# Лабораторна Робота №1
# Тема: Основи цифрової обробки зображень

**Об'єднує:** Теми 1 + 2 + 3

- Робота з цифровими зображеннями (RGB, grayscale, канали) (Відкрити зображення у Python (Pillow або OpenCV)), (Відобразити його та вивести розміри, кількість каналів), (Зберегти копію з іншими параметрами (формат, розмір));
- Застосування фільтрів: розмиття, підсилення різкості, виділення контурів (Накласти Gaussian blur та sharpening на фото);
- Виділення контурів: порогове виділення (threshold) та виділення контурів за допомогою Canny;
- Реалізація згортки у numpy (наприклад, Sobel для країв), порівняння з готовими функціями OpenCV (Порівняти результат із cv2.filter2D, Візуалізувати матрицю фільтра і результат на зображенні).

**Результат:** базові навички роботи з бібліотеками, розуміння математики згортки та фільтрів.

## 1. Робота з цифровими зображеннями

### 1.1. Перевірка та встановлення пакету OpenCV використовуючи Python

OpenCV має обгортку у вигляді Python пакету, який називається cv2.

Створіть файл та назвіть його наприклад *laboratorna.py*. Спочатку варто перевірити чи встановлений модуль cv2. Для цього у файлі додайте наступний код (Рис. 1.1).

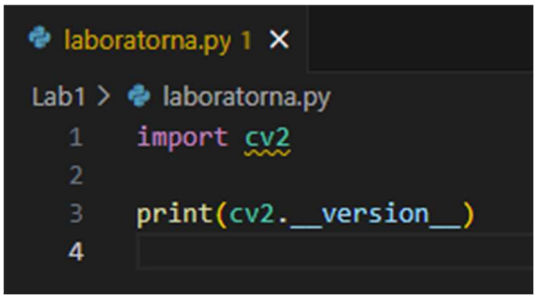


Рис. 1.1. Вміст файлу для перевірки чи встановлено пакет cv2

Виконайте його у консолі за допомогою наступної команди (Рис. 1.2).

```
python .\laboratorna.py
```

Рис. 1.2. Команда для виконання вмісту файлу laboratorna.py

Якщо модуль не встановлено, то буде виведено наступну помилку (Рис. 1.3.).

```
Traceback (most recent call last):
  File "I:\Лабораторні\Lab1\laboratorna.py", line 1, in <module>
    import cv2
ModuleNotFoundError: No module named 'cv2'
```

Рисунок 1.3. Помилка яку буде виведено у консолі, якщо пакет cv2 не встановлено

В такому випадку потрібно встановити модуль cv2 наступною командою з використанням менеджеру пакетів для python – pip (в деяких випадках може версія pip3) (Рис. 1.4.).

```
> pip install opencv-python
```

Рисунок 1.4. Команда для встановлення пакету cv2, використовуючи менеджер пакетів pip

Після цього почнеться встановлення потрібного нам модулю.

Тепер, якщо ви виконаєте знову команду для виконання файлу (Рис. 1.2), має вивестися версія встановленого пакету не виводячи ніяких помилок.

### 1.2. Відкриття зображення у OpenCV

Для того, щоб відкрити та відобразити зображення засобами opencv, у створений попередньо файл необхідно додати наступний код (Рис. 1.5.).

```
img = cv2.imread('cat.jpg')
cv2.imshow('Cat Image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.5. Код для відкриття та відображення зображення засобами OpenCV

При цьому, важливо зазначати, що файл, який ви збираєтеся відкрити був у тій же самій папці, що і наш файл з кодом. Далі, виконаємо код із файлу попередньою командою (Рис. 1.2.). Якщо правильно вказано файл, та він існує у вказаному шляху, відкриється вікно із зображенням (Рис. 1.6.).

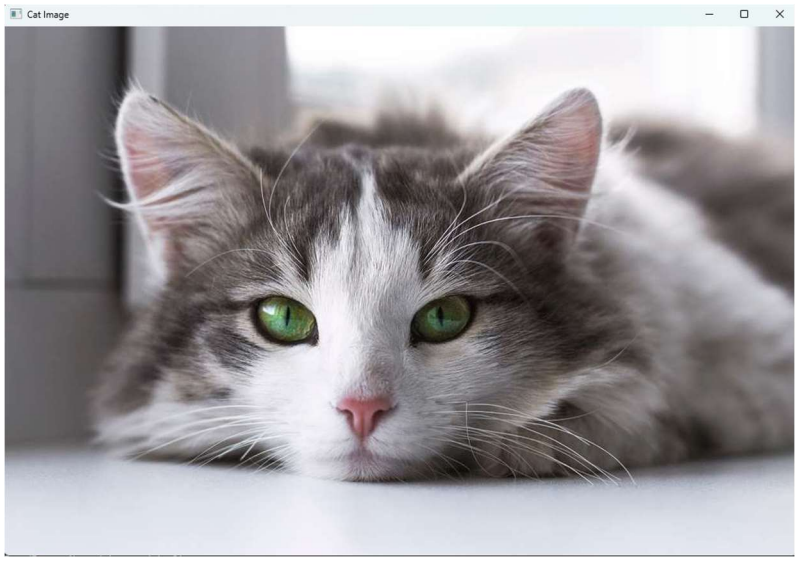


Рисунок 1.6. Вікно із зображенням

Окрім цього, використовуючи ту саму команду для відкриття файлу *cv2.imread* ми можемо вказати кольоровий простір відкритого файлу. Наприклад, якщо ми хочемо, щоб файл було відкрито у колірному просторі градацій сірого, ми можемо вказати відповідне значення другим параметром функції *cv2.imread(_filename_, color_space).* Щоб відобразити зображення відкрите зображення у колірному просторі градацій сірого

змініть функцію відкриття зображення. Загальний вигляд коду має бути наступний (Рис. 1.7.).

```
img = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)
cv2.imshow('Cat Image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.7. Код для відкриття зображення у колірному просторі градацій сірого

Ви повинні отримати наступний результат (Рис. 1.8.).


Рисунок 1.7. Вікно із відкритим зображенням у колірному просторі градацій сірого

Другим параметром функції cv2.imread виступає ціле число, яке ми отримуємо використовуючи константу cv2.IMREAD_GRAYSCALE, та може приймати такі значення:

- cv2.IMREAD_COLOR – завантажує кольорове зображення. Будь-яка прозорість зображення буде ігнорована;
- cv2.IMREAD_GRAYSCALE - завантажує кольорове зображення в режимі градації сірого;
- cv2.IMREAD_UNCHANGED - завантажує зображення повністю, включаючи альфа-канал.

### 1.3. Відображення інформації про зображення

OpenCV також може вивести різну інформацію про наше зображення.

Наприклад, ми можемо вивести інформацію про роздільну здатність, кількість каналів та тип даних.

Для відображення цією інформації можна використати наступний код (Рис. 1.8.).

```
img = cv2.imread('cat.jpg')
print("Форма зображення:", img.shape)
print("Тип даних:", img.dtype)
```

Рисунок 1.8. Код для відображення базової інформації про зображення

У результаті виконання даного коду ми маєте отримати наступний результат у терміналі(Рис. 1.9.).

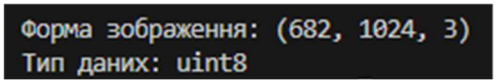


Рисунок 1.9. Результат виконання попереднього коду для отримання базової інформації про зображення

У результаті виконання коду, ми маємо отримати наступну інформацію:

- Рядок «Форма зображення» містить інформацію про роздільну здатність (висота –682 пікселі, ширина – 1024 пікселі) та кількість каналів у зображенні (3 число у дужках);
- Рядок «Тип даних» містить інформацію про тип даних елементів у зображенні – тобто, який формат чисел використовується для зберігання інтенсивності каналів. На зображенні (Рис. 1.9.) було отримано значення *uint8*, що означає, що в зображенні було використано тип 8-біт на канал із діапазоном значень 0-255. Також, серед найпоширеніших типів, які можна зустріти є наступні:
    - *int16* – використовується для зображень із більшим динамічним діапазоном -32768-32767;
    - *float32* – 32-бітний тим з плаваючою комою, який використовується при обробці чи нормалізації та має діапазон значень 0.0-1.0;
    - *float64* – 64-бітний тип з плаваючою комою, який використовується для наукових обчислень для високої точності, має діапазон 0.0-1.0 або більші.

**1.4. Розділення зображення на окремі канали**

Для розділення та відображення зображення в окремих каналах, у функції, яку ми використовували для виведення зображення необхідно додати другий параметр. Цим параметром ми вказуємо, який саме канал ми хочемо отримати.

Щоб відобразити зображення розділено по каналах можемо скористатися наступним кодом (Рис. 1.10.).

```
img = cv2.imread('cat.jpg')
b, g, r = cv2.split(img)
cv2.imshow("Blue channel", b)
cv2.imshow("Green channel", g)
cv2.imshow("Red channel", r)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.10. Приклад коду для розділення зображення на канали та вивід окремо кожного каналу

При цьому ми отримаємо три вікна із зображенням у градаціях сірого для окремих каналів (R, G, B) (Рис. 1.11.). Може виникнути питання, якщо ми виводимо зображення для каналів R, G, B чому отримані зображення у градаціях сірого? Річ у тім, що OpenCV розділяє кольорове зображення (BGR) на три одноканальні матриці:

- b — канал інтенсивності синього кольору;
- g — канал інтенсивності зеленого;
- r — канал інтенсивності червоного;

Кожен із них — це 2D-матриця яскравостей (0–255), тому при показі через cv2.imshow вони виглядають як сірі зображення:

- світлі ділянки = де канал має високе значення
- темні ділянки = де цього кольору мало.

Рисунок 1.11. Три вікна із зображенням для відповідного каналу R, G, B

Функція *cv2.imshow()* не знає, що це «синій канал» — вона бачить лише одноканальне зображення й показує його у відтінках сірого.

Щоб «візуалізувати» канал саме у своєму кольорі, потрібно додати нульові матриці для інших каналів і об'єднати. Для цього використаємо функцію *cv2.merge* (Рис. 1.12.), в якій ми вказуємо канал, який хочемо відобразити, а для інших вказуємо нулі.

```
img = cv2.imread('cat.jpg')
b, g, r = cv2.split(img)
zeros = np.zeros_like(b)
blue = cv2.merge([b, zeros, zeros])
green = cv2.merge([zeros, g, zeros])
red = cv2.merge([zeros, zeros, r])
cv2.imshow("Blue channel", blue)
cv2.imshow("Green channel", green)
cv2.imshow("Red channel", red)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.12. Приклад коду для розділення зображення на канали та вивід окремо кожного каналу у відповідному кольорі

При цьому ми отримаємо три вікна із зображенням у градаціях сірого для окремих каналів R, G, B у відповідному кольорі (Рис. 1.13.).

Рисунок 1.13. Три вікна із зображенням для відповідного каналу R, G, B

### 1.5. Перегляд гістограм каналів

Також використовуючи модулі Python ми можемо вивести гістограму зображення для окремих каналів. Для цього нам знадобиться інший пакет Python під назвою matplotlib. Для цього інсталюйте пакет matplotlib використовуючи попередньо використану команду *pip install* (Рис. 1.4).

