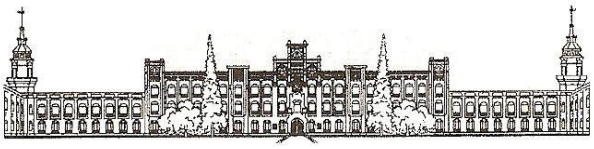


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Лабораторна робота №1

з дисципліни «Комп'ютерний зір»

«Базова обробка зображень»

Виконали студентки групи: КВ-11

ПІБ: Михайліченко Софія Віталіївна
Шевчук Ярослава Олегівна

Перевірила: _____

Київ 2024

Постановка задачі:

Реалізувати наведені нижче завдання у вигляді відповідного застосунку/застосунків, що передбачають завантаження зображень із файлів, їх обробку відповідним чином та візуалізацію і збереження файлів результату (рекомендовані мови програмування: C++ або Python, але за бажання припустимі і інші).

Для роботи із зображеннями рекомендовано використовувати бібліотеку OpenCV (зокрема, cv::Mat, cv::imread, cv::imwrite, cv::imshow, cv::waitKey), хоча, за бажання припустимо використовувати і інші бібліотеки, що дозволяють роботу з зображеннями (завантаження, збереження, візуалізація, доступ до значень пікселів). Завдання мають бути виконані самостійно, не використовуючи існуючі реалізації відповідних алгоритмів (в т.ч. реалізацію операції згортки) з бібліотеки OpenCV або інших бібліотек.

Бажано забезпечити автоматичне налаштування проекту із його залежностями (наприклад, через відповідний сценарій CMake, або відповідно сформований файл requirements.txt системи керування пакунками pip).

1. Реалізувати процедуру вирівнювання гістограми півтонового зображення.

2. Реалізувати процедуру умовного масштабування півтонового зображення (приведення значень середнього та дисперсії одного зображення до значень дисперсії та середнього іншого зображення).

3. Реалізувати процедуру застосування до півтонового зображення довільного лінійного локального оператора (заданого ядром фільтра).

4. Реалізувати та порівняти процедури згладжування зображень (видалення шуму) за допомогою прямокутного фільтра, медіанного фільтра, фільтра Гаусса та сигма-фільтра (спробувати різні комбінації параметрів, підібрати оптимальні для тестових зображень). При тестуванні бажано брати зображення, що містять помітний шум (або додати з цією метою випадковий шум до незашумлених зображень).

5. Реалізувати підвищення різкості зображень за допомогою нерізкого маскування (спробувати різні комбінації параметрів, в т.ч. різні операції згладжування, підібрати оптимальні для тестових зображень). При тестуванні брати розмиті зображення.

6. Реалізувати та порівняти детектори границь для півтонового зображенні на основі дискретних похідних, операторів Собеля, Шарра, Лапласса, та оператора Кенні (спробувати різні значення порогів та підібрати оптимальні для вхідних зображень). При тестуванні брати зображення, що містять границі.

7. Реалізувати та порівняти детектори кутів Гарріса та FAST (підібрати оптимальне значення параметрів, в т.ч. кількість пікселів в критерії наявності кута). При тестуванні брати зображення, що містять кути. Візуалізувати знайдені кути на зображеннях.

8. Реалізувати та порівняти детектори границь LoG та DoG. При тестуванні

рати зображення, що містять границі.

Додаткові завдання (за бажанням):

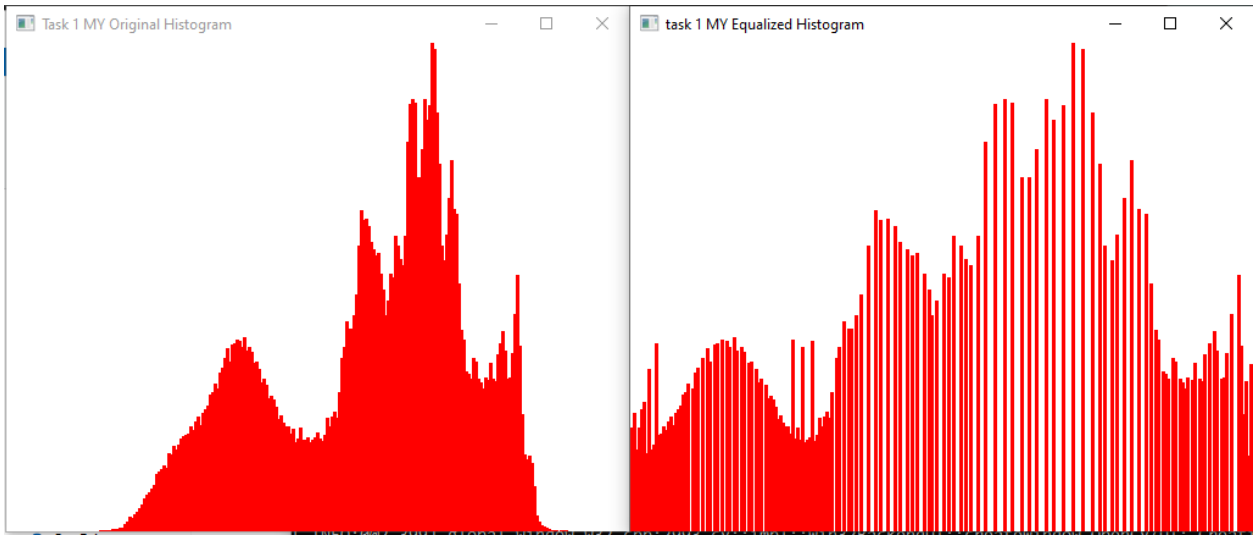
1. Де це можливо, виконати порівняння результатів роботи функцій, реалізованих самостійно, із результатами роботи відповідних функцій бібліотеки OpenCV (зокрема, cv::equalizeHist, cv::blur, cv::GaussianBlur, cv::medianBlur, cv::Sobel, cv::Sharr, cv:: Laplacian, cv:Canny, cv::cornerHarris, cv::FastFeatureDetector).

Порядок виконання роботи

Завдання 1

Реалізувати процедуру вирівнювання гістограми півтонового зображення.

Результати виконання:



Вирівнювання гістограми дозволяє покращити контраст зображення, особливо, якщо у вихідному зображенні пікселі зосереджені в одній області інтенсивностей. Як бачимо, у нашому прикладі на кінцях гістограми не має значень, що свідчить про те що у зображенні не домінують найсвітліші та найтемніші області інтенсивності. Це добре видно при порівнянні гістограм до і після вирівнювання. До обробки, інтенсивності пікселів були зосереджені в обмеженому діапазоні, тоді як після вирівнювання гістограма стає більш розподіленою по всьому діапазону інтенсивностей. Також деталі, які могли

бути погано видимі в темних або світлих ділянках зображення, стають краще помітними.

Порівняння з функціями OpenCV:

У результаті ми бачимо послідовно виведені: оригінальне зображення, зображення з вирівняною гістограмою, за допомогою самостійно створених функцій та зображення з вирівняною гістограмою, за допомогою вбудованих функцій. Хоча на перший погляд відмінностей не багато, але все ж вони є і досить вагомі.

Найбільш важливе це те, що власна реалізація функції є повільнішою, оскільки здійснює обхід кожного пікселя зображення без оптимізацій. Це робить її менш придатною для великих зображень. Водночас функція OpenCV оптимізована для швидкості, завдяки використанню векторизації та внутрішніх оптимізацій, що робить її значно швидшою в роботі з великими зображеннями. Одним із плюсів є те, що самостійно розроблена функція надає повний контроль над алгоритмом, що дозволяє вносити зміни або адаптувати його під специфічні задачі. З іншого боку, функція OpenCV дуже зручна у використанні для загальних випадків, але менш гнучка в тому, щоб дозволяти користувачу змінювати сам процес вирівнювання.

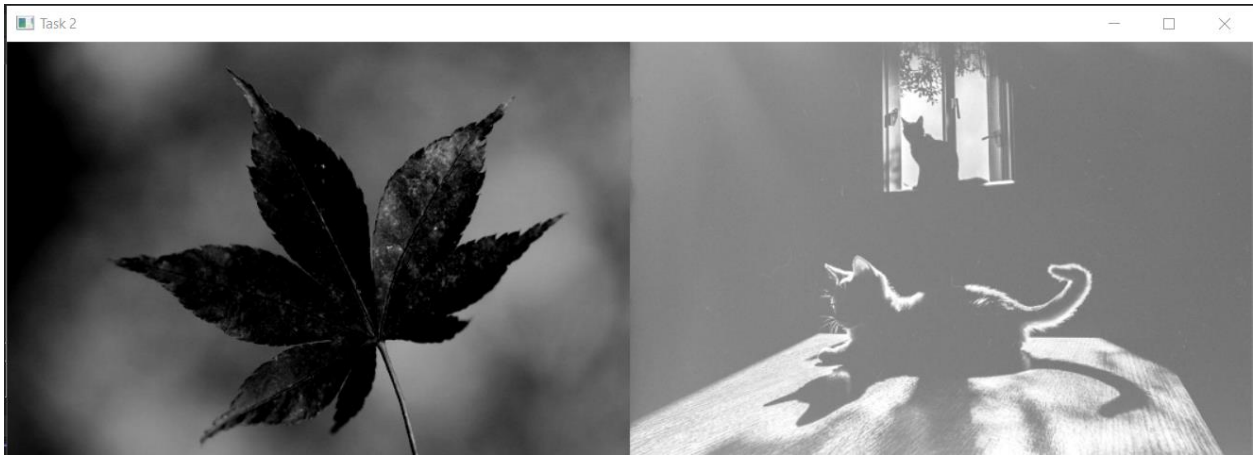
Хоча результати вирівнювання гістограми за допомогою обох функцій можуть бути візуально схожими, можуть спостерігатися деякі відмінності в контрастності, насиченості кольорів або в появі артефактів, особливо у випадку складних зображень.

Завдання 2

Реалізувати процедуру умовного масштабування півтонового зображення (приведення значень середнього та дисперсії одного зображення до значень дисперсії та середнього іншого зображення).

У цьому завданні реалізована функція, яка виконує масштабування середнього значення та стандартного відхилення двох зображень у градаціях сірого. Функція приймає два зображення у вигляді матриць, а також матрицю для збереження результату.
Спочатку функція обчислює середнє значення та стандартне відхилення для першого зображення. Ці значення зберігаються у змінних mean_img1 та std_img1. Далі, аналогічним чином, функція обчислює середнє значення та стандартне відхилення для другого зображення, які зберігаються у змінних mean_img2 та std_img2.
Після цього, перше зображення перетворюється у формат з плаваючою точкою з подвійною точністю. В результаті проводиться масштабування значень: кожен піксель першого зображення нормалізується відносно свого середнього значення та стандартного відхилення, потім значення масштабуються відповідно до стандартного відхилення другого зображення і додається середнє значення другого зображення.
На завершення, отримане зображення конвертується назад у початковий формат, щоб результат мав таку ж типізацію, як і перше зображення. Таким чином, функція дозволяє адаптувати характеристики одного зображення відповідно до характеристик іншого.

Результати виконання:



На даному зображенні показано процедуру фільтрації Фур’є. Ліворуч застосований фільтр низьких частот, який розмиває дрібні деталі. Це робить лист м'якшим і розмитим, видаляючи різкі контури. Праворуч зображення котів, оброблене фільтром високих частот. Це підкреслює контури та тіні, роблячи зображення чіткішим і контрастнішим. Фільтри допомагають акцентувати або приховати певні елементи зображення, залежно від того, які частоти пропускаються або блокуються.

Завдання 3

Реалізувати процедуру застосування до півтонового зображення довільного лінійного локального оператора (заданого ядром фільтра).

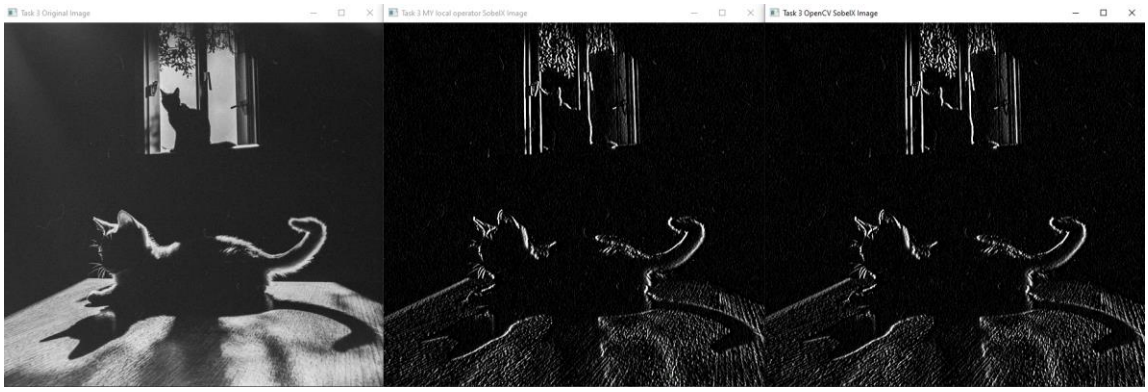
Було використано ядра, які відповідають існуючим ядрам фільтрів, щоб можна було порівняти результати.

Результати виконання:

Box розмір ядра 3x3:

Box розмір ядра 5x5:

Sobel розмір ядра 3x3:



Sobel розмір ядра 5x5:

Використання інших вагових коефіцієнтів:

Процедура застосування до півтонового зображення довільного лінійного локального оператора дозволяє ефективно обробляти зображення шляхом згортки пікселів з відповідними вагами, визначеними в ядрі фільтра. Із її допомогою ми можемо створювати власні фільтри, що адаптуються під конкретні потреби обробки. Це корисно для реалізації специфічних задач, таких як зменшення шуму, виділення контурів або покращення різкості. Хоча використання вбудованих функцій, таких як Sobel та blur, забезпечує оптимізовану та протестовану реалізацію для обробки зображень. Вони враховують різні аспекти обробки, включаючи ефективність, швидкість і точність.

