thị hình ảnh gốc và hình ảnh đã làm sắ́c nét
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Hình ảnh gốć')

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(sharpened_image, cv2.COLOR_BGR2RGB))
plt.title('Hình ảnh đã làm săć nét')

plt.show()
```




```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Đọc ảnh "parrots.jpg"
image = cv2.imread(r'parrots.jpg')

# Kernel làm sắ́c nét
```

```
sharpening_kernel = np.array([[0, -1, 0],
                              [-1, 5, -1],
                              [0, -1, 0]])
# Hàm tự viết để áp dụng kernel làm sắc nét (tích chập)
def apply_sharpen_kernel(image, kernel):
    # Lấy kích thước ảnh và kernel
    img_height, img_width, num_channels = image.shape
    kernel_height, kernel_width = kernel.shape

    # Tạo một mảng rỗng để chứa kết quả
    output_image = np.zeros_like(image)

    # Duyệt qua từng pixel (trừ biên vì kernel không áp dụng được cho
biên)
    for h in range(1, img_height - 1):
        for w in range(1, img_width - 1):
            for c in range(num_channels):  # Áp dụng trên từng kênh
màu (RGB)
                # Lấy vùng 3x3 pixel xung quanh pixel hiện tại
                region = image[h-1:h+2, w-1:w+2, c]

                # Tính tích chập giữa kernel và vùng ảnh
                output_image[h, w, c] = np.sum(region * kernel)

    # Đảm bảo giá trị điểm ảnh nằm trong khoảng [0, 255]
    output_image = np.clip(output_image, 0, 255)
    return output_image.astype(np.uint8)

# Áp dụng kernel làm sắc nét
sharpened_img = apply_sharpen_kernel(image, sharpening_kernel)

# Hiển thị ảnh gốc và ảnh sau khi làm sắc nét
plt.figure(figsize=[20, 10])
plt.subplot(121); plt.imshow(image); plt.title("Original image")
plt.subplot(122); plt.imshow(sharpened_img); plt.title("Sharpened
image")
plt.show()
```





Kết luận:

Bộ lọc làm sắc nét giúp làm nổi bật các đường viền và chi tiết nhỏ, đặc biệt là trong các vùng có sự thay đổi nhanh về cường độ hoặc màu sắc, chẳng hạn như lông chim hoặc các bề mặt gồ ghề. Các kết cấu mịn hoặc phẳng sẽ ít bị ảnh hưởng hơn so với các vùng có nhiều thay đổi.