Для отримання гістограми каналів необхідно скористатися наступним кодом (Рис. 1.14).

```
import matplotlib.pyplot as plt

img = cv2.imread('cat.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
    hist = cv2.calcHist([img],[i],None,[256],[0,256])
    plt.plot(hist, color=col)
plt.title('Histogram per channel')
plt.show()
```

Рисунок 1.14. Приклад коду для виведення гістограми каналів для зображення

У результаті, ми отримаємо наступне вікно із відображеною гістограмою каналів для нашого зображення (Рис. 1.15.).

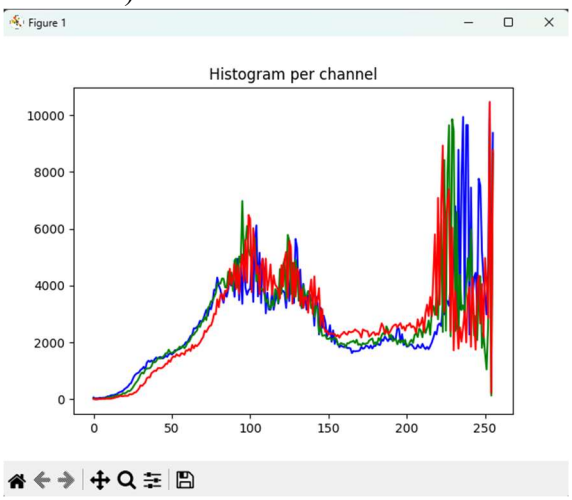


Рисунок 1.15. Вікно із виведеною гістограмою каналів для зображення із використанням matplotlib

Даний пакет matplotlib також можна використовувати для отримання базової інформації про зображення, як із допомогою OpenCV. Окрім цього, matplotlib може надати інформацію про середнє значення по каналах, тип каналу (RGBA, RGB, L – визначає чи є альфа-канал), діапазон пікселів – для перевірки нормалізації зображення.

### 1.6. Маніпулювання та збереження зображення

*1.6.1. Виділення «регіонів інтересу»*

Виділення “регіонів інтересу” (ROI) є важливим механізмом у процесі обробки зображень. Це можна зробити вручну за допомогою зрізів масивів. Для цього скористаймося наступним кодом (Рис. 1.16.).

```
img = cv2.imread('cat.jpg')
roi = img[300:500, 325:625]
cv2.imshow("RoI", roi)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.16. Приклад коду для вирізання області зображення RoI

В даному прикладі коду ми вказуємо координати пікселів для області, яку хочемо вирізати. Першим елементом масиву будуть координати за віссю x (300 та 500). Другим елементом масиву будуть координати за віссю y (325 та 625). В результаті виконання даного коду, ми отримаємо наступне вікно із вирізаним зображенням.

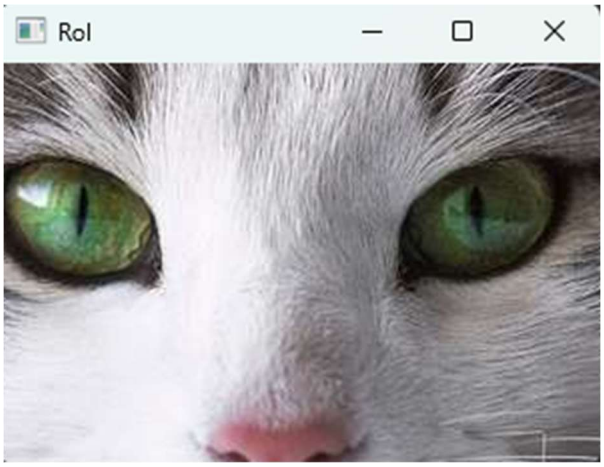


Рисунок 1.17. Вікно із вирізаною частиною зображення за вказаними координатами

*1.6.2. Додання намальованого прямокутника до зображення*

Щоб візуалізувати та краще зрозуміти систему координат та область яку вирізаємо, ми можемо розмістити прямокутник із червоними гранями для візуалізації виділення даної області на оригінальному зображенні. Для цього скористаймося наступним кодом (Рис. 1.18.).

```
img = cv2.imread('cat.jpg')
cv2.rectangle(img, (325, 300), (625, 500), (0, 0, 255), 2)
cv2.imshow("RoI", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.18. Приклад коду відображення прямокутника для виділення області

В даному прикладі, ми використовуємо команду *cv2.rectangle*, де параметрами виступають наше зображення, координати кутів області, які ми використали у попередньому прикладі, значення кольору для прямокутника у форматі RGB (0, 0, 255),

та товщина лінії прямокутника (рівна 2). У результаті отримаємо наступне вікно із зображенням (Рис. 1.19.).

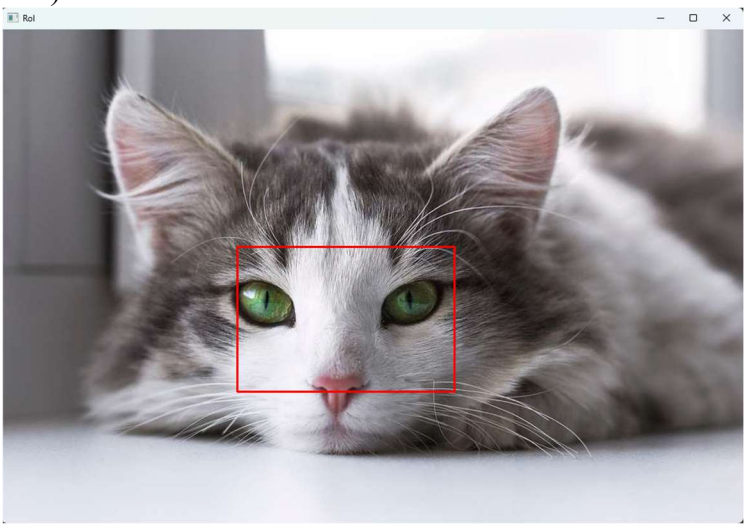


Рисунок 1.19. Вікно із виділеною частиною зображення за вказаними координатами

*1.6.3. Вирізання частини зображення та збереження її у окремому файлі*

Далі ми спробуємо вирізати виділену частину із попереднього прикладу та зберегти її у окремому файлі. Для цього нам потрібно скористатися командою *cv2.resize* для зміни розміру зображення та командою *cv2.imwrite* для збереження зображення у новому файлі (Рис. 1.20.).

```
img = cv2.imread('cat.jpg')
resized_roi = cv2.resize(img[300:500, 325:625], (300, 200))
cv2.imwrite('cat_resized.jpg', resized_roi)
cv2.imshow("RoI", resized_roi)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.20. Приклад коду зміни розміру зображення та збереження вирізаної області у новому файлі

У результаті виконання даного коду ми отримаємо вікно із зображенням схожим до попереднього результату (Рис. 1.17.). Також буде створено новий файл та збережено у нашій директорії – cat_resized.jpg.

Окрім цього, ми можемо вказати останнім параметром метод інтерполяції для незначного покращення якості зображення, якщо отримане зображення після зміни розміру є не дуже чітким. В результаті команда для обрізання зображення буде мати наступний вигляд:

```
resized_roi_interpolation = cv2.resize(img[300:500, 325:625],
        (300, 200), interpolation=cv2.INTER_CUBIC)
```

*1.6.4. Повертання зображення*

Наступним способом маніпулювання зображенням, яке ми розглянемо буде повертання зображення. Для цього ми скористаємося командою *cv2.getRotationMatrix2D*, яка приймає наступні параметри:

- координати центру, навколо якого будемо проводити обертання;
- кут обертання;
- коефіцієнт масштабування.

Даний метод обчислює матрицю афінного перетворення, яка кодує поворот навколо заданого центру та масштаб. При цьому він не змінює зображення, а лише будує матрицю зображення із застосуванням ефекту обертання. Для застосування отриманої матриці і власне отримання зображення із потрібним обертанням, потрібно використати інший метод *cv2.warpAffine*, який приймає наступні параметри:

- джерело зображення, над яким ми проводимо зміни;
- матриця для застосування, у нашому випадку матриця із застосуванням ефекту обертання отримана із методу *cv2.getRotationMatrix2D*;
- розміри нового зображення;
- додаткові параметри, напр., значення інтерполяції та ін..

У результаті застосування даних методів ми маємо отримати наступний код для застосування ефекту обертання нашого зображення на кут 45 градусів за годинниковою стрілкою навколо центру зображення без масштабування зображення (Рис. 1.21).

```
img = cv2.imread('cat.jpg')
h, w = img.shape[:2]     # висота і ширина зображення
center = (w/2, h/2)      # визначаємо центр зображення
angle = -45              # кут повороту за годинниковою стрілкою
scale = 1.0              # коефіцієнт масштабування
M = cv2.getRotationMatrix2D(center, angle, scale)
rotated_image = cv2.warpAffine(img, M, (w, h))
cv2.imshow("Rotated (center)", rotated_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 1.21. Приклад коду застосування ефекту обертання зображення

Після виконання даного коду ми маємо отримати наступне вікно із зображенням (Рис. 1.22.).

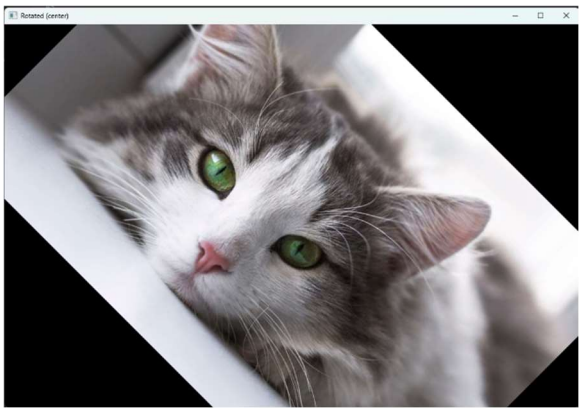


Рисунок 1.22. Вікно із зображенням із застосованим ефектом обертання

# 2. Застосування фільтрів

## 2.1. Розмиття зображень

*2.1.1. Усереднене розмиття*

Розглянемо наступний ефект, який часто використовується при обробці зображення – а саме ефект розмиття зображення. Почнемо із найпростішого варіанту розмиття – усередненого. Даний тип розмиття є простим і швидким. Дає м'яке, але дещо «пластилінове» зображення. Погано працює при наявності контурів – може розмазати їх. Для цього із пакету OpenCV використаємо метод *cv2.blur*. Він приймає наступні параметри:

- вхідне зображення, яке будемо розмивати;
- масив цифр, які позначають значення ядра розмиття.