Різні ядра, такі як kernel_box для розмивання і kernel_sobelX для виділення країв, ведуть до різних результатів. Наприклад, kernel_box зменшує шум і розмиває деталі, тоді як kernel_sobelX підкреслює зміни інтенсивності, що дозволяє виявити контури зображення.

Також як ми могли замітити, на результати певно мірою, впливали ваги ядер. У першому випадку з більшим ядром більше сусідніх пікселів вплине на значення кожного пікселя, що призведе до більшого розмивання. Це може допомогти зменшити шум, але також може знизити різкість деталей, зокрема дрібних об'єктів.

У операторі Собеля ж збільшення ваг для пікселів, що лежать ближче до країв, може призвести до кращого виявлення контурів, оскільки градієнти будуть більш виразними.

Завдання 4

Реалізувати та порівняти процедури згладжування зображень (видалення шуму) за допомогою прямокутного фільтра, медіанного фільтра, фільтра Гаусса та сигма-фільтра (спробувати різні комбінації параметрів, підібрати оптимальні для тестових зображень). При тестуванні бажано брати зображення, що містять помітний шум (або додати з цією метою випадковий шум до незашумлених зображень).

У цьому завданні реалізовані функції для згладжування зображень, що дозволяють видалити шум за допомогою різних фільтрів: прямокутного, медіанного, Гаусса та сигма-фільтра. Процес включає в себе додавання шуму до зображень(так як в початкових зображеннях він відсутній), а також тестування ефективності кожного фільтра з метою визначення оптимальних параметрів для зменшення шуму.

Функція addNoise приймає зображення та стандартне відхилення шуму як параметри. Вона генерує випадковий шум, використовуючи нормальний розподіл, і додає його до зображення. Це дозволяє створити зображення з помітним шумом, що є корисним для тестування згладжувальних фільтрів.

Функція BoxFilter реалізує прямокутний фільтр, що згладжує зображення шляхом обчислення середнього значення пікселів у заданій області (ядрі) навколо кожного пікселя. Вона проходить по всіх пікселях зображення, обчислює середнє значення для кожного з них, і зберігає результати в новій матриці. Це допомагає зменшити шум, але може також згладжувати деталі зображення.

Функція MedianFilter виконує медіанне згладжування. Вона збирає значення пікселів в околицях навколо кожного пікселя, сортує їх і замінює центральний піксель на медіанне значення. Цей метод особливо ефективний для видалення спайкових шумів, оскільки медіана менш чутлива до викидів, ніж середнє.

Функція GaussianFilter застосовує Гауссове згладжування, яке базується на Гауссовому розподілі. Спочатку створюється ядро фільтра, що має форму Гауссової функції, потім для кожного пікселя обчислюється взважене середнє значення, використовуючи пікселі навколо нього з відповідними вагами. Це дозволяє зберегти більш детальні риси зображення при видаленні шуму.

Функція SigmaFilter реалізує сигма-фільтр, який враховує як просторову, так і колірну інформацію. Для кожного пікселя вона розраховує ваги на основі відстані до сусідніх пікселів і різниці в значеннях пікселів, що дозволяє зберігати деталі при значному зменшенні шуму.

У процесі тестування всі ці фільтри застосовуються до зображень з доданим шумом, що дозволяє порівняти їхню ефективність. За результатами тестування підбираються оптимальні параметри для кожного фільтра, що забезпечують найкраще згладжування зображення.

Результати виконання:

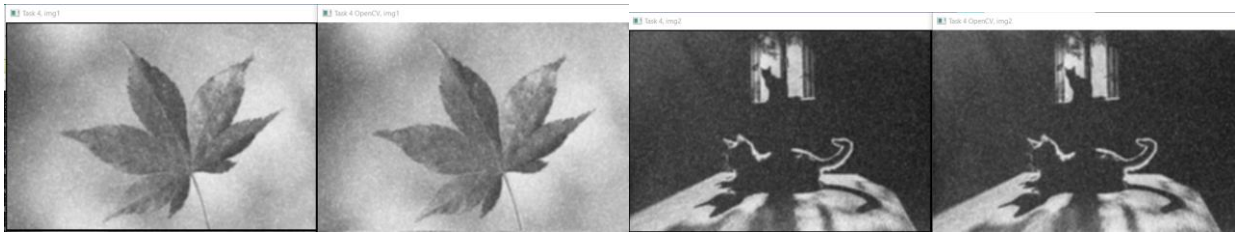


Рис.4.1 Порівняння процедури згладжування зображень
за допомогою прямокутного фільтра

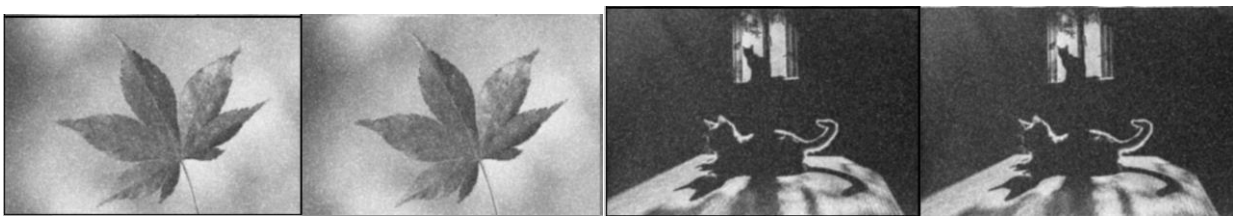


Рис.4.2 Порівняння процедури згладжування зображень
за допомогою медіанного фільтра

Рис.4.3 Порівняння процедури згладжування зображень
за допомогою фільтра Гаусса

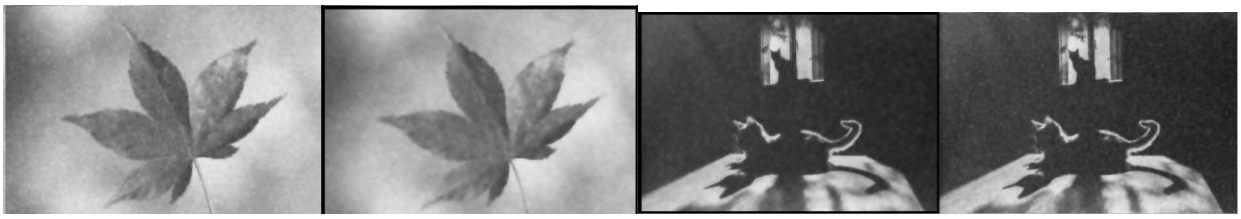


Рис.4.4 Порівняння процедури згладжування зображень
за допомогою сигма-фільтра

Порівняння з функціями OpenCV:

Реалізований прямокутний фільтр, який обчислює середнє значення пікселів у навколишній області, може бути менш ефективним у швидкості, порівняно з вбудованими функціями OpenCV, такими як cv::blur() та cv::boxFilter(), які оптимізовані для роботи з великими зображеннями завдяки використанню SIMD та інших технік. Вбудовані функції зазвичай виконуються швидше та забезпечують подібні результати при обробці, але можуть мати обмежену гнучкість у налаштуванні параметрів.

Реалізований медіанний фільтр, який сортує значення пікселів і замінює центральний піксель на медіанне значення, може бути менш швидким у порівнянні з вбудованою функцією OpenCV cv::medianBlur(), яка використовує оптимізовані алгоритми для досягнення кращої продуктивності, особливо при обробці великих зображень. Хоча власна реалізація медіанного

фільтра може надати можливість модифікації для специфічних потреб, вбудована функція забезпечує кращу швидкість виконання.

Власний Гауссовий фільтр, який створює Гауссове ядро і використовує його для обчислення ваг пікселів, має схожу логіку з функцією OpenCV cv::GaussianBlur(), але, ймовірно, буде менш оптимізованим. Вбудована функція забезпечує високу продуктивність та ефективність використання пам'яті, що робить її кращим вибором для більшості випадків.

Сигма-фільтр, реалізований вручну, враховує як просторову, так і колірну інформацію для досягнення значного зменшення шуму при збереженні деталей. Однак OpenCV не має прямої реалізації сигма-фільтра, але функція cv::bilateralFilter() може досягти подібних результатів, оскільки вона також враховує колірну інформацію та просторову близькість. Власна реалізація сигма-фільтра може бути специфічною для окремих випадків, тоді як вбудована функція є більш універсальною альтернативою, яка зазвичай працює швидше та ефективніше.

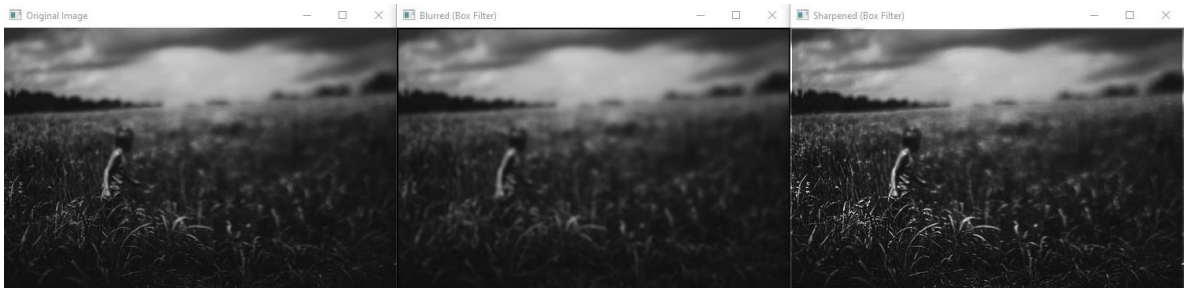
Завдання 5

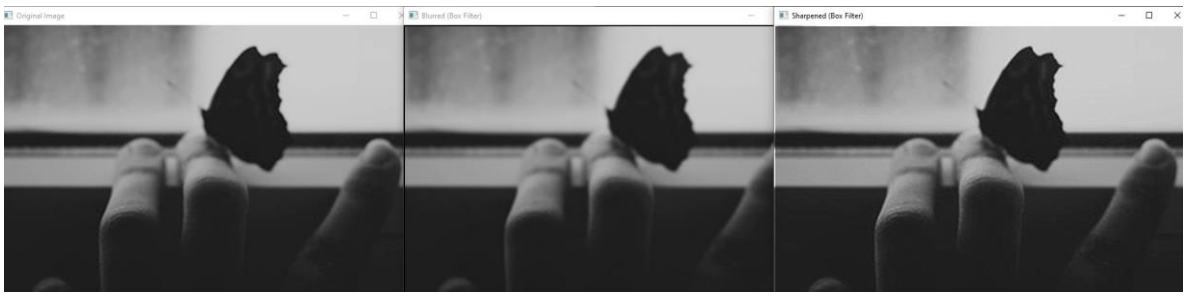
Реалізувати підвищення різкості зображень за допомогою нерізкого маскування (спробувати різні комбінації параметрів, в т.ч. різні операції згладжування, підібрати оптимальні для тестових зображень). При тестуванні брати розмиті зображення.

Результати виконання:

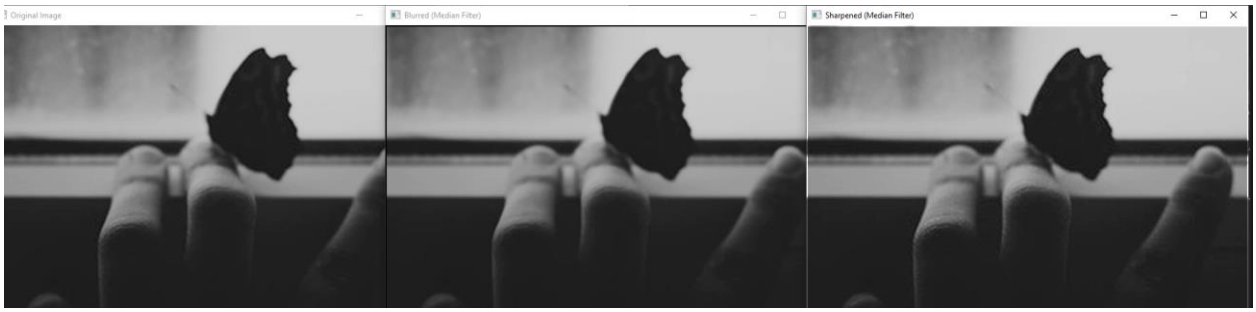
Коефіцієнт альфа = 1.5

Box filter:

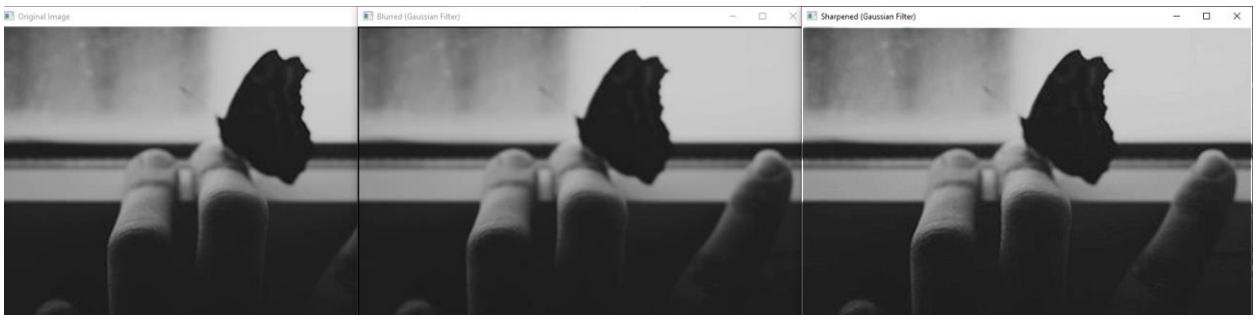




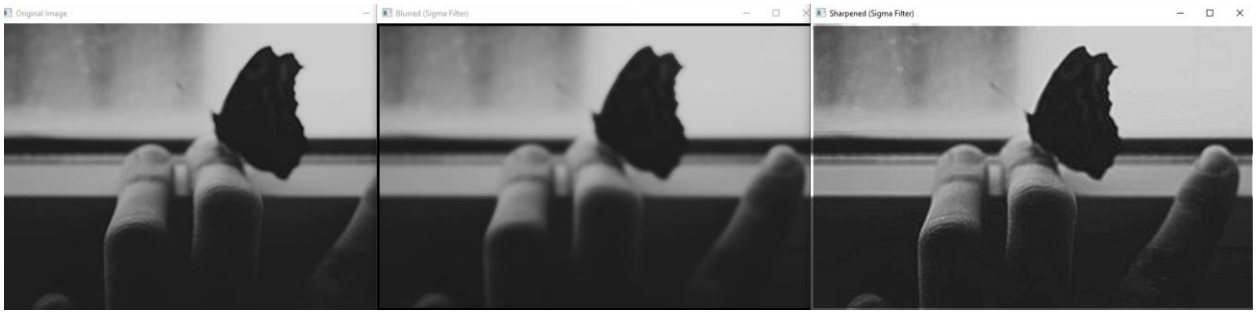
Median filter:



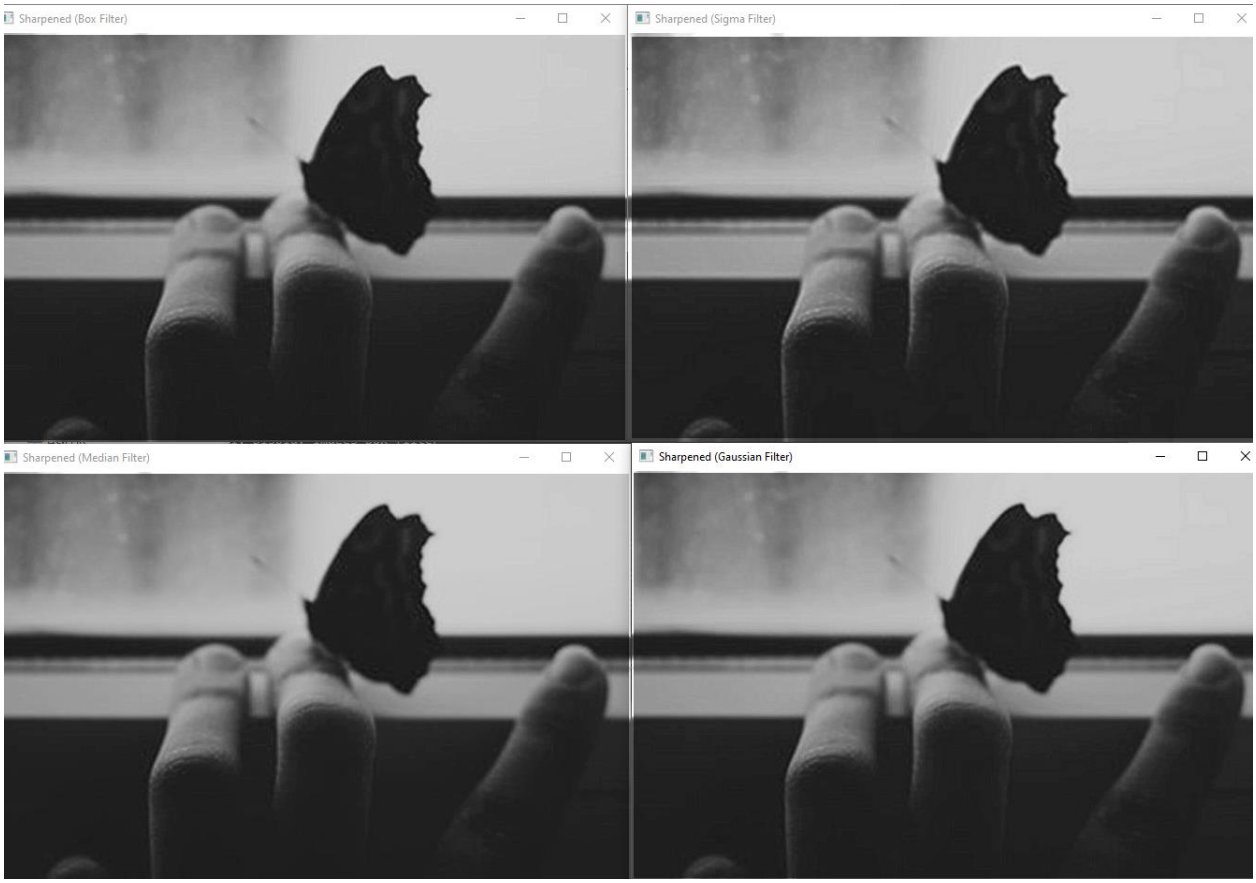
Gaussian filter:



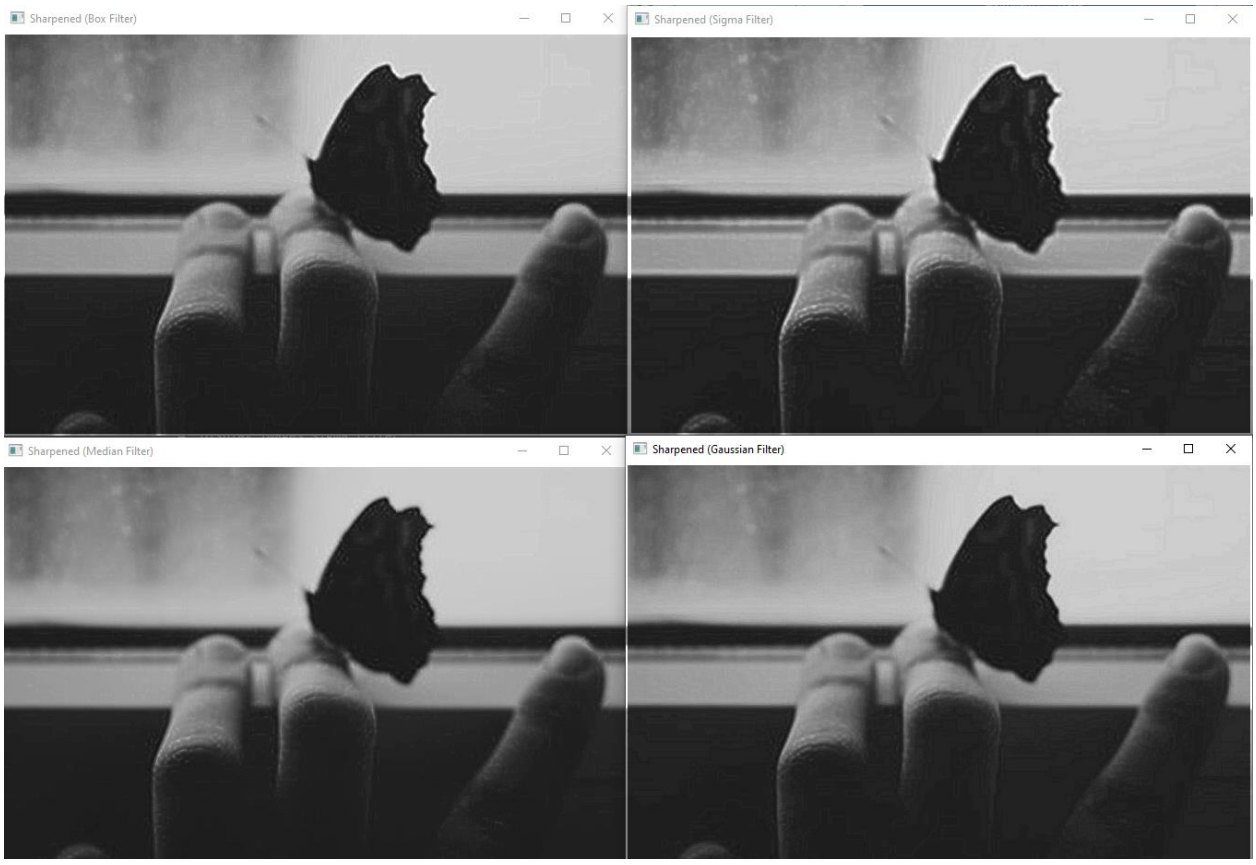
Sigma filter:



Разом:



Збільшимо коефіцієнт альфа для другого зображення на 3.5:



Основна мета нерізкого маскування це виділення контурів об'єктів і покращення деталей на зображенні шляхом застосування згладжувальних фільтрів для отримання фону, який потім віднімається від оригінального зображення. Далі, підвищені контури додаються до оригінального зображення з використанням коефіцієнта підсилення (alpha).

Вибір оптимального фільтра та налаштування його параметрів залежить від конкретних умов, таких як тип зображення, наявність шуму, і бажаний рівень різкості. Наприклад, для зображень з високим рівнем шуму краще використовувати медійний або сигмовий фільтр, тоді як для зображень з низьким шумом і необхідністю підвищення різкості Гауссовий або коробковий фільтри можуть бути більш ефективними.

Завдання 6

Реалізувати та порівняти детектори границь для півтонового зображенні на основі дискретних похідних, операторів Собеля, Шарра, Лапласса, та оператора Кенні (спробувати різні значення порогів та підібрати оптимальні для вхідних зображень). При тестуванні брати зображення, що містять границі.

Функція sobelEdgeDetection реалізує оператор Собеля, який використовує два конволюційні ядра для обчислення градієнтів по осі X та Y. Цей метод виявляє краю, обчислюючи величину градієнта для кожного пікселя на зображенні. Оператор Собеля добре підходить для виявлення вертикальних і горизонтальних границь, проте він може бути чутливим до шуму, що може вплинути на якість виявлення країв.

Функція scharrEdgeDetection реалізує оператор Шарра, який, подібно до оператора Собеля, використовує два ядра для обчислення градієнтів. Оператор Шарра є вдосконаленою версією Собеля і забезпечує кращу чутливість до границь, особливо в умовах шуму, оскільки його ядра мають кращу спрямованість у виявленні границь.

Функція laplaceEdgeDetection реалізує оператор Лапласса, який виявляє границі, обчислюючи другу похідну зображення. Цей метод менш чутливий до напрямку границь і виявляє границі у всіх напрямках, але може бути менш точним у присутності шуму. Якість виявлення границь може варіюватися в залежності від значень порогів, застосованих після обчислення градієнтів.

Функція cannyEdgeDetection реалізує оператор Кенні, який є більш складним методом виявлення границь. Він складається з кількох етапів, включаючи обчислення градієнтів, ненав'язливе подавлення максимумів та порогову обробку. Оператор Кенні, як правило, забезпечує високу точність виявлення країв, оскільки він спочатку видаляє менше значні границі, а потім виконує подвійне порогування для фільтрації шумих сигналів. Цей метод є дуже популярним завдяки своїй здатності виявляти деталі на зображеннях.

У тестуванні використовуються зображення з помітними границями, а також пробують різні значення порогів для досягнення оптимальних результатів виявлення границь. Кожен з реалізованих алгоритмів має свої сильні та слабкі сторони, і їхня ефективність може змінюватися залежно від конкретних умов зображення, включаючи наявність шуму та тип границь. Порівнюючи результати, можна виявити, що оператор Кенні найчастіше дає найкращі результати, але й інші методи можуть бути корисними в специфічних випадках.

Результати виконання:

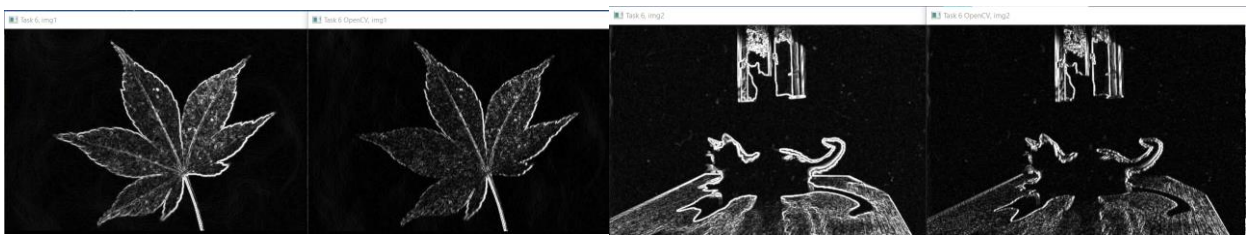


Рис.6.1 Порівняння детекторів границь
на основі дискретних похідних за допомоги оператора Собеля

Рис.6.2 Порівняння детекторів границь
на основі дискретних похідних за допомоги оператора Шарра

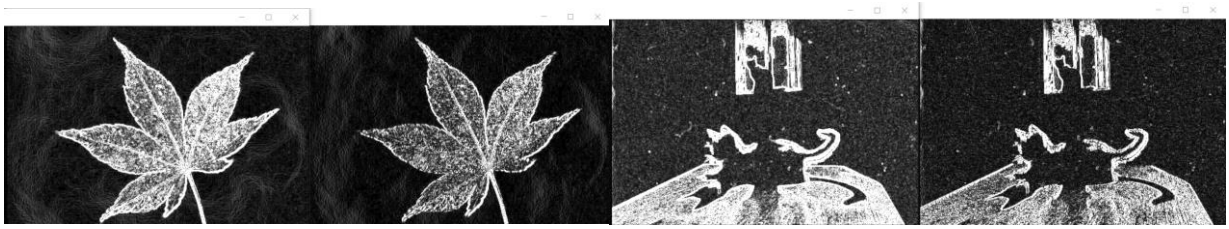


Рис.6.3 Порівняння детекторів границь
на основі дискретних похідних за допомоги оператора Лапласса

Рис.6.4 Порівняння детекторів границь
на основі дискретних похідних за допомоги оператора Кенні

Порівняння з функціями OpenCV:

Бібліотечні функції OpenCV, наприклад, cv::Sobel, cv::Scharr, cv::Laplacian та cv::Canny, надають оптимізовані рішення для виявлення границь, що часто включають апаратні оптимізації та додаткові налаштування, які підвищують їхню продуктивність. OpenCV реалізує ці алгоритми в компактному та ефективному вигляді, що дозволяє легко інтегрувати їх у проекти. Також бібліотечні функції часто мають параметри, які дозволяють користувачеві налаштувати поведінку алгоритму, що надає більшу гнучкість під час виявлення границь.

Реалізовані функції можуть бути менш ефективними в обробці великих зображень або при наявності шуму, оскільки вони не завжди включають оптимізації, які є в бібліотечних версіях. Наприклад, оператор Кенні в OpenCV реалізує складні етапи, такі як ненав'язливе подавлення максимумів і подвійне

порогування, що забезпечує високу точність виявлення границь. У власній реалізації цього алгоритму можуть бути відсутні деякі з цих етапів або вони можуть бути реалізовані менш оптимально.

В цілому, хоча реалізовані функції і демонструють основи роботи з виявлення границь, бібліотечні функції OpenCV пропонують більш ефективні, надійні та швидкі рішення, які підходять для широкого спектра застосувань у комп'ютерному зорі.

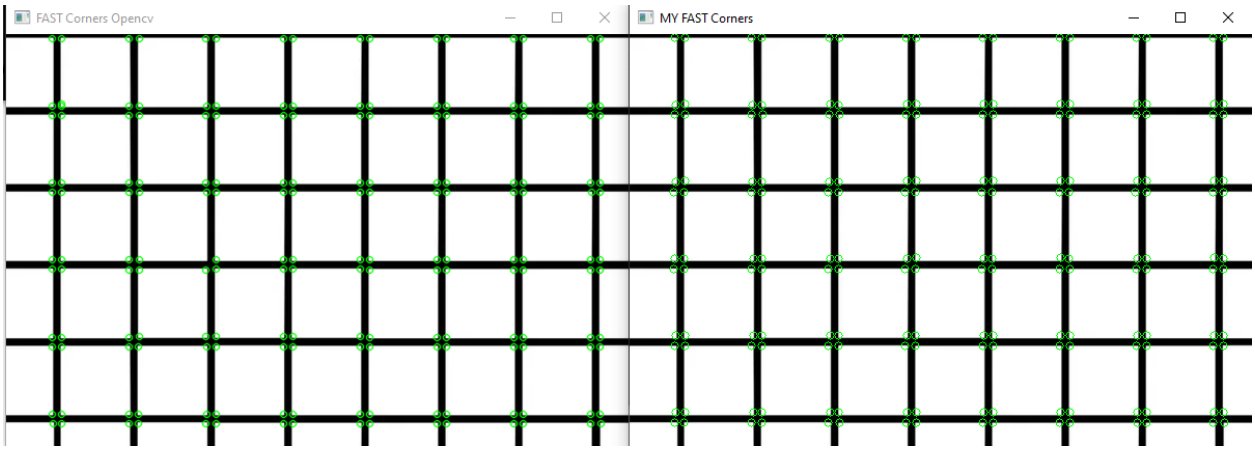
Завдання 7

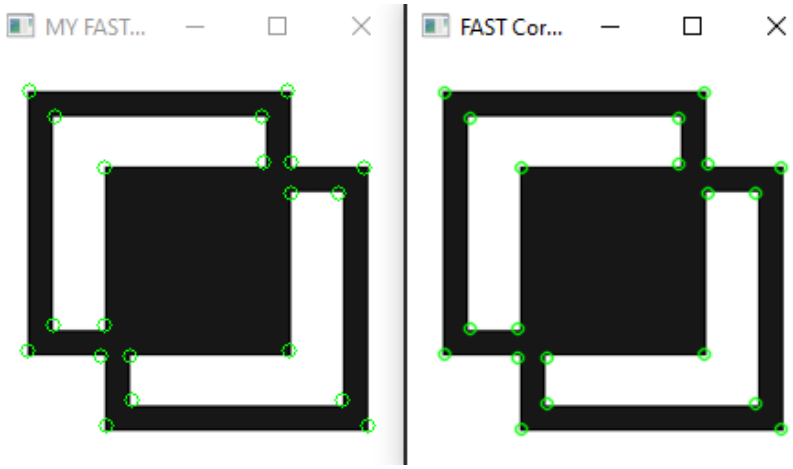
Реалізувати та порівняти детектори кутів Гарріса та FAST (підібрати оптимальне значення параметрів, в т.ч. кількість пікселів в критерії наявності кута). При тестуванні брати зображення, що містять кути. Візуалізувати знайдені кути на зображеннях.

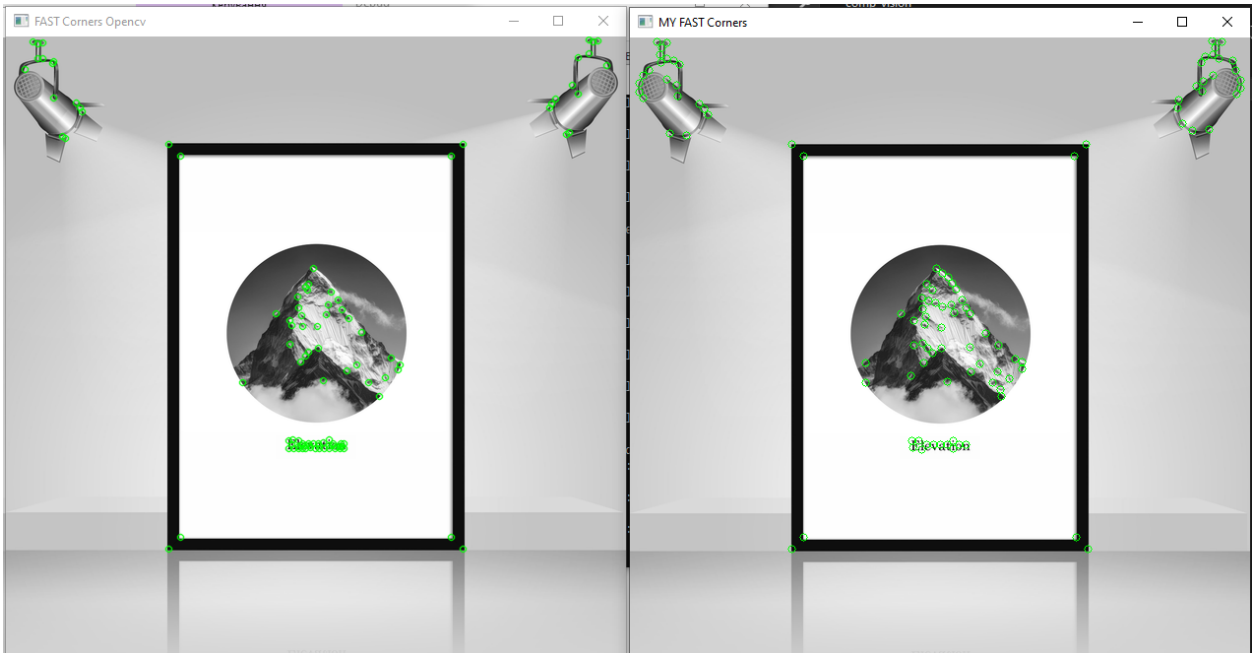
Результати виконання:

threshold = 80, кількість пікселів для наявності кута 9

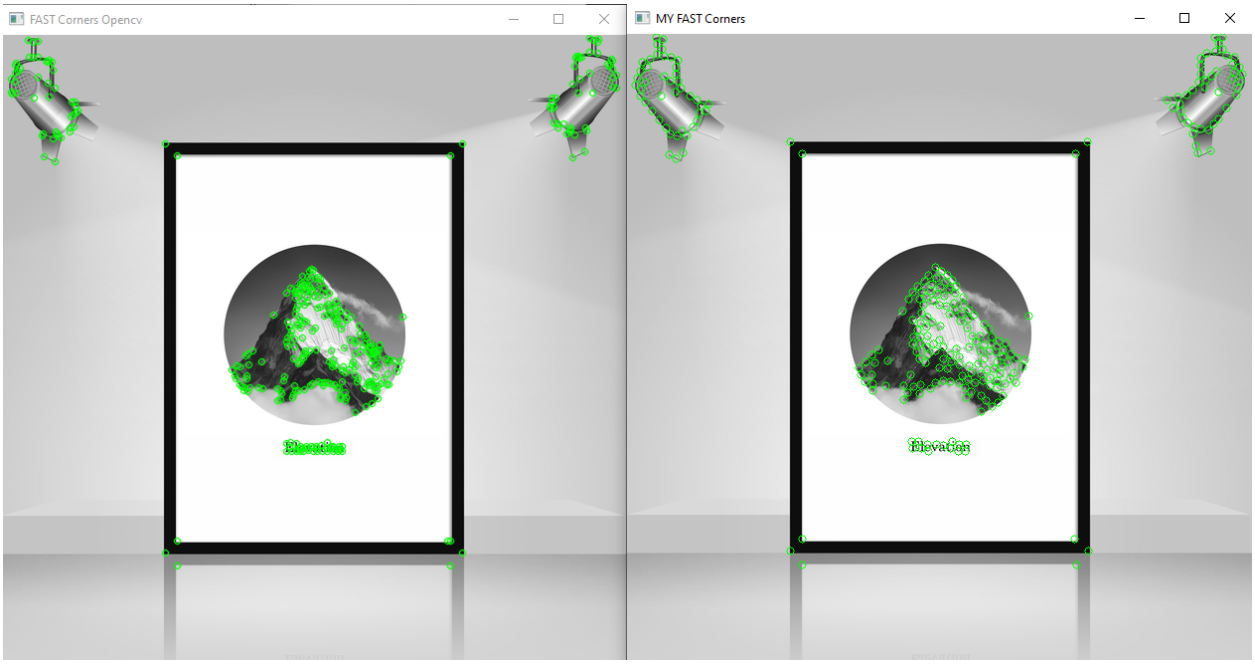
FAST:



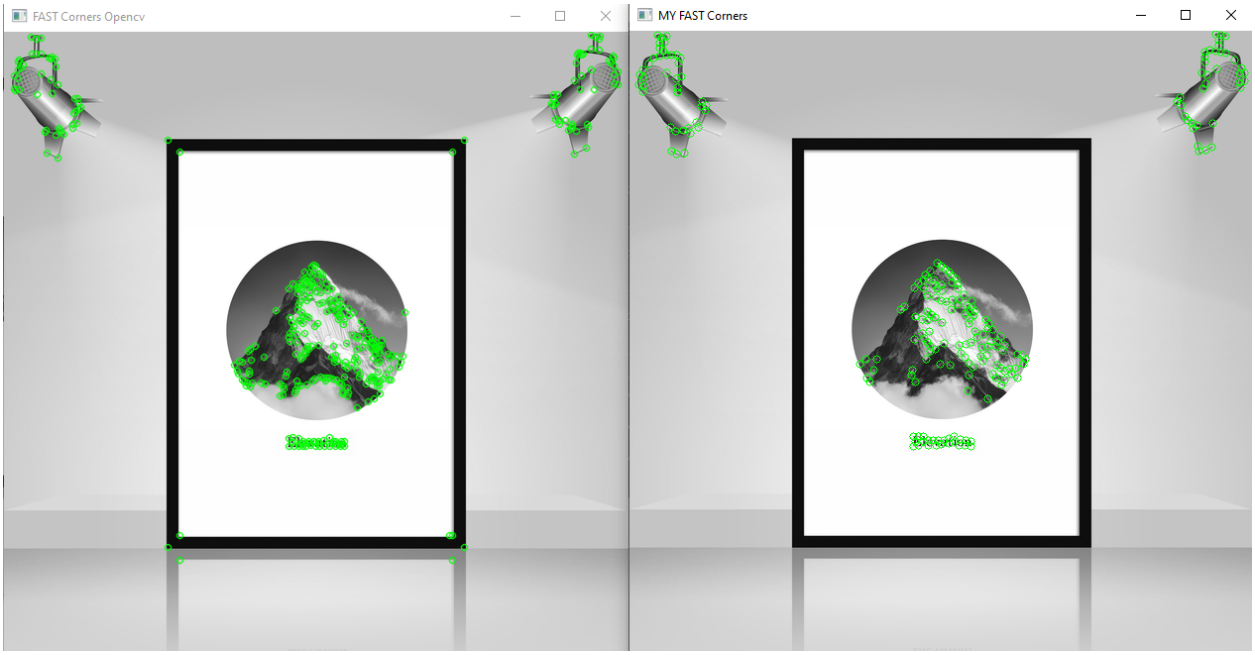




threshold = 40



кількість пікселів для наявності кута 12



У нашій реалізації ми використовували масив зміщень (offset) для аналізу сусідніх пікселів відносно центру. Кожен піксель оцінюється на основі яскравості в порівнянні з центровим пікселем. Для того щоб вважати піксель ключовою точкою, потрібно, щоб кількість яскравих або темних пікселів перевищувала 9 та 12. Як бачимо, збільшення до 12 призвело до якіснішого виявлення кутів. Це забезпечує відносно стійке виявлення кутів у різних умовах освітлення. Також було додано перевірку унікальності, щоб уникнути дублювання ключових точок виявлених у близьких позиціях. Обидва підходи показали свою ефективність при тестуванні на зображеннях, що містять кути. Однак вбудований детектор часто виявляв більше ключових точок у складних областях зображення. Це може бути пов'язано з оптимізаціями в алгоритмі OpenCV, які враховують специфічні аспекти обробки зображень.

У цілому, вбудований детектор FAST в OpenCV виявився більш надійним і ефективним у порівнянні з нашою реалізацією, адже він використовує параметри, які вбільшості підходять усім зображенням. Реалізований ж потрібно було оптимізовувати під певне зображення, хоча у деяких випадках це може покращити його результат порівняно з вбудованою функцією.

Завдання 8

Реалізувати та порівняти детектори границь LoG та DoG. При тестуванні

брати зображення, що містять границі.

Обидва ці методи є популярними в комп'ютерному зорі для виявлення границь у зображеннях, особливо в тих, що містять різкі переходи в яскравості, такі як контури об'єктів.