Додайте у наш файл наступний код та виконайте його (Рис. 2.1).

```
img = cv2.imread('cat.jpg')
blurred = cv2.blur(img, (11, 11))
cv2.imwrite('cat_blurred.jpg', blurred)
cv2.imshow("Blurred Image", blurred)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.1. Приклад коду для застосування ефекту усередненого розмиття зображення

Після виконання даного коду ви повинні отримати схоже зображення (Рис. 2.2). Також ви повинні отримати новий файл у поточній папці з назвою cat_blurred.jpg. Дане розмите зображення будемо використовувати для наступних маніпуляцій.

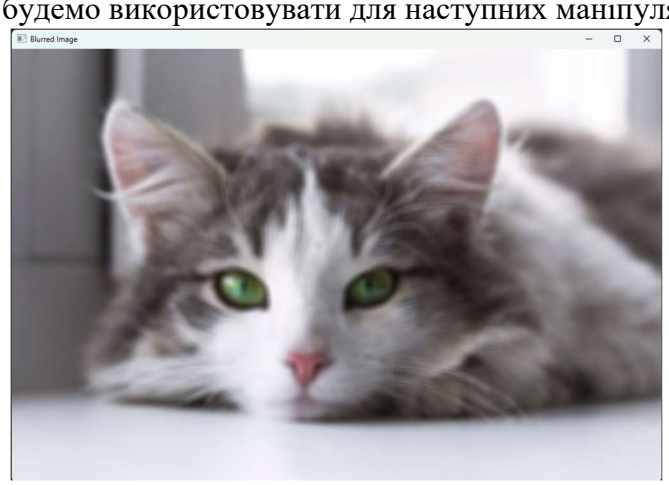


Рисунок 2.2. Вікно із зображенням із застосованим ефектом усередненого розмиття

Поекспериментуйте з різними значеннями ядра розмиття, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.1.2. Гаусове розмиття*

Наступним розглянемо метод *cv2.GaussianBlur* пакету OpenCV для Гаусового розмиття. Даний метод дає найбільш природне розмиття, при цього добре прибираючи шум із зображення. Також зберігає контури краще, ніж звичайне усереднення. Для нашого прикладу використаємо ядро розмиття рівне 11 пікселів. Також останнім

параметром методу буде вказано 0, що означає σ для підбирання автоматичного значення, щоб фільтр мав природний рівень згладжування. Параметр σ (sigma) у cv2.GaussianBlur() визначає, наскільки сильно буде розмивання. Це стандартне відхилення гаусового ядра — воно описує, як швидко зменшується вплив сусідніх пікселів. Розглянемо деякі приклади значень цього параметру:

- Мале σ (0.5–1) → слабке розмиття, чіткі контури зберігаються;
- Середнє σ (2–3) → помітне згладжування шуму;
- Велике σ (5 і більше) → сильне розмиття, зникають дрібні деталі.

Для застосування ефекту розмиття, ми використаємо наступний код (Рис. 2.3).

```
img = cv2.imread('cat.jpg')
blurred = cv2.GaussianBlur(img, (11, 11), 2) # розмиття з ядром 11x11, sigma=2
cv2.imshow("Blurred Image", blurred)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.3. Приклад коду для застосування ефекту Гаусового розмиття зображення

У результаті, маємо отримати наступне вікно із розмитим зображенням (Рис. 2.4).

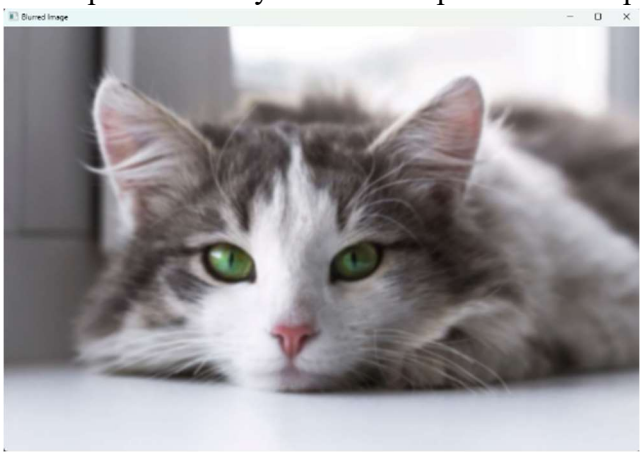


Рисунок 2.4. Вікно із зображенням із застосованим ефектом Гаусового розмиття

Проекспериментуйте різні варіанти значень гаусового ядра та стандартного відхилення гаусового ядра (сигма), щоб порівняти якими будуть результати від застосування різних значень. Для яких значень зображення почне втрачати відчутну якість та важко буде розібрати окремі деталі зображення. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.1.3. Медіанне розмиття*

Наступним розглянемо медіанне розмиття, для якого використаємо метод *cv2.medianBlur*. Даний метод здатен зберігати контури чіткими при застосуванні. Часто використовується в обробці медичних або документальних зображень. На відміну від

```
img = cv2.imread('cat.jpg')
blurred = cv2.medianBlur(img, 11)
cv2.imshow("Blurred Image", blurred)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

попередніх методів, даний приймає не масив значень ядра розмиття, а тільки одне значення (Рис. 2.5).

Рисунок 2.5. Приклад коду для застосування ефекту медіанного розмиття зображення

В результаті отримаємо наступне зображення (Рис. 2.6).

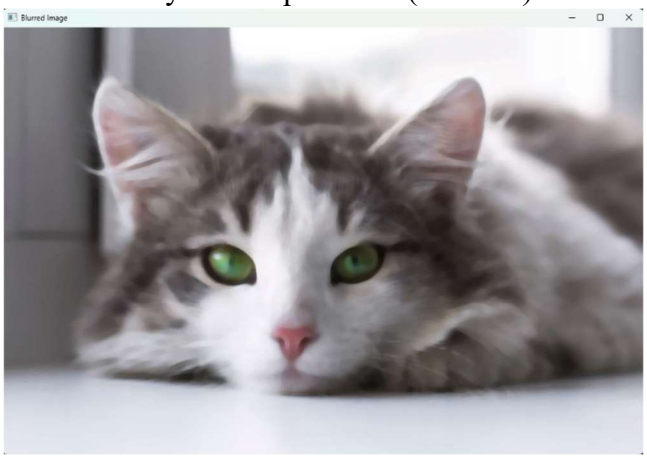


Рисунок 2.6. Вікно із зображенням із застосованим ефектом медіанного розмиття

Поекспериментуйте з різними варіантами значень ядра, щоб порівняти якими будуть результати від застосування різних значень. Для яких значень зображення почне втрачати відчутну якість та важко буде розібрати окремі деталі зображення. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.1.4. Білатеральне розмиття*

Наступним розглянемо білатеральне розмиття, для якого будемо використовувати метод *cv2.bilateralFilter*. Даний тип розмиття є одним із найкращим методів при якому відбувається збереження контурів при розмитті. Застосування методу є повільнішим ніж Гаусове розмиття, бо воно враховує колірну відстань. Добре підходить для портретів та зображень, де важлива деталізація контурів.

Даний метод має наступні параметри:

- вхідне зображення, яке будемо розмивати;
- d - діаметр ядра розмиття. Якщо вказати d = 0, OpenCV обчислить його автоматично на основі sigmaSpace;
- sigmaColor - параметр згладжування в колірному просторі. Він визначає, наскільки сильно пікселі з різними кольорами можуть впливати один на одного. Якщо sigmaColor малий, то будуть зберігатися дрібні кольорові переходи (менше розмиття). Якщо великий, то кольори сильніше усереднюються (сильніше розмиття всередині областей);
- sigmaSpace - параметр згладжування в просторовій (геометричній) області. Він визначає, наскільки далеко сусідні пікселі впливають на центральний. Якщо значення мале, то будуть враховуватися лише близькі пікселі. Якщо велике, то вплив мають навіть далекі пікселі в межах вікна d.

Для нашого прикладу використаємо значення sigmaColor = 35, та sigmaSpace = 15. Чому саме такі значення ми приймаємо? Оскільки значення ядра розмиття ми використовуємо таке ж як і для інших прикладів розмиття – 11 пікселів, то значення sigmaColor та sigmaSpace будуть вираховуватися за наступними формулами:

$$\sigmaColor \approx 2 \times d \text{ до } 3 \times d$$
$$\sigmaSpace \approx d$$

Відтак при використанні значення ядра 11, ми отримаємо такі значення sigmaColor ≈ 20–40, sigmaSpace ≈ 10–15. Наш код для застосування білатерального розмиття буде виглядати наступним чином (Рис. 2.7).

```
img = cv2.imread('cat.jpg')
blurred = cv2.bilateralFilter(img, d=11, sigmaColor=35, sigmaSpace=15)
cv2.imshow("Blurred Image", blurred)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.7. Приклад коду для застосування ефекту білатерального розмиття зображення

В результаті виконання даного коду, маємо отримати наступне зображення (Рис. 2.8).

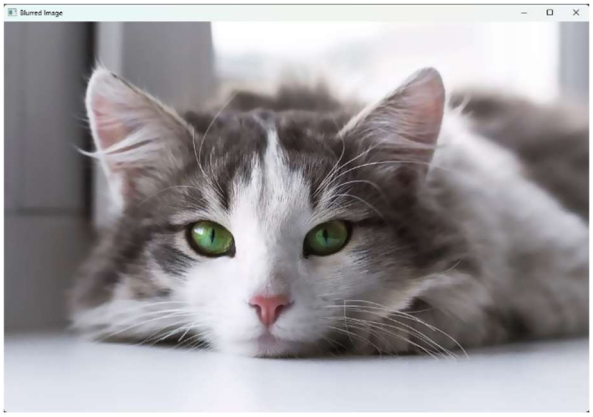


Рисунок 2.8. Вікно із зображенням із застосованим ефектом білатерального розмиття

Поекспериментуйте з різними варіантами значень ядра, щоб порівняти якими будуть результати від застосування різних значень. Для яких значень зображення почне втрачати відчутну якість та важко буде розібрати окремі деталі зображення. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

Зауважте, що для кожного типу розмиття, ми використовували те саме значення ядра розмиття. Порівняйте застосовані типи розмиття між собою. Який тип розмиття найкраще зберігає контури та деталі зображення? Який найгірший?

### 2.2. Підсилення різкості

*2.2.1. Unsharp Masking*

Наступним типом маніпулювання із зображеннями, яке ми розглянемо буде підсилення різкості. Першим типом підсилення різкості розглянемо Unsharp Masking (USM). Цей тип – є базовим вибором, який найчастіше використовується. Даний тип підсилення різкості створює спочатку розмиту копію використовуючи оригінальне зображення, яке ми віднімаємо від оригіналу та підсилюємо залишок (високі частоти). Такий підхід забезпечує природний вигляд зображення та контрольованість. Проте,

серед недоліків використання можна відзначити можливу появу «ореолів» на контрастних межах та підсилення шумів. Найчастіше даний тип використовується для загальних фотографій, друку, інтернет-сайтах та підготовки перед детекцією контурів.

Отже, для використання даного підходу нам спочатку потрібно розмити зображення, щоб отримати розмиту маску. Для цього використаємо Гаусове розмиття. Наступним кроком буде застосування маски, для того, щоб відняти маску від оригінального зображення. Для цього буде використано метод cv2.addWeighted, який приймає наступні параметри:

- вхідне зображення;
- alpha — вага оригіналу (посилює контраст деталей);
- зображення, яке виступає як розмита маска;
- beta — вага розмиття (негативна, для “віднімання” м’якості);
- gamma — зсув яскравості (зазвичай 0).