Для реалізації методу LoG використовується спочатку згладжування зображення за допомогою гауссового фільтра, який допомагає зменшити шум і підготувати зображення для подальшого аналізу. Функція gaussianBlur створює гауссовий розподіл на основі заданих параметрів, таких як розмір ядра і стандартне відхилення, і використовує його для згладжування зображення. Це зменшує деталізацію зображення, дозволяючи виявити значущі границі.

Після згладжування застосовується лапласіан, що обчислюється у функції laplacianFilter. Лапласіан є другим порядком похідної, що вказує на точки, де зображення змінюється швидше, виявляючи границі. Поєднання цих двох етапів дає змогу виявити границі за допомогою LoG.

Метод DoG, з іншого боку, реалізується через різницю між двома згладженими зображеннями, створеними за допомогою двох різних значень стандартного відхилення. У функції DoG спочатку зображення перетворюється в відтінки сірого, після чого застосовуються два різних гауссових фільтри з різними значеннями σ (sigma1 і sigma2). Різниця між цими двома зображеннями дає результат, який підкреслює границі.

Обидва методи підходять для виявлення границь, однак між ними існують важливі відмінності. LoG є більш чутливим до шуму через своє згладжування, а також може виявляти границі з більшою точністю за рахунок детальної обробки зображення. DoG є швидшим і менш обчислювально витратним, але може бути менш точним, особливо на зображеннях із значним шумом.

Результати виконання:

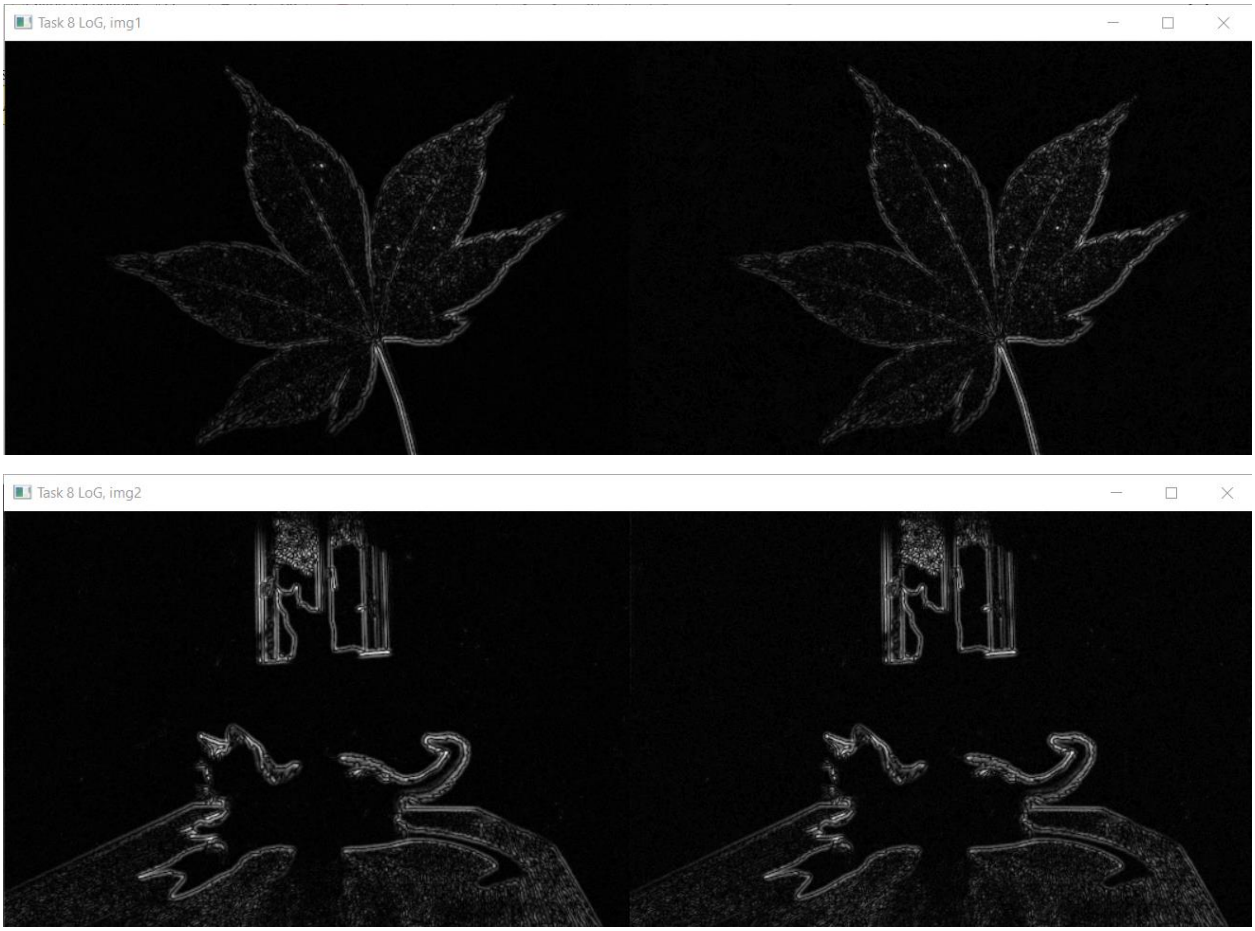


Рис.8.1 Порівняння детекторів границь LoG

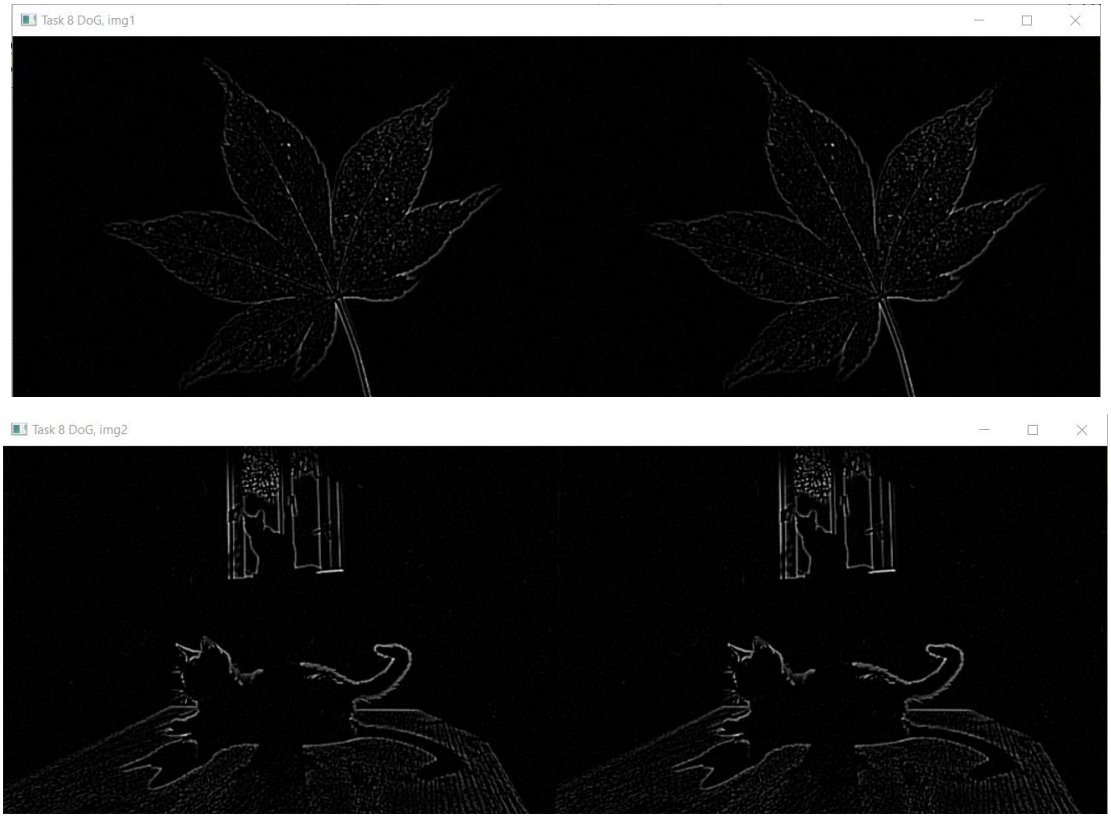


Рис.8.2 Порівняння детекторів границь DoG

Порівняння з функціями OpenCV:

Метод LoG спочатку вимагає згладжування зображення, щоб зменшити вплив шуму. Для цього в OpenCV часто використовують функцію GaussianBlur, яка застосовує гауссовий фільтр до зображення, згладжуючи його на основі параметрів, таких як розмір ядра і стандартне відхилення. Ця функція ефективно підготовляє зображення до подальшого аналізу. Після згладжування використовується оператор Лапласа, реалізований через Laplacian, що дозволяє виявити границі, розраховуючи другу похідну зображення. У OpenCV реалізація Laplacian може бути оптимізована для швидкого обчислення, що робить її зручною для реальних застосувань.

З іншого боку, метод DoG також починається зі згладжування зображення. Для його реалізації в OpenCV знову використовується GaussianBlur, але тут застосовуються два фільтри з різними значеннями стандартного відхилення. Після цього результати обох згладжених зображень віднімуться одне від одного, щоб виявити границі. Для цього у OpenCV можна використовувати функцію subtract, яка дозволяє легко обчислити різницю між двома матрицями.

Обидва методи мають свої переваги та недоліки. LoG, з використанням обох згладжувальних і детектуючих операцій, забезпечує високу точність виявлення границь, але може бути більш чутливим до шуму, особливо якщо значення стандартного відхилення обрані неправильно. Використання OpenCV дозволяє швидко реалізувати цей процес з відповідними функціями, такими як GaussianBlur та Laplacian.

Метод DoG, навпаки, є більш швидким і менш обчислювально витратним. Він дозволяє виявляти границі з більшою швидкістю, що є перевагою в умовах обмежених ресурсів або необхідності обробки великих обсягів даних. Однак, в деяких випадках, DoG може давати менш чіткі результати в порівнянні з LoG через використання двох фільтрів.

Висновок:

У роботі реалізовано ряд завдань, пов'язаних із обробкою зображень за допомогою програмування, з акцентом на використання бібліотеки OpenCV. Першим кроком було реалізовано вирівнювання гістограми півтонового зображення та умовне масштабування, що дозволяє привести значення середнього та дисперсії одного зображення до значень іншого. Далі було розроблено процедуру застосування лінійного локального оператора до півтонового зображення, а також порівняння різних методів згладжування зображень, включаючи прямокутний, медіанний, Гауссовий та сигма-фільтри, для видалення шуму.

Наступним етапом було підвищення різкості зображень за допомогою нерізкого маскування з оптимізацією параметрів. У роботі також були реалізовані та порівняні різні детектори границь на основі дискретних похідних, таких як оператори Собеля, Шарра, Лапласса та Кенні, з підбором оптимальних порогів. Досліджувалися детектори кутів Гарріса та FAST, а також границі LoG і DoG, з візуалізацією знайдених куточків та границь.

Додатково, для верифікації результатів роботи реалізованих функцій передбачено порівняння з аналогічними функціями бібліотеки OpenCV, що дозволяє оцінити ефективність самостійно розроблених алгоритмів. Усі задачі виконані відповідно до вимог проекту з акцентом на самостійну реалізацію без використання готових алгоритмів бібліотек.

Додаток А:

Func.hpp

```
#ifndef FUNC_HPP
#define FUNC_HPP

#include <opencv2/opencv.hpp>

void scaledMeanStd(const cv::Mat& img1_gray, const cv::Mat& img2_gray,
cv::Mat& result);
void addNoise(cv::Mat& image, double noise_stddev);
void BoxFilter(const cv::Mat& src, cv::Mat& dst, cv::Size kernelSize);
void MedianFilter(const cv::Mat& src, cv::Mat& dst, int kernelSize);
void GaussianFilter(const cv::Mat& src, cv::Mat& dst, cv::Size kernelSize,
double sigma);
void SigmaFilter(const cv::Mat& src, cv::Mat& dst, int diameter, double
sigmaColor, double sigmaSpace);
void sobelEdgeDetection(const cv::Mat& src, cv::Mat& dst);
void scharrEdgeDetection(const cv::Mat& src, cv::Mat& dst);
void laplaceEdgeDetection(const cv::Mat& src, cv::Mat& dst);
void cannyEdgeDetection(const cv::Mat& src, cv::Mat& dst, double
lowerThreshold, double upperThreshold);

cv::Mat gaussianBlur(const cv::Mat& image, cv::Size ksize, double sigma);
cv::Mat laplacianFilter(const cv::Mat& image);
cv::Mat DoG(const cv::Mat& input, cv::Size ksize, double sigma1, double
sigma2);

#endif
```

Func.cpp

```
#include "func.hpp"
#include <opencv2/opencv.hpp>
#include <cmath>
#define _USE_MATH_DEFINES
#include <math.h>
#include <vector>

using namespace cv;
using namespace std;

// Task 1
void scaledMeanStd(const Mat& img1_gray, const Mat& img2_gray, Mat& result) {
    Scalar meanValue1, stdDevValue1;
    meanStdDev(img1_gray, meanValue1, stdDevValue1);
    double mean_img1 = meanValue1[0];
    double std_img1 = stdDevValue1[0];

    Scalar meanValue2, stdDevValue2;
    meanStdDev(img2_gray, meanValue2, stdDevValue2);
    double mean_img2 = meanValue2[0];
    double std_img2 = stdDevValue2[0];

    Mat img1_scaled;
    img1_gray.convertTo(img1_scaled, CV_64F);
    result = (img1_scaled - mean_img1) / std_img1 * std_img2 + mean_img2;

    result.convertTo(result, img1_gray.type());
}

//Task 2

void addNoise(Mat& image, double noise_stddev)
```

```
{
    Mat noise(image.size(), image.type());
    randn(noise, 0, noise_stddev);
    image += noise;
}

void BoxFilter(const Mat& src, Mat& dst, Size kernelSize)
{
    dst = Mat::zeros(src.size(), src.type());
    int kx = kernelSize.width / 2;
    int ky = kernelSize.height / 2;
    for (int y = ky; y < src.rows - ky; y++)
    {
        for (int x = kx; x < src.cols - kx; x++)
        {
            float sum = 0.0;
            for (int j = -ky; j <= ky; j++)
            {
                for (int i = -kx; i <= kx; i++)
                {
                    sum += src.at<uchar>(y + j, x + i);
                }
            }
            dst.at<uchar>(y, x) = sum / (kernelSize.width *
kernelSize.height);
        }
    }
}

void MedianFilter(const Mat& src, Mat& dst, int kernelSize)
{
    dst = Mat::zeros(src.size(), src.type());
    int k = kernelSize / 2;
    for (int y = k; y < src.rows - k; y++)
    {
        for (int x = k; x < src.cols - k; x++)
        {
            std::vector<uchar> neighborhood;
            for (int j = -k; j <= k; j++)
            {
                for (int i = -k; i <= k; i++)
                {
                    neighborhood.push_back(src.at<uchar>(y + j, x + i));
                }
            }
            sort(neighborhood.begin(), neighborhood.end());
            dst.at<uchar>(y, x) = neighborhood[neighborhood.size() / 2];
        }
    }
}

void GaussianFilter(const Mat& src, Mat& dst, Size kernelSize, double sigma)
{
    dst = Mat::zeros(src.size(), src.type());
    int kx = kernelSize.width / 2;
    int ky = kernelSize.height / 2;
    std::vector<std::vector<double>> kernel(kernelSize.height,
vector<double>(kernelSize.width));
    double sum = 0.0;

    for (int y = -ky; y <= ky; y++)
    {
        for (int x = -kx; x <= kx; x++)
        {
```

```
            kernel[y + ky][x + kx] = exp(-(x * x + y * y) / (2 * sigma *
sigma));
            sum += kernel[y + ky][x + kx];
        }
    }

    for (int y = 0; y < kernelSize.height; y++)
    {
        for (int x = 0; x < kernelSize.width; x++)
        {
            kernel[y][x] /= sum;
        }
    }

    for (int y = ky; y < src.rows - ky; y++)
    {
        for (int x = kx; x < src.cols - kx; x++)
        {
            float sum = 0.0;
            for (int j = -ky; j <= ky; j++)
            {
                for (int i = -kx; i <= kx; i++)
                {
                    sum += src.at<uchar>(y + j, x + i) * kernel[j + ky][i +
kx];
                }
            }
            dst.at<uchar>(y, x) = sum;
        }
    }
}

void SigmaFilter(const Mat& src, Mat& dst, int diameter, double sigmaColor,
double sigmaSpace)
{
    dst = Mat::zeros(src.size(), src.type());
    int radius = diameter / 2;
    for (int y = radius; y < src.rows - radius; y++)
    {
        for (int x = radius; x < src.cols - radius; x++)
        {
            float sum = 0.0;
            float weightSum = 0.0;
            uchar center = src.at<uchar>(y, x);
            for (int j = -radius; j <= radius; j++)
            {
                for (int i = -radius; i <= radius; i++)
                {
                    uchar neighbor = src.at<uchar>(y + j, x + i);
                    double spaceWeight = exp(-(i * i + j * j) / (2 *
sigmaSpace * sigmaSpace));
                    double colorWeight = exp(-((neighbor - center) *
(neighbor - center)) /
                        (2 * sigmaColor * sigmaColor));
                    double weight = spaceWeight * colorWeight;
                    sum += neighbor * weight;
                    weightSum += weight;
                }
            }
            dst.at<uchar>(y, x) = sum / weightSum;
        }
    }
}
//Task 3
```

```
void sobelEdgeDetection(const Mat& src, Mat& dst) {
    dst = Mat::zeros(src.size(), CV_64F);
    int sobelKernelX[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };
    int sobelKernelY[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };

    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            double gx = 0.0, gy = 0.0;
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    gx += sobelKernelX[i + 1][j + 1] * src.at<uchar>(y + i, x
+ j);
                    gy += sobelKernelY[i + 1][j + 1] * src.at<uchar>(y + i, x
+ j);
                }
            }
            dst.at<double>(y, x) = sqrt(gx * gx + gy * gy);
        }
    }
    dst.convertTo(dst, CV_8U);
}

void scharrEdgeDetection(const Mat& src, Mat& dst) {
    dst = Mat::zeros(src.size(), CV_64F);
    int scharrKernelX[3][3] = { {-3, 0, 3}, {-10, 0, 10}, {-3, 0, 3} };
    int scharrKernelY[3][3] = { {-3, -10, -3}, {0, 0, 0}, {3, 10, 3} };

    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            double gx = 0.0, gy = 0.0;
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    gx += scharrKernelX[i + 1][j + 1] * src.at<uchar>(y + i,
x + j);
                    gy += scharrKernelY[i + 1][j + 1] * src.at<uchar>(y + i,
x + j);
                }
            }
            dst.at<double>(y, x) = sqrt(gx * gx + gy * gy);
        }
    }
    dst.convertTo(dst, CV_8U);
}

void laplaceEdgeDetection(const Mat& src, Mat& dst) {
    dst = Mat::zeros(src.size(), CV_64F);
    int laplaceKernel[3][3] = { {0, 1, 0}, {1, -4, 1}, {0, 1, 0} };

    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            double sum = 0.0;
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    sum += laplaceKernel[i + 1][j + 1] * src.at<uchar>(y + i,
x + j);
                }
            }
            dst.at<double>(y, x) = sum;
        }
    }
    dst.convertTo(dst, CV_8U);
}
```

```
void cannyEdgeDetection(const Mat& src, Mat& dst, double lowerThreshold = 50,
double upperThreshold = 150) {
    dst = Mat::zeros(src.size(), CV_8U);

    Mat gradientMagnitude = Mat::zeros(src.size(), CV_64F);
    Mat gradientDirection = Mat::zeros(src.size(), CV_64F);
    int sobelKernelX[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };
    int sobelKernelY[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };

    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            double gx = 0.0, gy = 0.0;
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    gx += sobelKernelX[i + 1][j + 1] * src.at<uchar>(y + i, x
+ j);
                    gy += sobelKernelY[i + 1][j + 1] * src.at<uchar>(y + i, x
+ j);
                }
            }
            gradientMagnitude.at<double>(y, x) = sqrt(gx * gx + gy * gy);
            gradientDirection.at<double>(y, x) = atan2(gy, gx);
        }
    }

    Mat nonMaxSuppressed = Mat::zeros(src.size(), CV_64F);
    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            double direction = gradientDirection.at<double>(y, x) * 180.0 /
CV_PI;
            direction = fmod(direction + 180.0, 180.0);

            double magnitude = gradientMagnitude.at<double>(y, x);
            double q = 0.0, r = 0.0;
            if ((0 <= direction && direction < 22.5) || (157.5 <= direction
&& direction <= 180)) {
                q = gradientMagnitude.at<double>(y, x + 1);
                r = gradientMagnitude.at<double>(y, x - 1);
            }
            else if (22.5 <= direction && direction < 67.5) {
                q = gradientMagnitude.at<double>(y + 1, x - 1);
                r = gradientMagnitude.at<double>(y - 1, x + 1);
            }
            else if (67.5 <= direction && direction < 112.5) {
                q = gradientMagnitude.at<double>(y + 1, x);
                r = gradientMagnitude.at<double>(y - 1, x);
            }
            else if (112.5 <= direction && direction < 157.5) {
                q = gradientMagnitude.at<double>(y - 1, x - 1);
                r = gradientMagnitude.at<double>(y + 1, x + 1);
            }

            if (magnitude >= q && magnitude >= r) {
                nonMaxSuppressed.at<double>(y, x) = magnitude;
            }
        }
    }

    for (int y = 1; y < src.rows - 1; y++) {
        for (int x = 1; x < src.cols - 1; x++) {
            if (nonMaxSuppressed.at<double>(y, x) >= upperThreshold) {
                dst.at<uchar>(y, x) = 255;
            }
            else if (nonMaxSuppressed.at<double>(y, x) < lowerThreshold) {
```

```
                dst.at<uchar>(y, x) = 0;
            }
            else {
                bool connected = false;
                for (int i = -1; i <= 1; i++) {
                    for (int j = -1; j <= 1; j++) {
                        if (dst.at<uchar>(y + i, x + j) == 255) {
                            connected = true;
                            break;
                        }
                    }
                    if (connected) break;
                }
                dst.at<uchar>(y, x) = connected ? 255 : 0;
            }
        }
    }
}


//Task 4

Mat gaussianBlur(const Mat& image, Size ksize, double sigma) {
    int kx = ksize.width;
    int ky = ksize.height;
    Mat kernel = Mat::zeros(kx, ky, CV_64F);
    int center_x = kx / 2;
    int center_y = ky / 2;
    double sigma2 = 2 * sigma * sigma;
    double sum_val = 0.0;

    for (int i = 0; i < kx; ++i) {
        for (int j = 0; j < ky; ++j) {
            int x = i - center_x;
            int y = j - center_y;
            kernel.at<double>(i, j) = exp(-(x * x + y * y) / sigma2) / (2 *
M_PI * sigma2);
            sum_val += kernel.at<double>(i, j);
        }
    }

    kernel /= sum_val;


    Mat padded_image;
    copyMakeBorder(image, padded_image, center_x, center_x, center_y,
center_y, BORDER_REFLECT);
    Mat output = Mat::zeros(image.size(), CV_64F);

    for (int i = 0; i < image.rows; ++i) {
        for (int j = 0; j < image.cols; ++j) {
            double sum = 0.0;
            for (int m = 0; m < kx; ++m) {
                for (int n = 0; n < ky; ++n) {
                    int x = i + m;
                    int y = j + n;
                    sum += padded_image.at<uchar>(x, y) *
kernel.at<double>(m, n);
                }
            }
            output.at<double>(i, j) = sum;
        }
    }
    output.convertTo(output, CV_8U);
```

```
    return output;
}

Mat laplacianFilter(const Mat& image) {
    Mat laplacian_kernel = (Mat_<double>(3, 3) << 0, 1, 0,
        1, -4, 1,
        0, 1, 0);
    int kx = laplacian_kernel.rows;
    int ky = laplacian_kernel.cols;
    int pad_x = kx / 2;
    int pad_y = ky / 2;

    Mat padded_image;
    copyMakeBorder(image, padded_image, pad_x, pad_x, pad_y, pad_y,
BORDER_REFLECT);
    Mat output = Mat::zeros(image.size(), CV_64F);

    for (int i = 0; i < image.rows; ++i) {
        for (int j = 0; j < image.cols; ++j) {
            double sum = 0.0;
            for (int m = 0; m < kx; ++m) {
                for (int n = 0; n < ky; ++n) {
                    int x = i + m;
                    int y = j + n;
                    sum += padded_image.at<uchar>(x, y) *
laplacian_kernel.at<double>(m, n);
                }
            }
            output.at<double>(i, j) = sum;
        }
    }

    return output;
}

Mat DoG(const Mat& input, Size ksize, double sigma1, double sigma2) {
    Mat gray;
    if (input.channels() == 3) {
        cvtColor(input, gray, COLOR_BGR2GRAY);
    }
    else {
        gray = input;
    }

    Mat blur1 = gaussianBlur(gray, ksize, sigma1);
    Mat blur2 = gaussianBlur(gray, ksize, sigma2);
    Mat dog, dogImage;
    subtract(blur1, blur2, dog);


    normalize(dog, dogImage, 0, 255, NORM_MINMAX);
    dogImage.convertTo(dogImage, CV_8UC1);

    return dogImage;
}
```

App.cpp

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include "func.hpp"

using namespace cv;
using namespace std;
```

```
void Task2(const Mat& img1, const Mat& img2) {
    Mat scaledImage1, scaledImage2, combinedImage;
    scaledMeanStd(img1, img2, scaledImage1);
    scaledMeanStd(img2, img1, scaledImage2);

    hconcat(scaledImage1, scaledImage2, combinedImage);

    imshow("Task 2", combinedImage);
    waitKey();
}

void Task4(const Mat& img1, const Mat& img2) {
    Mat img11Noise = (img1).clone(), img12Noise = (img2).clone();
    Mat img21Noise = (img1).clone(), img22Noise = (img2).clone();
    addNoise(img11Noise, 50);
    addNoise(img12Noise, 50);
    addNoise(img21Noise, 50);
    addNoise(img22Noise, 50);

    Mat BFImg1, MGImg1, GFImg1, SFImg1, BFImg2, MGImg2, GFImg2, SFImg2;
    Mat BFImg12, MGImg12, GFImg12, SFImg12, BFImg22, MGImg22, GFImg22,
SFImg22;
    BoxFilter(img11Noise, BFImg1, Size(5, 5));
    MedianFilter(img11Noise, MGImg1, 5);
    GaussianFilter(img11Noise, GFImg1, Size(5, 5), 1.5);
    SigmaFilter(img11Noise, SFImg1, 9, 75, 75);
    BoxFilter(img12Noise, BFImg2, Size(5, 5));
    MedianFilter(img12Noise, MGImg2, 5);
    GaussianFilter(img12Noise, GFImg2, Size(5, 5), 1.5);
    SigmaFilter(img12Noise, SFImg2, 9, 75, 75);

    blur(img21Noise, BFImg12, Size(5, 5));
    medianBlur(img21Noise, MGImg12, 5);
    GaussianBlur(img21Noise, GFImg12, Size(5, 5), 1.5);
    bilateralFilter(img21Noise, SFImg12, 9, 75, 75);
    blur(img22Noise, BFImg22, Size(5, 5));
    medianBlur(img22Noise, MGImg22, 5);
    GaussianBlur(img22Noise, GFImg22, Size(5, 5), 1.5);
    bilateralFilter(img22Noise, SFImg22, 9, 75, 75);

    Mat topRow41, bottomRow41, combined41;
    hconcat(BFImg1, MGImg1, topRow41);
    hconcat(GFImg1, SFImg1, bottomRow41);
    vconcat(topRow41, bottomRow41, combined41);