Наш код для застосування методу підвищення чіткості USM буде виглядати наступним чином (Рис. 2.9).

```
img = cv2.imread('cat_blurred.jpg')
cv2.imshow("Original Image", img)
mask = cv2.GaussianBlur(img, (3,3), 0.8)
sharpen = cv2.addWeighted(img, 3.6, mask, -2.6, 0)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.9. Приклад коду для застосування USM для підвищення чіткості

У даному прикладі, ми використовуємо Гаусове розмиття з ядром 3x3, яке має забезпечити м’яке розмиття для отримання маски. Параметр alpha у даному випадку приймаємо 3.6 для збільшення ваги оригіналу, beta приймаємо -2.6 для збільшення «негативної» ваги розмитої копії, gamma приймаємо 0, щоб не змінюватися яскравість зображення. Дані параметри можна вважати агресивною конфігурацією для сильного посилення чіткості. Зазвичай, базовим варіантом буде спробувати почати із параметрами alpha = 2.2 та beta = -1.2.

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті спроби підвищити чіткість (Рис. 2.10).

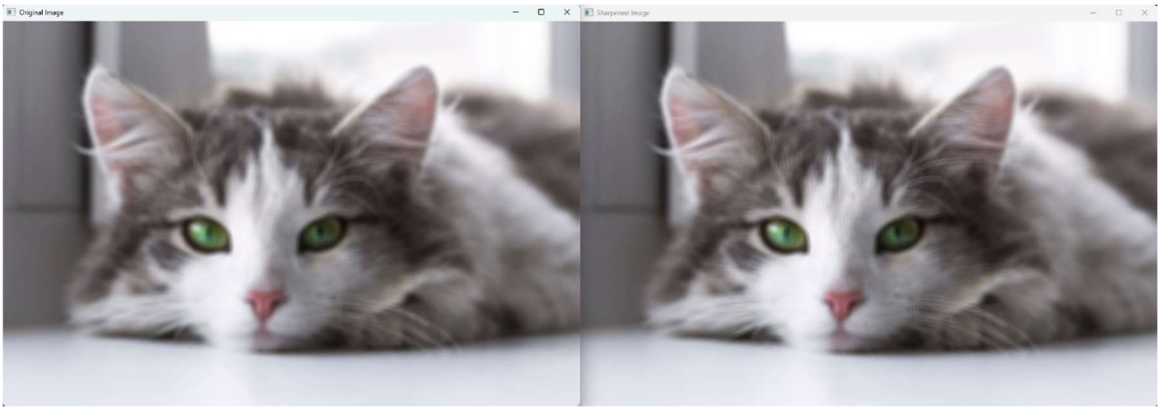

Рисунок 2.10. 2 вікна із оригінальним зображенням та підвищеною чіткістю

На перший погляд може здатися, що два зображення абсолютно однакові та не відрізняються між собою.

1. Річ у тім, що як приклад ми використали зображення, яке отримали з використанням усередненого розмиття з ядром 11×11, яке прибрало занадто багато високочастотних деталей, які USM фізично вже не може відновити.
2. Якщо придивитися уважніше, можна помітити, що деякі елементи зображення стали справді чіткішими та легше розрізнити окремі деталі, на відміну від оригінального розмитого зображення (Рис. 2.11).

Рисунок 2.11. Порівняння деталей між оригінальним розмитим зображенням та після підвищення чіткості

Поекспериментуйте з різними варіантами значень ядра та ваг маскування, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.2.2. Підвищення чіткості із використанням оператора Лапласа*

Далі розглянемо підвищення чіткості з використанням оператора Лапласа, яке краще використовувати для технічних та текстових зображень. Даний тип чітко підкреслює межі та його краще використовувати для тексту чи сканувань. Проте, також має і недоліки: легко «шумить», може виглядати неприродно на фото людей.

Використання даного підходу вимагає наступних кроків:

1. Переведення зображення у градації сірого використовуючи метод *cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).* Цей крок є необхідним, оскільки оператор Лапласа працює із значеннями інтенсивності.
2. Обчислення Лапласіана використовуючи метод *cv2.Laplacian(gray, cv2.CV_16S, ksize=3)*. Як параметри, ми вказуємо: зображення у градаціях сірого, яке ми отримали з попереднього кроку; тип вхідних даних, яке у нашому прикладі має значення *cv2.CV_16S* – 16-бітне знакове зображення; *ksize=3* представляє собою розмір ядра Лапласа, значення 3 – оптимальне значення між точністю та шумом. На цьому кроці ми обчислюємо другу похідну для виявлення області, де яскравість змінюється найсильніше.
3. Перетворення до 8-бітного формату використовуючи метод *cv2.convertScaleAbs(lap)*, який приймає як значення зображення, яке ми отримали на попередньому кроці. Цей крок необхідний, оскільки OpenCV не може напряму показати/додати 16-бітне зображення. У результаті отримаємо зображення контурів (чорне тло, білі межі).

4. Перетворення назад у BGR використовуючи метод *cv2.cvtColor(lap, cv2.COLOR_GRAY2BGR)*, який приймає як параметр зображення, отримане з попереднього кроку. Цей крок необхідний для об'єднання з оригінальним кольоровим зображенням, бо вони мають бути однакової кількості каналів (3).
5. Останній крок, на якому ми об'єднуємо зображення контурів з оригінальним зображенням. Для цього використовуємо такий же метод, який ми використовувати для USM типу *cv2.addWeighted(img, 1.0, lap_bgr, 1.0, 0)*.

Для об'єднання контурів з оригінальним зображенням, ми використовуємо значення параметрів: alpha=1.0 – вага оригіналу, якою ми визначаємо скільки залишити оригінальної яскравості; beta=1.0 – значення контурів для високих частот для підсилення за рахунок додавання; gamma=0 – зміщення яскравості.

Наш код для застосування методу підвищення чіткості із використанням оператора Лапласа буде виглядати наступним чином (Рис. 2.12).

```
img = cv2.imread('cat_blurred.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
lap = cv2.Laplacian(gray, cv2.CV_16S, ksize=3)
lap = cv2.convertScaleAbs(lap)
lap_bgr = cv2.cvtColor(lap, cv2.COLOR_GRAY2BGR)
sharpen = cv2.addWeighted(img, 1.0, lap_bgr, 1.0, 0)
cv2.imshow("Original Image", img)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.12. Приклад коду для застосування оператора Лапласа для підвищення чіткості

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті спроби підвищити чіткість (Рис. 2.13).

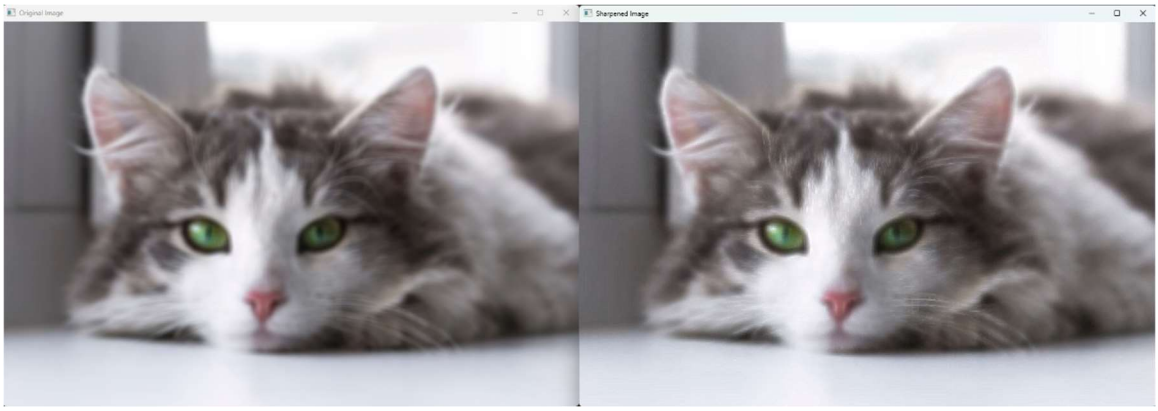

Рисунок 2.13. 2 вікна із оригінальним зображенням та підвищеною чіткістю з використанням оператора Лапласа

Цього разу ми отримали зображення, яке дещо більше відрізняється від оригінального, якщо порівнювати із методом USM. Можна помітити, що контури та деталі здаються більш чіткими та вираженими.

Поекспериментуйте з  різними варіантами значень ядра та ваг маскування, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.2.3. Edge-Preserving Sharpen (bilateral + USM)*

Далі розглянемо підвищення чіткості з використанням методу *Edge-Preserving Sharpen*, який поєднує білатеральний підхід та USM підхід. Основна ідея цього підходу полягає у розмиванні USM через білатеральний підхід, щоб не розмазувати межі. В даному випадку ми отримаємо менше ореолів так кращі краї об'єктів. Разом з тим до недоліків використання такого методу можна віднести: повільніше виконання в порівнянні з Гаусовим та можливість виникнення ефекту «пластиліну» на великих ділянках зображення. Переважно даний підхід використовується для роботи із портретами та фотографій з дрібними краями.

Для застосування даного методу нам необхідно трохи змінити код, який ми використовували для USM методу, а саме змінити Гаусове розмиття на білатеральне (Рис. 2.14).

```
img = cv2.imread('cat_blurred.jpg')
base = cv2.bilateralFilter(img, d=9, sigmaColor=35, sigmaSpace=15)
sharpen = cv2.addWeighted(img, 1.6, base, -0.6, 0)
cv2.imshow("Original Image", img)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.14. Приклад коду для застосування білатерального розмиття як маски для підвищення чіткості

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті застосування поточного підходу (Рис. 2.15).

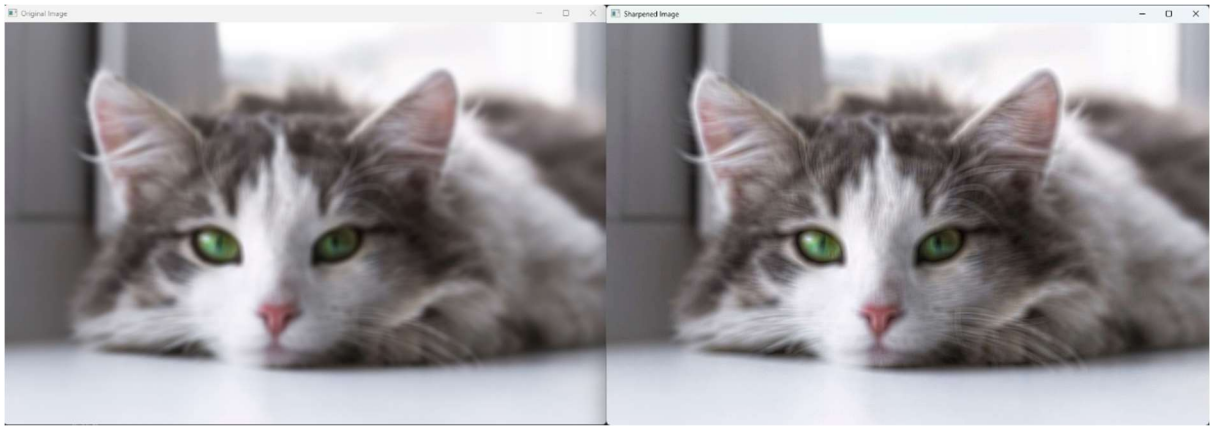


Рисунок 2.15. 2 вікна із оригінальним зображенням та підвищеною чіткістю з використанням оператора Лапласа

Зверніть увагу, для методу *cv2.addWeighted* ми використали ті ж значення, що і для підходу USM із пункту 2.2.1.. Проте зараз, ми отримали зображення, на якому можна помітити більшу відмінність між зображеннями. Деякі окремі деталі та краї окремих

об’єктів стали чіткішими та виразнішими. Також, ми отримали відсутність ореолів, в порівняні з попереднім підходом з використанням оператора Лапласа.

Поекспериментуйте з різними варіантами значень ядра та ваг маскування, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.2.4. Контрольоване маскування країв (Canny-mask)*

Даний метод застосовується для точкової дії, а саме підсилення чіткості лише там, де є краї чи деталі. Завдяки цьому зображення будуть мати мінімум шумових артефактів на фонах. Недоліком можна назвати необхідність чіткого та довгого процесу налаштування, який полягає у підборі значень для отримання кращих результатів. Найкраще використовувати даний підхід у випадках, коли зображення містять сцени з плоским фоном та чіткими об’єктами, наприклад для зображень, які містять продукти або маркофотографії.

Використання даного підходу досить схожим із підходом з використанням оператора Лапласа, оскільки має більшу кількість кроків, ніж інші методи, і також має необхідність перетворення зображення у градації сірого. Використання даного методу передбачає виконання наступних кроків:

1. Переведення зображення у градації сірого використовуючи метод *cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).* Цей крок є необхідним, оскільки нам необхідно отримати значення інтенсивності.
2. Виявлення країв методом Canny. Метод Canny знаходить краї на основі градієнтів яскравості. Приймає наступні параметри:
    a. вхідне зображення;
    b. нижній поріг значень сірого, для ігнорування дуже слабких градієнтів;
    c. верхній поріг значень сірого, для визначення сильних країв.

    Для нашого прикладу приймемо значення 50 та 150. Якщо градієнт має значення більше 150, піксель вважається краєм. Якщо менше 50, не враховується. Якщо значення градієнту між 50 та 150 – піксель враховується, якщо з’єднаний із сильним краєм.
3. Розмиття та логічна маска. Для розмиття ми використаємо Гаусове розмиття. Мета даного кроку – зробити краї трохи ширшими та м’якшими, щоб різкість додавалася не тільки в одну лінію, а з невеликим розширенням. При цьому ми використовуємо перетворення в булеву маску (True/False), де True = область країв. В результаті цього кроку, ми отримаємо логічну матрицю, яку можна використати для умовного накладення.
4. Об’єднання зображення контурів з оригінальним зображенням. Для цього використовуємо такий же метод, який ми використовувати для USM типу із Гаусовим розмиттям *cv2.addWeighted(img, 3.6, cv2.GaussianBlur(img,(3,3),1), -2.6, 0)*. Це USM (Unsharp Masking), але в м’якій формі. Результатом буде більш різке зображення, проте ще не накладене на маску.
5. Умовне накладання різкості лише на краї. При цьому значення *mask[...,None]* буде додавати вимір, щоб розмір маски *(H×W×1)* відповідав кольоровому

зображенню *(H×W×3)*. Значення *np.where(condition, X, Y)* буде брати піксель із *sharp*, якщо умова істинна (край), або з *img*, якщо фон. Результатом буде контрольоване (edge-aware) підвищення чіткості, а саме деталі підсилюються, фон залишається гладким.

В результаті ми маємо отримати наступний код (Рис. 2.16).

```
img = cv2.imread('cat_blurred.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150)
mask = cv2.GaussianBlur(edges, (3,3), 0) > 0
sharp = cv2.addWeighted(img, 3.6, cv2.GaussianBlur(img,(3,3),1), -2.6, 0)
sharpen = np.where(mask[...,None], sharp, img)
cv2.imshow("Original Image", img)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.16. Приклад коду для застосуванням контрольованого маскування країв

Для накладання маски, ми використали ті ж самі значення, що і для попереднього випадку та USM, а саме 3.6 та -2.6.

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті застосування поточного підходу (Рис. 2.17).

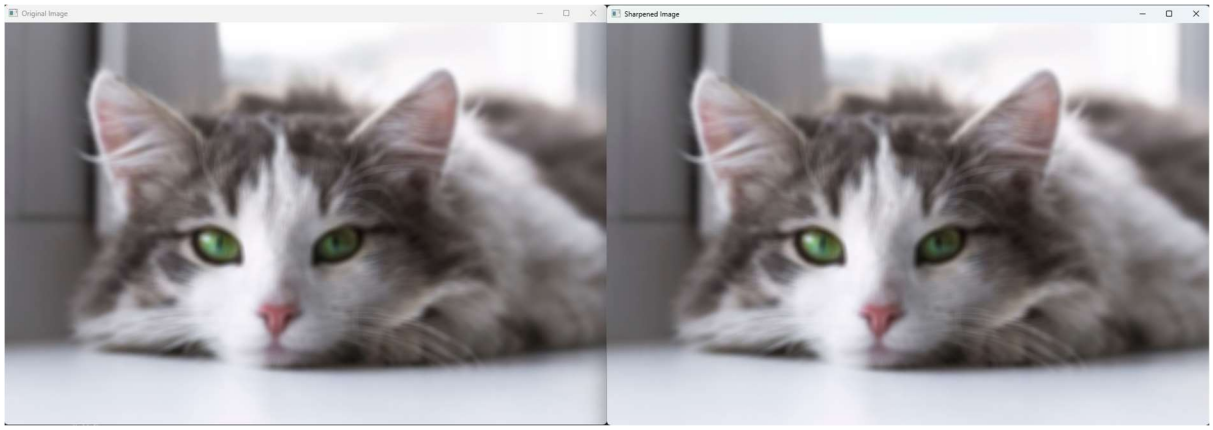


Рисунок 2.17. 2 вікна із оригінальним зображенням та застосуванням контрольованого маскування країв

Як можна помітити, різниця між зображеннями не дуже помітна: деякі краї стали дещо чіткішими. Такий результат прогнозований, адже для цей підхід краще підходить для зображень із плоским фоном.

Поекспериментуйте з різними варіантами значень ядра та ваг маскування, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.2.5. CLAHE + м'який USM*

Ідея даного методу полягає у піднятті локального контрасту (CLAHE), щоб потім накласти легке розмиття USM. При застосуванні даного методу, ми можемо отримати

кращу детальність без сильних ореолів. Проте, можливе також і отримання шуму в темних ділянках. Найкраще застосувати даний метод для нічних/контрових кадрів або зображень із слабким мікроконтрастом.

Використання даного методу передбачає виконання наступних кроків:

1. Перехід у колірний простір LAB використовуючи метод *cv2.cvtColor(img, cv2.COLOR_BGR2LAB)* та отриманням виключно значення яскравості L. У цьому колірному просторі, яскравість відділена від кольору (A, B), тому можна піднімати контраст/різкість без кольорових ореолів.
2. Локальне вирівнювання гістограми CLAHE використовуючи метод *cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8)).apply(L)*. CLAHE (Contrast Limited Adaptive Histogram Equalization) підвищує локальний контраст, обмежуючи перетягання шуму. При цьому ми використовуємо значення для параметрів: *clipLimit=2.0* для обмеження посилення контрасту в кожній області, при якому значення 2.0 буде відповідати за помітне підсилення локального контрасту без зриву у шум; *tileGridSize=(8,8)* – що представляє собою сітку локальних областей, у яких робиться рівняння гістограми, про якому значення 8,8 – буде відповідати збалансованому розміру для більшості фотографій, інакше кажучи середньому значенню між його діапазоном (4,4 – 8,8).
3. Повернення до BGR. Для цього ми об’єднуємо зображення із вирівняною гістограмою із значеннями кольору (A, B) - *cv2.merge([clahe, A, B])* та перетворення зображення у колірний простір BGR - *cv2.cvtColor(lab2, cv2.COLOR_LAB2BGR)*. При цьому ми отримуємо кольорове зображення з оновленою яскравістю (краще локальний контраст), готове до різкості.
4. Об’єднання зображення контурів з оригінальним зображенням. Для цього використовуємо такий же метод, який ми використовувати для USM типу із Гаусовим розмиттям *cv2.addWeighted(img, 3.6, cv2.GaussianBlur(img,(3,3),1), -2.6, 0)*. Це USM (Unsharp Masking), але в м’якій формі. Результатом буде більш різке зображення, проте ще не накладене на маску.
5. Умовне накладання різкості лише на краї. При цьому значення *mask[...,None]* буде додавати вимір, щоб розмір маски *(H×W×1)* відповідав кольоровому зображенню *(H×W×3)*. Значення *np.where(condition, X, Y)* буде брати піксель із *sharp*, якщо умова істинна (край), або з *img*, якщо фон. Результатом буде контрольоване (edge-aware) підвищення чіткості, а саме деталі підсилюються, фон залишається гладким.