    Mat topRow42, bottomRow42, combined42;
    hconcat(BFImg2, MGImg2, topRow42);
    hconcat(GFImg2, SFImg2, bottomRow42);
    vconcat(topRow42, bottomRow42, combined42);

    Mat topRow411, bottomRow411, combined411;
    hconcat(BFImg12, MGImg12, topRow411);
    hconcat(GFImg12, SFImg12, bottomRow411);
    vconcat(topRow411, bottomRow411, combined411);

    Mat topRow422, bottomRow422, combined422;
    hconcat(BFImg22, MGImg22, topRow422);
    hconcat(GFImg22, SFImg22, bottomRow422);
    vconcat(topRow422, bottomRow422, combined422);

    imshow("Task 4, img1", combined41);
    imshow("Task 4 OpenCV, img1", combined411);
    waitKey();
```

```
    imshow("Task 4, img2", combined42);
    imshow("Task 4 OpenCV, img2", combined422);
    waitKey();
}

void Task6(const Mat& img1, const Mat& img2) {
    Mat sobelEdges1, scharrEdges1, laplaceEdges1, cannyEdges1, sobelEdges2,
scharrEdges2, laplaceEdges2, cannyEdges2;
    Mat sobelEdges12, scharrEdges12, laplaceEdges12, cannyEdges12,
sobelEdges22, scharrEdges22, laplaceEdges22, cannyEdges22;
    sobelEdgeDetection(img1, sobelEdges1);
    scharrEdgeDetection(img1, scharrEdges1);
    laplaceEdgeDetection(img1, laplaceEdges1);
    cannyEdgeDetection(img1, cannyEdges1, 70, 200);
    sobelEdgeDetection(img2, sobelEdges2);
    scharrEdgeDetection(img2, scharrEdges2);
    laplaceEdgeDetection(img2, laplaceEdges2);
    cannyEdgeDetection(img2, cannyEdges2, 70, 200);

    Sobel(img1, sobelEdges12, CV_64F, 1, 0, 3);
    Scharr(img1, scharrEdges12, CV_64F, 1, 0);
    Laplacian(img1, laplaceEdges12, CV_64F);
    Canny(img1, cannyEdges12, 70, 200);
    Sobel(img2, sobelEdges22, CV_64F, 1, 0, 3);
    Scharr(img2, scharrEdges22, CV_64F, 1, 0);
    Laplacian(img2, laplaceEdges22, CV_64F);
    Canny(img2, cannyEdges22, 70, 200);

    convertScaleAbs(sobelEdges12, sobelEdges12);
    convertScaleAbs(scharrEdges12, scharrEdges12);
    convertScaleAbs(laplaceEdges12, laplaceEdges12);

    convertScaleAbs(sobelEdges22, sobelEdges22);
    convertScaleAbs(scharrEdges22, scharrEdges22);
    convertScaleAbs(laplaceEdges22, laplaceEdges22);

    Mat topRow61, bottomRow61, combined61, topRow62, bottomRow62, combined62;
    Mat topRow612, bottomRow612, combined612, topRow622, bottomRow622,
combined622;
    hconcat(sobelEdges1, scharrEdges1, topRow61);
    hconcat(laplaceEdges1, cannyEdges1, bottomRow61);
    vconcat(topRow61, bottomRow61, combined61);
    hconcat(sobelEdges2, scharrEdges2, topRow62);
    hconcat(laplaceEdges2, cannyEdges2, bottomRow62);
    vconcat(topRow62, bottomRow62, combined62);

    hconcat(sobelEdges12, scharrEdges12, topRow612);
    hconcat(laplaceEdges12, cannyEdges12, bottomRow612);
    vconcat(topRow612, bottomRow612, combined612);
    hconcat(sobelEdges22, scharrEdges22, topRow622);
    hconcat(laplaceEdges22, cannyEdges22, bottomRow622);
    vconcat(topRow622, bottomRow622, combined622);

    imshow("Task 6, img1", combined61);
    imshow("Task 6 OpenCV, img1", combined612);
    waitKey();

    imshow("Task 6, img2", combined62);
    imshow("Task 6 OpenCV, img2", combined622);
    waitKey();
}

void Task8(const Mat& img1, const Mat& img2) {
    Size ksize(5, 5);
```

```
    double sigma1 = 1.0;
    double sigma2 = 2.0;

    Mat gaussian_blurred1 = gaussianBlur(img1, ksize, sigma1);
    Mat gaussian_blurred2 = gaussianBlur(img2, ksize, sigma1);
    gaussian_blurred1.convertTo(gaussian_blurred1, CV_8U);
    gaussian_blurred2.convertTo(gaussian_blurred2, CV_8U);

    Mat log_scratch1 = laplacianFilter(gaussian_blurred1);
    Mat log_scratch2 = laplacianFilter(gaussian_blurred2);
    log_scratch1 = abs(log_scratch1);
    log_scratch2 = abs(log_scratch2);
    Mat log_scratch_norm1, log_scratch_norm2;
    normalize(log_scratch1, log_scratch_norm1, 0, 255, NORM_MINMAX);
    normalize(log_scratch2, log_scratch_norm2, 0, 255, NORM_MINMAX);
    log_scratch_norm1.convertTo(log_scratch_norm1, CV_8U);
    log_scratch_norm2.convertTo(log_scratch_norm2, CV_8U);

    Mat blurred1, blurred2;
    GaussianBlur(img1, blurred1, ksize, sigma1);
    GaussianBlur(img2, blurred2, ksize, sigma1);

    Mat laplacian1, laplacian2;
    Laplacian(blurred1, laplacian1, CV_16S, 3);
    Laplacian(blurred2, laplacian2, CV_16S, 3);

    Mat absLaplacian1, combinedImage81LoG, absLaplacian2, combinedImage82LoG;
    convertScaleAbs(laplacian1, absLaplacian1);
    convertScaleAbs(laplacian2, absLaplacian2);

    hconcat(absLaplacian1, log_scratch_norm1, combinedImage81LoG);
    hconcat(absLaplacian2, log_scratch_norm2, combinedImage82LoG);

    imshow("Task 8 LoG, img1", combinedImage81LoG);
    waitKey();

    imshow("Task 8 LoG, img2", combinedImage82LoG);
    waitKey();


    Mat customDoGResult1 = DoG(img1, ksize, sigma1, sigma2);
    Mat customDoGResult2 = DoG(img2, ksize, sigma1, sigma2);

    Mat blur11, blur12, openCvDoGResult1, dogImage1, blur21, blur22,
openCvDoGResult2, dogImage2, combinedImage81DoG, combinedImage82DoG;
    GaussianBlur(img1, blur11, ksize, sigma1);
    GaussianBlur(img1, blur12, ksize, sigma2);
    subtract(blur11, blur12, openCvDoGResult1);
    GaussianBlur(img2, blur21, ksize, sigma1);
    GaussianBlur(img2, blur22, ksize, sigma2);
    subtract(blur21, blur22, openCvDoGResult2);

    normalize(openCvDoGResult1, dogImage1, 0, 255, NORM_MINMAX);
    dogImage1.convertTo(dogImage1, CV_8UC1);

    normalize(openCvDoGResult2, dogImage2, 0, 255, NORM_MINMAX);
    dogImage2.convertTo(dogImage2, CV_8UC1);

    hconcat(customDoGResult1, dogImage1, combinedImage81DoG);
    hconcat(customDoGResult2, customDoGResult2, combinedImage82DoG);

    imshow("Task 8 DoG, img1", combinedImage81DoG);
    waitKey();
```

```
    imshow("Task 8 DoG, img2", combinedImage82DoG);
    waitKey();
}

int main() {
    Mat img1 = imread("img.jpg", IMREAD_GRAYSCALE);
    Mat img2 = imread("img2.jpg", IMREAD_GRAYSCALE);

    if (img1.empty() || img2.empty()) {
        cout << "Could not open or find the images!" << endl;
        return -1;
    }
    resize(img1, img1, Size(img1.cols * 0.8, img1.rows * 0.8));
    if (img1.size() != img2.size()) {
        resize(img2, img2, img1.size());
    }

    int choice;
    while (true) {
        cout << "Choose a task to display (2, 4, 6, 8) or 0 to exit: ";
        cin >> choice;

        switch (choice) {
        case 2:
            Task2(img1, img2);
            break;
        case 4:
            Task4(img1, img2);
            break;
        case 6:
            Task6(img1, img2);
            break;
        case 8:
            Task8(img1, img2);
            break;
        case 0:
            return 0;
        default:
            cout << "Invalid choice, try again!" << endl;
        }
    }
    return 0;
}
```

Додаток Б:

Func.hpp

```
#pragma once
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

namespace comp_vis {

    Mat histogramEqualization(const Mat& src);
    vector<int> computeHistogram(const Mat& src);
    void displayHistogram(const std::vector<int>& hist, const std::string&
filename);

    Mat myblur(const Mat& input, int kernelSize);
    Mat local_operator(const Mat& input, const Mat& kernel);
```

```
    Mat sharpenImage(const Mat& inputImage, const Mat& blurredImage, double alpha);

    void computeGradients(const Mat& input, Mat& gradX, Mat& gradY);
    std::vector<cv::Point> fastDetector(const cv::Mat& image, int threshold);

    void customBoxFilter(const Mat& src, Mat& dst, Size kernelSize);
    void customMedianFilter(const Mat& src, Mat& dst, int kernelSize);
    void customGaussianFilter(const Mat& src, Mat& dst, Size kernelSize, double
sigma);
    void customSigmaFilter(const Mat& src, Mat& dst, int diameter, double
sigmaColor, double sigmaSpace);
    void computeGradients(const Mat& input, Mat& gradX, Mat& gradY);
    void harrisCornerDetector(const Mat& input, Mat& output, double k = 0.04);
}
```

Func.cpp:

```
#include "func.hpp"
#include <cstdint>
#include <opencv2/opencv.hpp>

namespace comp_vis

{
    std::vector<int> computeHistogram(const Mat& src) {

        int height = src.rows;
        int width = src.cols;
        int histSize = 256;

        std::vector<int> hist(histSize, 0);

        for (int i = 0; i < height; ++i) {
            for (int j = 0; j < width; ++j) {
                int pixelValue = src.at<uchar>(i, j);
                hist[pixelValue]++;
            }
        }

        return hist;
    }

std::vector<float> normalizeHistogram(const std::vector<int>& hist) {

    int histSize = hist.size();
    std::vector<float> cdf(histSize, 0);

    float totalPixels = 0;
    for (int i = 0; i < histSize; ++i) {
        totalPixels += hist[i];
        cdf[i] = totalPixels;
    }


    for (int i = 0; i < histSize; ++i) {
        cdf[i] = cdf[i] / totalPixels;
    }

    return cdf;
}

Mat histogramEqualization(const Mat& src) {
    Mat dst = src.clone();
```

```
    std::vector<int> hist = computeHistogram(src);


    std::vector<float> cdf = normalizeHistogram(hist);


    for (int i = 0; i < dst.rows; ++i) {
        for (int j = 0; j < dst.cols; ++j) {
            int pixelValue = src.at<uchar>(i, j);
            dst.at<uchar>(i, j) = static_cast<uchar>(cdf[pixelValue] * 255);
        }
    }


    return dst;
}

void displayHistogram(const std::vector<int>& hist, const std::string& title) {

    int histHeight = 400;
    int histWidth = 512;
    int binWidth = cvRound((double)histWidth / hist.size());

    Mat histogramImage(histHeight, histWidth, CV_8UC3, Scalar(255, 255, 255));


    for (size_t i = 0; i < hist.size(); i++) {

        int normalizedHeight = cvRound((double)hist[i] * histHeight /
*std::max_element(hist.begin(), hist.end()));
        rectangle(histogramImage, Point(i * binWidth, histHeight),
            Point((i + 1) * binWidth, histHeight - normalizedHeight),
            Scalar(0, 0, 255), -1);
    }

    imshow(title, histogramImage);

}

Mat myblur(const Mat& input, int kernelSize) {
    Mat output = input.clone();

    if (kernelSize % 2 == 0) {
        cout << "Kernel size must be odd." << endl;
        return output;
    }

    int offset = kernelSize / 2;

    for (int y = 0; y < input.rows; y++) {
        for (int x = 0; x < input.cols; x++) {
            double sum = 0.0;
            int count = 0;

            for (int j = -offset; j <= offset; j++) {
                for (int i = -offset; i <= offset; i++) {
                    int newY = min(max(y + j, 0), input.rows - 1);
                    int newX = min(max(x + i, 0), input.cols - 1);
                    sum += input.at<uchar>(newY, newX);
                    count++;
                }
            }

            output.at<uchar>(y, x) = saturate_cast<uchar>(sum / count);
```

```
        }
    }

    return output;
}

Mat local_operator(const Mat& input, const Mat& kernel) {
    Mat output = input.clone();

    int kernelSize = kernel.rows;
    if (kernelSize % 2 == 0) {
        cout << "Kernel size must be odd." << endl;
        return output;
    }

    int offset = kernelSize / 2;

    for (int y = 0; y < input.rows; y++) {
        for (int x = 0; x < input.cols; x++) {
            double sum = 0.0;

            for (int j = -offset; j <= offset; j++) {
                for (int i = -offset; i <= offset; i++) {
                    int newY = min(max(y + j, 0), input.rows - 1);
                    int newX = min(max(x + i, 0), input.cols - 1);
                    sum += input.at<uchar>(newY, newX) * kernel.at<double>(j +
offset, i + offset);
                }
            }

            output.at<uchar>(y, x) = saturate_cast<uchar>(sum);
        }
    }

    return output;
}

Mat sharpenImage(const Mat& inputImage, const Mat& blurredImage, double alpha) {

    Mat highPassImage = inputImage - blurredImage;
    Mat outputImage = inputImage + alpha * highPassImage;
    return outputImage;