В результаті ми маємо отримати наступний код (Рис. 2.18).

```
img = cv2.imread('cat_blurred.jpg')
lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
L, A, B = cv2.split(lab)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8)).apply(L)
lab2 = cv2.merge([clahe, A, B])
img2 = cv2.cvtColor(lab2, cv2.COLOR_LAB2BGR)
sharpen = cv2.addWeighted(img2, 3.6, cv2.GaussianBlur(img2,(3,3),0.8), -2.6, 0)
cv2.imshow("Original Image", img)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.18. Приклад коду для застосуванням методу вирівнювання гістограми CLAHE

Для накладання маски, ми використали ті ж самі значення, що і для попереднього випадку та USM, а саме 3.6 та -2.6.

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті застосування поточного підходу (Рис. 2.19).

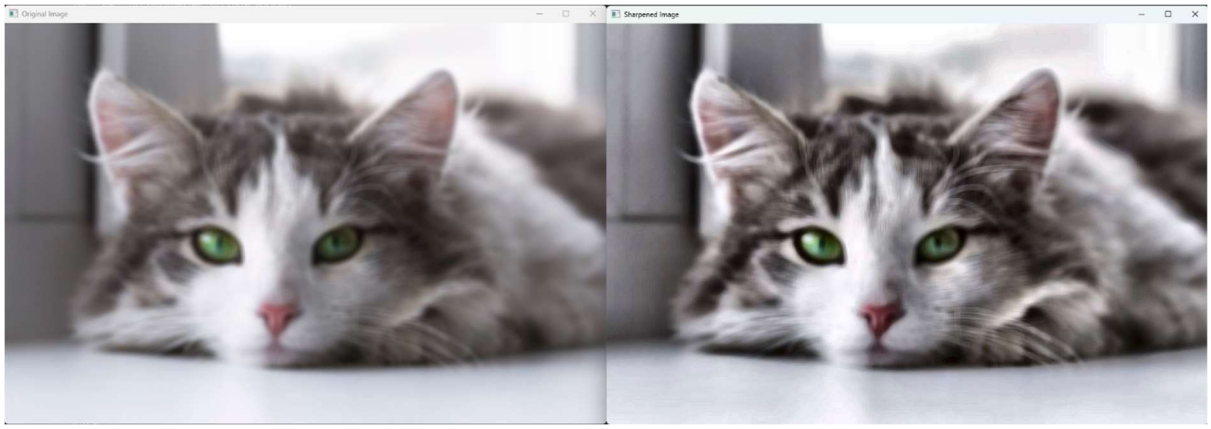

Рисунок 2.19. 2 вікна із оригінальним зображенням та застосуванням методу вирівнювання гістограми CLAHE

Як можна помітити, різниця між зображеннями дуже помітна: краї стали дещо чіткішими, як і змінився в цілому контраст зображення.

Поекспериментуйте з різними варіантами значень ядра та ваг маскування, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

*2.2.6. Деталізоване покращення*

Ідея даного методу полягає у внутрішньому поєднанні edge-preserving обробки та підсилення дрібних деталей. Часто його ще називають «one-liner» для деталей, адже це готовий високорівневий метод OpenCV для підвищення локальної чіткості та контрасту, який базується на edge-preserving фільтрації (збереженні країв). Цей метод є дуже простим, що і є його основною перевагою. Проте, як результат цей метод володіє і

обмеженою кількістю можливих налаштувань. Найчастіше його використовують для природних сцен, пейзажів.

Даний метод приймає наступні параметри:

- вхідне зображення;
- радіус просторового згладжування — наскільки далеко враховуються сусідні пікселі; якщо більше, то буде сильніше згладження, ширші області впливу у діапазоні 0-200. Будемо використовувати значення 10, яке має забезпечити помірне розширення;
- коефіцієнт чутливості до різниці кольору (intensity range); чим менше значення, тим сильніше збереження країв, чим більше значення, тим буде більше застосовуватися розмиття навіть між різними кольорами. Має значення у діапазоні 0-1. Будемо використовувати значення 0.15, яке має забезпечити помірне підсилення текстур;

Для кращого вибору значень параметрів, також можна скористатися наступною таблицею:

| Сценарій | sigma_s | sigma_r | Результат |
|---|---|---|---|
| Портрети / люди | 5–10 | 0.1–0.2 | м’яке згладження шкіри, природні краї |
| Пейзажі / архітектура | 10–20 | 0.1–0.3 | підкреслення дрібних деталей трави, листя, каменю |
| Технічні сцени / текстури | 20–40 | 0.2–0.4 | сильне підсилення структури, схоже на HDR |
| М’яке "покращення фото" | 10 | 0.15 | стандартний баланс (варіант з прикладу) |

Для застосування даного методу ми будемо використовувати наступний код (Рис. 2.20).

```
img = cv2.imread('cat_blurred.jpg')
sharpen = cv2.detailEnhance(img, sigma_s=10, sigma_r=0.15)
cv2.imshow("Original Image", img)
cv2.imshow("Sharpened Image", sharpen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 2.20. Приклад коду для застосуванням готового методу OpenCV detailEnhance

У результаті виконання даного коду ми отримаємо два вікна із оригінальним зображенням та тим, яке ми отримали в результаті застосування поточного підходу (Рис. 2.21).

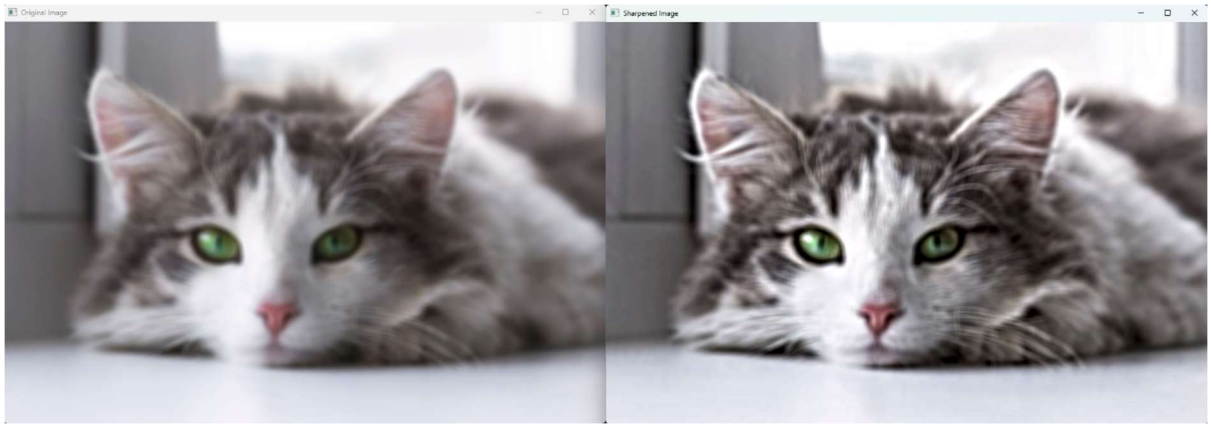


Рисунок 2.21. 2 вікна із оригінальним зображенням та застосуванням готового методу OpenCV detailEnhance

Як можна помітити, результат дуже схожий із попереднім результатом, а різниця між зображеннями також дуже помітна: краї стали дещо чіткішими, як і змінився в цілому контраст зображення. Проте, також можна помітити і збільшення шуму на фоні зображення.

Поекспериментуйте з  різними значеннями радіусу просторового згладжування та коефіцієнту чутливості до різниці кольору, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

# 3. Виділення контурів

## 3.1. Порогове виділення (threshold)

Порогове виділення використовується для спрощення зображення, коли потрібно чітко відокремити об’єкти від фону. Його головна перевага полягає у простоті та швидкості — метод не потребує складних обчислень і ефективно працює для зображень із рівномірним освітленням або контрастним фоном. Він добре підходить для технічних задач, де об’єкти мають чіткі границі, наприклад для підрахунку деталей на конвеєрі, аналізу біомедичних знімків, розпізнавання символів чи тексту. Недоліком є залежність від освітлення та якості зображення: якщо освітлення нерівномірне, фон має тіні або градієнти, метод може виділити об’єкти неправильно. Крім того, порогове виділення погано справляється з природними сценами, де різниця між об’єктом і фоном невиразна.

Даний підхід краще використовувати для простих, контрастних зображень або як попередній етап обробки, коли потрібно відокремити об’єкт від фону перед подальшим аналізом.

Щоб застосувати даний підхід до зображення потрібно скористатися методом *cv2.threshold*, який приймає наступні параметри:

- вхідне зображення (зазвичай у відтінках сірого);
- поріг (0–255), який визначає, де відсікати;
- яке значення присвоювати білим пікселям (зазвичай 255);

- тип порогового методу.

Тип порогів може мати наступні значення:

- *cv2.THRESH_BINARY*, має діапазон 0-255, якщо величина значення пікселя більше за порогове, то приймати 255, інакше 0;
- *cv2.THRESH_BINARY_INV*, має діапазон 255-0, протилежний тип до попереднього;
- *cv2.THRESH_TRUNC*, приймається коли значення менше або рівне пороговому значенню;
- *cv2.THRESH_TOZERO*, приймається 0 якщо значення нижче порогового, якщо вище, то залишає оригінал;
- *cv2.THRESH_OTSU*, приймає автоматично порогове значення, яке визначається автоматично з гістограми;

Для нашого прикладу приймаємо порогове значення рівне 127, адже це середнє значення, при якому кожен піксель буде порівнюватися з порогом 127. Якщо інтенсивність більша за 127, то значення пікселя встановлюється в 255 (білий). Якщо інтенсивність менша або рівна 127, то значення пікселя приймається 0 (чорний).

Для застосування даного методу ми будемо використовувати наступний код (Рис. 3.1).

```
img = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)
_, thresh = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)
cv2.imshow("Threshold", thresh)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 3.1. Приклад коду для застосуванням порогового виділення threshold

У результаті виконання даного коду ми отримаємо вікно із зображенням, на якому виділено межі об’єктів (Рис. 3.2).


Рисунок 3.2. Вікно із зображенням після застосування порогове виділення

Поекспериментуйте з різними значеннями порогового виділення та типів порогів, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

### 3.2. Виділення контурів за допомогою Canny

Виявлення контурів за допомогою алгоритму Canny застосовується тоді, коли потрібно знайти точні межі об'єктів або їхню форму. Його перевага — висока точність і стійкість до шумів завдяки вбудованому попередньому згладженню та подвійній системі порогів. Canny дозволяє отримати тонкі, чіткі лінії контурів, що особливо корисно для аналізу геометрії, виявлення об'єктів у складних сценах або підготовки даних для подальших етапів — наприклад, побудови контурів, визначення кутів чи сегментації за формою. Недолік цього методу — чутливість до вибору порогів: при надто низьких значеннях з'являється шум, при надто високих — частина контурів може втратитися. Також він потребує більше часу на обчислення, ніж просте порогове виділення.

Алгоритм Canny ефективніший тоді, коли важливо зберегти точність меж, виявити форму або структуру об'єкта, особливо у складних або природних сценах.

Алгоритм складається з кількох етапів:

- гаусове згладжування (для придушення шуму);
- визначення градієнтів (за напрямом і величиною);
- нонмаксимальне придушення (залишаються лише локальні максимуми);
- подвійний поріг і гістерезис:
    - якщо градієнт > верхнього порогу → піксель = край;
    - якщо між порогами → край, якщо з'єднаний зі "сильним" краєм.

Щоб застосувати даний підхід до зображення потрібно скористатися методом *cv2.Canny*, який приймає наступні параметри:

• вхідне зображення (зазвичай у відтінках сірого);

• нижній поріг (наприклад, 100) — все що менше, вважається шумом;

• верхній поріг (наприклад, 200) — все що вище, вважається сильним краєм.

Типове правило для вибору значення порогів наступне:

$$\text{threshold\_top} \approx 2 \times \text{threshold\_bottom}$$

Також, для вибору оптимальних значень порогу можна скористатися наступною таблицею:

| Тип зображення | Параметри |
|---|---|
| Чітке, контрастне | (100, 200) |
| Темне, з шумом | (50, 150) |
| Дуже світле | (150, 250) |
| Автоматично (через Otsu) | комбінують з cv2.threshold |

Для застосування даного методу ми будемо використовувати наступний код (Рис. 3.3).

```
img = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)
edges = cv2.Canny(img, 100, 200)
cv2.imshow("Canny edges", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 3.3. Приклад коду для застосуванням порогового виділення threshold

У результаті виконання даного коду ми отримаємо вікно із зображенням, на якому виділено межі об’єктів (Рис. 3.4).


Рисунок 3.4. Вікно із зображенням після застосування виділення контурів із Canny

Поекспериментуйте з різними значеннями порогового виділення, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації.

### 3.3. Виділення контурів за допомогою Canny

Дуже часто ці два методи використовують разом для отримання кращого результату: спочатку застосовують порогове виділення для спрощення зображення, а потім — Canny для точного окреслення контурів.

Для цього прикладу ми використаємо тип порогу *cv2.THRESH_BINARY + cv2.THRESH_OTSU* для порогового виділення *threshold* для автоматичного та оптимального значення порогу між фоном і об’єктами — особливо ефективно, якщо фон рівномірний, а освітлення стабільне. Оскільки тип автоматичний, вказуємо відповідно мінімальне та максимальне значення порогів як 0 та 255.

Для застосування *Canny* вибираємо 50 та 150, що представляє собою нижній поріг для слабких градієнтів та верхній поріг для сильних градієнтів відповідно. Зв’язок між ними типовий: верхній ≈ 3× нижнього або 2–3× нижчого. Така пропорція забезпечує збалансоване виявлення країв: слабкі краї, пов’язані із сильними, залишаються; шумові точки нижче порогу — відкидаються. 50 — досить низьке значення, щоб не втратити тонкі контури; 150 — обмежує надлишкові “фальшиві” краї, утримуючи лише суттєві переходи. Цей діапазон (50–150) є найчастіше вживаним базовим набором у практиці, бо він добре працює для зображень середнього контрасту.

Для застосування даного підходу ми будемо використовувати наступний код (Рис. 3.5).

```
gray = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)
_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
edges = cv2.Canny(thresh, 50, 150)
cv2.imshow("Canny edges", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Рисунок 3.5. Приклад коду для застосуванням комбінації порогового виділення threshold та виділення контурів Canny

У результаті виконання даного коду ми отримаємо вікно із зображенням, на якому виділено межі об’єктів (Рис. 3.6).


Рисунок 3.6. Вікно із зображенням після застосуванняv комбінації порогового виділення threshold та виділення контурів Canny

Поекспериментуйте з  різними значеннями порогового виділення, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації. Чи є відмінність від застосування виділення контурів Canny та поєднання його із пороговим виділенням *threshold*?

## 4. Реалізація згортки у numpy та порівняння з готовими функціями OpenCV

Згортка (convolution) є базовою операцією в цифровій обробці зображень, яка використовується для виділення певних ознак, зміни контрасту, згладжування або підсилення деталей. Її суть полягає в тому, що кожен піксель нового зображення обчислюється як зважена сума сусідніх пікселів у вихідному зображенні, де ваги задає спеціальна матриця — ядро (kernel або фільтр). Така операція дозволяє локально аналізувати зображення, реагуючи на зміни яскравості, кольору чи текстури в певних ділянках.

Згортку використовують для виконання великої кількості завдань: від базових, як-от розмиття, підвищення різкості, виявлення країв (оператори Собеля, Лапласіана), до більш складних, таких як виділення текстур, підготовка даних для сегментації чи фільтрація шуму. Вона лежить в основі сучасних методів комп’ютерного зору й нейронних мереж, зокрема згорткових нейронних мереж (CNN), де ядра навчаються автоматично.

Основна перевага згортки — її універсальність. Використовуючи різні ядра, можна досягти найрізноманітніших ефектів: від розмиття до підсилення контрасту. Крім того, вона добре інтерпретується — можна візуально побачити, як певне ядро реагує на напрямні або структури в зображенні. Однак у чистій реалізації на Python/NumPy згортка має серйозний недолік — повільну роботу, оскільки вона виконується через вкладені цикли, а не оптимізовані обчислення. Також результат згортки може залежати

від вибору розміру ядра, типу заповнення країв (padding) і масштабу значень, що потребує точного налаштування.

Для реалізації згортки у Python, ми будемо використовувати бібліотеку *NumPy* та для відображення результатів *matplotlib*. Інсталюйте дані пакети даними командами:

```
pip install numpy
pip install matplotlib
```

У всіх наступних прикладах ми будемо використовувати наступну функцію для реалізації згортки в Python (Рис. 4.1).

```python
def conv2d_same(img, kernel):
    kh, kw = kernel.shape
    pad_h, pad_w = kh//2, kw//2
    ker = np.flipud(np.fliplr(kernel))
    x = np.pad(img, ((pad_h,pad_h),(pad_w,pad_w)), mode='reflect')
    out = np.empty_like(img, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            roi = x[i:i+kh, j:j+kw]
            out[i,j] = np.sum(roi * ker)
    return out
```

Рисунок 4.1. Приклад коду для застосуванням комбінації порогового виділення threshold та виділення контурів Canny

Дана функція приймає два параметри:

- img — зображення (2D-масив у градаціях сірого);
- kernel — ядро фільтра (2D матриця коефіцієнтів).

На першому рядку методу, ми зберігаємо висоту (*kh*) та ширину (*kw*) ядра. Це потрібно, щоб знати, скільки сусідів враховувати навколо кожного пікселя.

Далі, ми розраховуємо розмір відступу (padding) з кожного боку, щоб результат мав такий самий розмір, як оригінальне зображення (same padding). Наприклад, для ядра 3×3, *pad_h* та *pad_w* буде дорівнювати 1.

На наступному рядку, ми повертаємо ядро на 180°. Це важливо, тому що математична згортка передбачає перевертання фільтра (на відміну від кореляції, яку використовує cv2.filter2D за замовчуванням). Якщо ядро симетричне (наприклад, гаусове), ефект буде однаковий.

Далі, ми додаємо рамку навколо зображення, щоб можна було обробити краї. При цьому використовуємо *mode='reflect'*, який означає, що пікселі віддзеркалюються за межами зображення — це дозволяє уникнути чорних рамок та втрати інформації на краях.

Наступною операцією є створення порожньої матриці результату (*out*), того ж розміру, що й *img*. Тип *float32* вибраний для точності при обчисленнях.

Далі, ми маємо два вкладені цикли проходять кожен піксель зображення, де i — рядок, j — стовпець у матриці зображення.

У даних циклах, ми спочатку вирізаємо локальну область (Region Of Interest) того ж розміру, що й ядро. Це ті пікселі, які впливають на поточне значення результату:

```
roi = x[i:i+kh, j:j+kw]
```

Далі, ми множимо локальну область *roi* на ядро *ker* (попіксельно) та обчислюємо суму добутків — це і є значення нового пікселя у вихідному зображенні. Іншими словами, виконуємо дане рівняння:

$$out(i,j) = \sum_{m,n} img(i+m, j+n) \cdot kernel(m,n)$$

І останнім рядком повертаємо результат – нове зображення після згортки, де пікселі мають нові значення, розраховані на основі сусідніх. Виконуємо її для фільтрації, покращення або аналізу зображення, зокрема для розмиття, виявлення країв, підвищення різкості чи підготовки до комп'ютерного аналізу.

### 4.1. Згортка в NumPy

*4.1.1. Розмиття згорткою*

Спочатку ми протестуємо розмиття зображення, використовуючи згортку. Для цього використаємо усереднювальне розмиття зображення використовуючи ядро, у якому всі елементи мають однакову вагу.

Для цього ми спочатку вказуємо розмір ядра:

```
k = 5
```

Це означає, що для кожного пікселя програма братиме його сусідів у квадраті 5×5 і обчислюватиме середнє значення. Чим більше *k*, тим сильніше розмиття (менше деталей і м'якіший вигляд).

Далі, ми створюємо ядро згортки (Box filter):

```
box = np.ones((k, k), dtype=np.float32) / (k*k)
```

Даний код створює матрицю з усіх одиниць. Таким чином, ядро усереднює значення сусідніх пікселів, замінюючи поточний піксель середнім яскравості його оточення.

Наступний код власне застосовує згортку до зображення, використовуючи ядро:

```
blur_box = conv2d_same(img, box)
```

Останньою операцією буде обмеження діапазону та типу перетворення:

```
blur_box = np.clip(blur_box, 0, 255).astype(np.uint8)
```

Це необхідно, тому що під час обчислень згортки результат зберігається у float32, і значення можуть трохи вийти за межі через округлення.

Використовуючи пакет *plt* ми візуалізуємо отриманий результат.

Повна версія коду для усередненого розмиття зображення згорткою подано на Рис. 4.2.

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

img = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)

def conv2d_same(img, kernel):
    kh, kw = kernel.shape
    pad_h, pad_w = kh//2, kw//2
    ker = np.flipud(np.fliplr(kernel))
    x = np.pad(img, ((pad_h,pad_h),(pad_w,pad_w)), mode='reflect')
    out = np.empty_like(img, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            roi = x[i:i+kh, j:j+kw]
            out[i,j] = np.sum(roi * ker)
    return out

k = 5
box = np.ones((k, k), dtype=np.float32) / (k*k)
blur_box = conv2d_same(img, box)
blur_box = np.clip(blur_box, 0, 255).astype(np.uint8)

plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title('Оригінал')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(blur_box, cmap='gray')
plt.title('Box Blur (усереднення)')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Рисунок 4.2. Повна версія коду для усередненого розмиття зображення згорткою

У результаті виконання даного коду ми отримаємо вікно із двома зображеннями: перше – оригінал у градації сірого, друге – з усередненим розмиттям згорткою (Рис. 4.3).


Рисунок 4.3. Вікно із оригінальним зображенням та зображення із усередненим розмиттям згорткою

Поекспериментуйте з різними значеннями ядра згортки, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому

файлі, щоб мати змогу порівняти ефект різних типів фільтрації. Порівняйте дане зображення із попередніми зображеннями із розмиттям, для яких використовувався OpenCv. Яке розмиття вам більше подобається та чому?

*4.1.2. Підвищення чіткості згорткою*

Наступним прикладом використання згортки буде підвищення різкості. Для цього використаємо класичне ядро «sharpen». Функція згортки та візуалізації результатів залишиться незмінною. Для підвищення чіткості використаємо наступний код:

```
sharpen_kernel = np.array([[ 0, -1,  0],[-1,  5, -1],[ 0, -1,  0]],
                          dtype=np.float32)
sharp_simple = conv2d_same(img, sharpen_kernel)
sharp_simple = np.clip(sharp_simple, 0, 255).astype(np.uint8)
```

Першим рядком коду ми створюємо ядро згортки для підвищення різкості, в якому вказуємо матрицю для ядра фільтра. Центральний елемент 5 означає, що поточний піксель зберігається із підвищеною вагою. Навколишні пікселі з коефіцієнтом -1 віднімаються — це приглушує фон і підсилює контраст між сусідніми пікселями. Сума всіх коефіцієнтів ядра ≈ 1, тому загальна яскравість зображення зберігається. Тобто — центральний піксель “посилюється”, а сусіди “віднімаються”, тому краї стають помітнішими, а зображення — різкішим.

Наступним рядком ми застосовуємо згортку до нашого зображення використовуючи попередньо створене ядро.

І останнім рядком, як і в попередньому прикладі ми обмежуємо діапазон значень та приводимо результуючий масив до типу зображення.

Повна версія коду для підвищення різкості згорткою подано на Рис. 4.4.

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

img = cv2.imread('cat_blurred.jpg', cv2.IMREAD_GRAYSCALE)

def conv2d_same(img, kernel):
    kh, kw = kernel.shape
    pad_h, pad_w = kh//2, kw//2
    ker = np.flipud(np.fliplr(kernel))
    x = np.pad(img, ((pad_h,pad_h),(pad_w,pad_w)), mode='reflect')
    out = np.empty_like(img, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            roi = x[i:i+kh, j:j+kw]
            out[i,j] = np.sum(roi * ker)
    return out

sharpen_kernel = np.array([[ 0, -1,  0],
                           [-1,  5, -1],
                           [ 0, -1,  0]], dtype=np.float32)
sharp_simple = conv2d_same(img, sharpen_kernel)
sharp_simple = np.clip(sharp_simple, 0, 255).astype(np.uint8)

plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title('Оригінал')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(sharp_simple, cmap='gray')
plt.title('Підвищення різкості sharpen')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Рисунок 4.4. Повна версія коду для підвищення різкості зображення згорткою

У результаті виконання даного коду ми отримаємо вікно із двома зображеннями: перше – оригінал у градації сірого, друге – з підвищено різкістю після використання згортки (Рис. 4.5). Зверніть увагу, що для даного прикладу ми використовували попередньо розмите зображення.

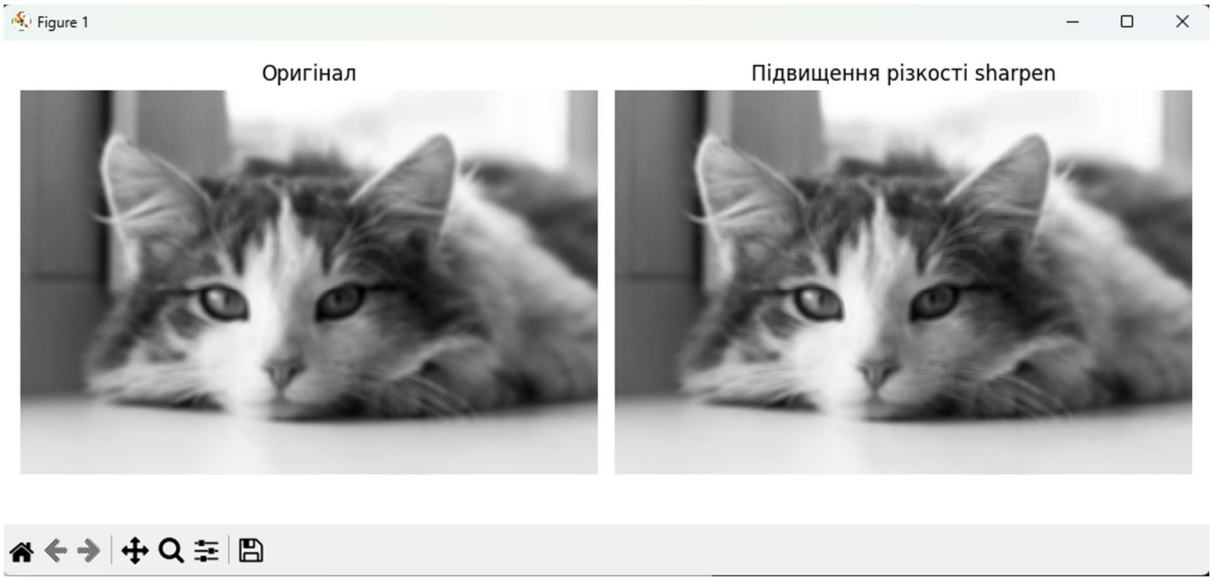


Рисунок 4.5. Вікно із оригінальним зображенням та зображення із застосованим ефектом підвищення різкості згорткою

Поекспериментуйте з різними значеннями ядра згортки, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації. Порівняйте дане зображення із попередніми зображеннями із підвищенням різкості, для яких використовувався OpenCv. Який підхід для підвищення різкості дав найкращий на вашу думку результат?

*4.1.3. Виділення контурів згорткою*

Наступним прикладом використання згортки буде виділення контурів. Для цього використаємо оператор Собеля. Оператор Собеля — це спеціальний фільтр згортки, який використовується для виявлення країв у зображенні, тобто ділянок, де різко змінюється яскравість. Його мета — оцінити градієнт яскравості у двох напрямках (по горизонталі й вертикалі) та знайти місця найбільших змін. В результаті він показує, де у зображенні знаходяться контури — межі між об’єктами.

Функція згортки та візуалізації результатів залишиться незмінною. Для виділення контурів згорткою, ми використаємо наступний код:

```
Sx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)
Sy = np.array([[-1,-2,-1], [ 0, 0, 0], [ 1, 2, 1]], dtype=np.float32)
                    gx_np = conv2d_same(img, Sx)
                    gy_np = conv2d_same(img, Sy)
                  mag_np = np.hypot(gx_np, gy_np)
        mag_np = (mag_np / mag_np.max() * 255).astype(np.uint8)
```

Спочатку ми маємо створити ядро для оператора Собеля по осям X та Y. Значення ядер Собеля (−1, 0, 1 і –1, –2, –1) обрані так, щоб оператор одночасно знаходив напрям

зміни яскравості та приглушував шум, тобто забезпечував чітке і стабільне виявлення країв.

Далі ми обчислюємо вертикальні градієнти для кожної осі використовуючи функцію згортки.

Наступним рядком ми обчислюємо модуль градієнта, тобто наскільки різко змінюється яскравість у будь-якому напрямку. Таким чином, якщо *gx_np* і *gy_np* показують зміни окремо по X і Y, то *mag_np* показує загальну інтенсивність країв незалежно від напрямку.

І останнім кроком – нормалізуємо дані та приводимо їх до формату зображення.

Повна версія коду для виділення країв згорткою подано на Рис. 4.6.

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

img = cv2.imread('cat.jpg', cv2.IMREAD_GRAYSCALE)

Sx = np.array([[-1, 0, 1],
               [-2, 0, 2],
               [-1, 0, 1]], dtype=np.float32)
Sy = np.array([[-1,-2,-1],
               [ 0, 0, 0],
               [ 1, 2, 1]], dtype=np.float32)

def conv2d_same(img, kernel):
    kh, kw = kernel.shape
    pad_h, pad_w = kh//2, kw//2
    ker = np.flipud(np.fliplr(kernel))
    x = np.pad(img, ((pad_h,pad_h),(pad_w,pad_w)), mode='reflect')
    out = np.empty_like(img, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            roi = x[i:i+kh, j:j+kw]
            out[i,j] = np.sum(roi * ker)
    return out

gx_np = conv2d_same(img, Sx)
gy_np = conv2d_same(img, Sy)
mag_np = np.hypot(gx_np, gy_np)
mag_np = (mag_np / mag_np.max() * 255).astype(np.uint8)

plt.figure(figsize=(12,7))
plt.subplot(2,3,1); plt.imshow(img, cmap='gray'); plt.title('Original'); plt.axis('off')
plt.subplot(2,3,2); plt.imshow(gx_np, cmap='gray'); plt.title('NumPy Sobel X'); plt.axis('off')
plt.subplot(2,3,3); plt.imshow(gy_np, cmap='gray'); plt.title('NumPy Sobel Y'); plt.axis('off')
plt.subplot(2,3,4); plt.imshow(mag_np, cmap='gray'); plt.title('NumPy |grad|'); plt.axis('off')
plt.tight_layout(); plt.show()
```

Рисунок 4.6. Повна версія коду для виділення країв згорткою

У результаті виконання даного коду ми отримаємо вікно із чотирьома зображеннями: перше – оригінал у градації сірого, друге та третє – із обчисленими вертикальними градієнтами для осі X та Y, четверте – зображення із виділеними краями згорткою (Рис. 4.6).

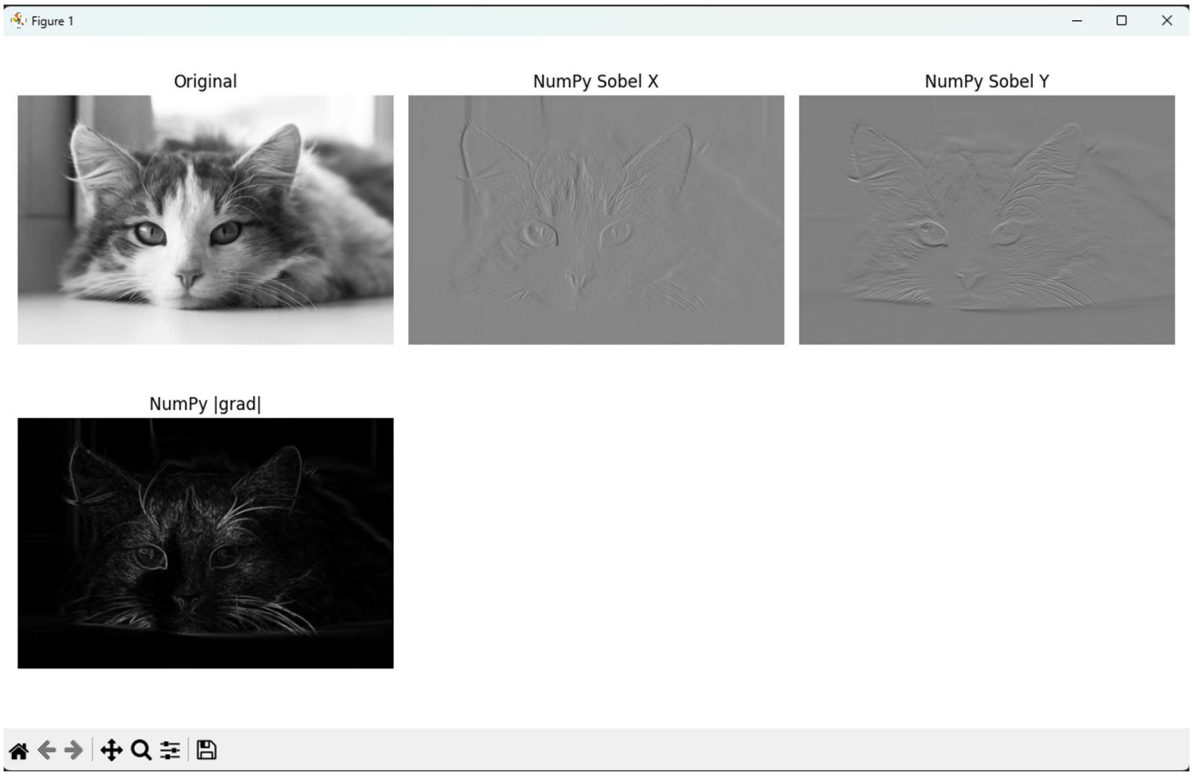


Рисунок 4.6. Вікно із отриманими зображеннями після застосування виділення країв згорткою

Поекспериментуйте з різними значеннями ядра згортки Собеля, щоб порівняти якими будуть результати від застосування різних значень. Збережіть результат у окремому файлі, щоб мати змогу порівняти ефект різних типів фільтрації. Порівняйте дане зображення із попередніми зображеннями для виділення країв, для яких використовувався OpenCv. Який підхід для підвищення різкості дав найкращий на вашу думку результат?