}

void customNormalize(const Mat& input, Mat& output, double newMin, double newMax) {
    double minVal, maxVal;
    minMaxLoc(input, &minVal, &maxVal);

    output = Mat::zeros(input.size(), input.type());

    for (int y = 0; y < input.rows; y++) {
        for (int x = 0; x < input.cols; x++) {

            double normalizedValue = ((input.at<double>(y, x) - minVal) / (maxVal -
minVal)) * (newMax - newMin) + newMin;
            output.at<double>(y, x) = saturate_cast<uchar>(normalizedValue);
        }
    }
}

void computeGradients(const Mat& input, Mat& gradX, Mat& gradY) {

    double sobelX[3][3] = { {-1, 0, 1},
                            {-2, 0, 2},
```

```
                            {-1, 0, 1} };

    double sobelY[3][3] = { {-1, -2, -1},
                            {0, 0, 0},
                            {1, 2, 1} };


    int rows = input.rows;
    int cols = input.cols;
    gradX = Mat::zeros(rows, cols, CV_64F);
    gradY = Mat::zeros(rows, cols, CV_64F);

    for (int y = 1; y < rows - 1; y++) {
        for (int x = 1; x < cols - 1; x++) {
            double sumX = 0.0;
            double sumY = 0.0;

            for (int j = -1; j <= 1; j++) {
                for (int i = -1; i <= 1; i++) {
                    sumX += input.at<uchar>(y + j, x + i) * sobelX[j + 1][i + 1];
                    sumY += input.at<uchar>(y + j, x + i) * sobelY[j + 1][i + 1];
                }
            }

            gradX.at<double>(y, x) = sumX;
            gradY.at<double>(y, x) = sumY;
        }
    }
}

void harrisCornerDetector(const Mat& input, Mat& output, double k) {
    Mat gradX, gradY;
    computeGradients(input, gradX, gradY);


    Mat Ixx = gradX.mul(gradX);
    Mat Iyy = gradY.mul(gradY);
    Mat Ixy = gradX.mul(gradY);

    int rows = input.rows;
    int cols = input.cols;
    output = Mat::zeros(rows, cols, CV_64F);


    for (int y = 1; y < rows - 1; y++) {
        for (int x = 1; x < cols - 1; x++) {
            double sumIxx = 0.0, sumIyy = 0.0, sumIxy = 0.0;


            for (int j = -1; j <= 1; j++) {
                for (int i = -1; i <= 1; i++) {
                    sumIxx += Ixx.at<double>(y + j, x + i);
                    sumIyy += Iyy.at<double>(y + j, x + i);
                    sumIxy += Ixy.at<double>(y + j, x + i);
                }
            }


            double det = sumIxx * sumIyy - sumIxy * sumIxy;
            double trace = sumIxx + sumIyy;
            output.at<double>(y, x) = det - k * trace * trace;
        }
    }
```

```
    Mat normalizedOutput;
    customNormalize(output, normalizedOutput, 0, 255);
    normalizedOutput.convertTo(output, CV_8U);
}
    std::vector<cv::Point> fastDetector(const cv::Mat& image, int threshold) {
        std::vector<cv::Point> keypoints;
        int offset[16][2] = {
            {0, 3}, {1, 3}, {2, 2}, {3, 1}, {3, 0}, {3, -1}, {2, -2}, {1, -3},
            {0, -3}, {-1, -3}, {-2, -2}, {-3, -1}, {-3, 0}, {-3, 1}, {-2, 2}, {-1,
3}
        };

        for (int y = 3; y < image.rows - 3; ++y) {
            for (int x = 3; x < image.cols - 3; ++x) {
                int centerPixel = image.at<uchar>(y, x);
                int countBright = 0, countDark = 0;

                for (int i = 0; i < 16; ++i) {
                    int newY = y + offset[i][0];
                    int newX = x + offset[i][1];
                    int pixel = image.at<uchar>(newY, newX);

                    if (pixel > centerPixel + threshold) {
                        countBright++;
                    }
                    else if (pixel < centerPixel - threshold) {
                        countDark++;
                    }

                    if (countBright >= 12 || countDark >= 12) {

                        bool isUnique = true;
                        for (const auto& kp : keypoints) {
                            if (cv::norm(kp - cv::Point(x, y)) < 5) {
                                isUnique = false;
                                break;
                            }
                        }
                        if (isUnique) {
                            keypoints.push_back(cv::Point(x, y));
                        }
                        break;
                    }
                }
            }
        }
        return keypoints;
    }

    void customBoxFilter(const Mat& src, Mat& dst, Size kernelSize)
    {
        dst = Mat::zeros(src.size(), src.type());
        int kx = kernelSize.width / 2;
        int ky = kernelSize.height / 2;
        for (int y = ky; y < src.rows - ky; y++)
        {
            for (int x = kx; x < src.cols - kx; x++)
            {
                float sum = 0.0;
                for (int j = -ky; j <= ky; j++)
                {
                    for (int i = -kx; i <= kx; i++)
                    {
                        sum += src.at<uchar>(y + j, x + i);
                    }
```

```
                }
                dst.at<uchar>(y, x) = sum / (kernelSize.width * kernelSize.height);
            }
        }
    }

    void customMedianFilter(const Mat& src, Mat& dst, int kernelSize)
    {
        dst = Mat::zeros(src.size(), src.type());
        int k = kernelSize / 2;
        for (int y = k; y < src.rows - k; y++)
        {
            for (int x = k; x < src.cols - k; x++)
            {
                std::vector<uchar> neighborhood;
                for (int j = -k; j <= k; j++)
                {
                    for (int i = -k; i <= k; i++)
                    {
                        neighborhood.push_back(src.at<uchar>(y + j, x + i));
                    }
                }
                sort(neighborhood.begin(), neighborhood.end());
                dst.at<uchar>(y, x) = neighborhood[neighborhood.size() / 2];
            }
        }
    }

    void customGaussianFilter(const Mat& src, Mat& dst, Size kernelSize, double
sigma)
    {
        dst = Mat::zeros(src.size(), src.type());
        int kx = kernelSize.width / 2;
        int ky = kernelSize.height / 2;
        std::vector<std::vector<double>> kernel(kernelSize.height,
vector<double>(kernelSize.width));
        double sum = 0.0;


        for (int y = -ky; y <= ky; y++)
        {
            for (int x = -kx; x <= kx; x++)
            {
                kernel[y + ky][x + kx] = exp(-(x * x + y * y) / (2 * sigma *
sigma));
                sum += kernel[y + ky][x + kx];
            }
        }

        for (int y = 0; y < kernelSize.height; y++)
        {
            for (int x = 0; x < kernelSize.width; x++)
            {
                kernel[y][x] /= sum;
            }
        }


        for (int y = ky; y < src.rows - ky; y++)
        {
            for (int x = kx; x < src.cols - kx; x++)
            {
                float sum = 0.0;
                for (int j = -ky; j <= ky; j++)
                {
```

```
                    for (int i = -kx; i <= kx; i++)
                    {
                        sum += src.at<uchar>(y + j, x + i) * kernel[j + ky][i + kx];
                    }
                }
                dst.at<uchar>(y, x) = sum;
            }
        }
    }

    void customSigmaFilter(const Mat& src, Mat& dst, int diameter, double
sigmaColor, double sigmaSpace)
    {
        dst = Mat::zeros(src.size(), src.type());
        int radius = diameter / 2;
        for (int y = radius; y < src.rows - radius; y++)
        {
            for (int x = radius; x < src.cols - radius; x++)
            {
                float sum = 0.0;
                float weightSum = 0.0;
                uchar center = src.at<uchar>(y, x);
                for (int j = -radius; j <= radius; j++)
                {
                    for (int i = -radius; i <= radius; i++)
                    {
                        uchar neighbor = src.at<uchar>(y + j, x + i);
                        double spaceWeight = exp(-(i * i + j * j) / (2 * sigmaSpace
* sigmaSpace));
                        double colorWeight = exp(-((neighbor - center) * (neighbor -
center)) /
                            (2 * sigmaColor * sigmaColor));
                        double weight = spaceWeight * colorWeight;
                        sum += neighbor * weight;
                        weightSum += weight;
                    }
                }
                dst.at<uchar>(y, x) = sum / weightSum;
            }
        }
    }


}
```

Task.cpp:

```
    #include <opencv2/opencv.hpp>

    #include "func.hpp"
    #include "task.hpp"

    using namespace cv;
    using namespace std;

    void task1(int subtask)
    {
        Mat img =
    imread("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\input\\image.jpg"
    , IMREAD_GRAYSCALE);

        if (img.empty()) {
            cout << "Could not read the image" << endl;
            return;
```

```
    }

    switch (subtask) {

    case 1: {
        vector<int> originalHist = comp_vis::computeHistogram(img);
        comp_vis::displayHistogram(originalHist, "Task 1 MY Original
Histogram");

        Mat equalizedImg = comp_vis::histogramEqualization(img);
        vector<int> equalizedHist =
comp_vis::computeHistogram(equalizedImg);
        comp_vis::displayHistogram(equalizedHist, "task 1 MY Equalized
Histogram");

        cout << "Press any key to return to menu..." << endl;
        waitKey(0);
        break;
    }
    case 2: {
        Mat equalizedImg = comp_vis::histogramEqualization(img);

        imshow("Task 1 Original Image", img);
        moveWindow("Task 1 Original Image", 0, 0);

        imshow("Task 1 MY Equalized Image", equalizedImg);
        moveWindow("Task 1 MY Equalized Image", 500, 0);


imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_image
_equalized.jpg", equalizedImg);

        Mat cv_equ_hist = img.clone();
        equalizeHist(img, cv_equ_hist);

        imshow("Task 1 OpenCV Equalized Image", cv_equ_hist);
        moveWindow("Task 1 OpenCV Equalized Image", 1000, 0);

        cout << "Press any key to return to menu..." << endl;
        waitKey(0);
        break;
    }
    default:
        cout << "Invalid subtask choice." << endl;
        break;
    }
}

void task3(int subtask)
{
    Mat img =
imread("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\input\\image_2.jp
g", IMREAD_GRAYSCALE);

    if (img.empty()) {
        cout << "Could not read the image" << endl;
        return;
    }

    Mat kernel_box = (Mat_<double>(5, 5) << 1, 1, 1, 1, 1,
                                            1, 1, 1, 1, 1,
                                            1, 1, 1, 1, 1,
                                            1, 1, 1, 1, 1,
                                            1, 1, 1, 1, 1) / 25.0;
```

```
    Mat kernel_sobelX = (Mat_<double>(5, 5) <<
        -2, -1, 0, 1, 2,
        -4, -3, 0, 3, 4,
        -6, -5, 0, 5, 6,
        -4, -3, 0, 3, 4,
        -2, -1, 0, 1, 2);
   /*
    Mat kernel_sobelX = (Mat_<double>(5, 5) <<
        -1, -2, 0, 2, 1,
        -4, -8, 0, 8, 4,
        -6, -12, 0, 12, 6,
        -4, -8, 0, 8, 4,
        -1, -2, 0, 2, 1);*/

    /*Mat kernel_sobelX = (Mat_<double>(5, 5) <<
        -1, -2, 0, 2, 1,
        -4, -6, 0, 6, 4,
        -6, -10, 0, 10, 6,
        -4, -6, 0, 6, 4,
        -1, -2, 0, 2, 1);*/


   /* Mat kernel_sobelX = (Mat_<double>(3, 3) <<
        -1, 0, 1,
        -2, 0, 2,
        -1, 0, 1);*/

    switch (subtask) {
    case 1: {
        Mat image_box = comp_vis::local_operator(img, kernel_box);

        imshow("Task 3 Original Image", img);
        moveWindow("Task 3 Original Image", 0, 0);

        imshow("Task 3 MY local operator box Blurred Image", image_box);
        moveWindow("Task 3 MY local operator box Blurred Image", 500, 0);


imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_blurred_image.jpg", image_box);

        Mat result;
        blur(img, result, Size(5, 5));
        imshow("Task 3  OpenCV Blur Image", result);
        moveWindow("Task 3  OpenCV Blur Image", 1000, 0);

        waitKey(0);
        break;
    }
    case 2: {
        imshow("Task 3 Original Image", img);
        moveWindow("Task 3 Original Image", 0, 0);

        Mat image_sobelX = comp_vis::local_operator(img, kernel_sobelX);
        imshow("Task 3 MY local operator SobelX Image", image_sobelX);
        moveWindow("Task 3 MY local operator SobelX Image", 500, 0);


imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_sobelX_image.jpg", image_sobelX);

        Mat grad_x;
        Sobel(img, grad_x, CV_8U, 1, 0, 5);
```

```
        Mat abs_grad_x;
        convertScaleAbs(grad_x, abs_grad_x);
        imshow("Task 3 OpenCV SobelX Image", abs_grad_x);
        moveWindow("Task 3 OpenCV SobelX Image", 1000, 0);

        waitKey(0);
        break;
    }
    default:
        cout << "Invalid subtask choice." << endl;
        break;
    }
}


void task5(int subtask)
{
    Mat img =
imread("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\input\\image_31.j
pg", IMREAD_GRAYSCALE);

    if (img.empty()) {
        cout << "Could not read the image" << endl;
        return;
    }

    double alpha = 1.5;
    Size kernelSize(5, 5);
    double sigma = 1.0;
    int sigmaFilterDiameter = 9;
    double sigmaColor = 75.0;
    double sigmaSpace = 75.0;

    Mat blurredImage, sharpenedImageBox, sharpenedImageMedian,
sharpenedImageGaussian, sharpenedImageSigma;

    int startX = 20;
    int windowOffset = 500;

    switch (subtask) {
    case 1:

        comp_vis::customBoxFilter(img, blurredImage, kernelSize);
        sharpenedImageBox = comp_vis::sharpenImage(img, blurredImage,
alpha);
        imshow("Original Image", img);
        moveWindow("Original Image", startX, 20);
        imshow("Blurred (Box Filter)", blurredImage);
        moveWindow("Blurred (Box Filter)", startX + windowOffset, 20);
        imshow("Sharpened (Box Filter)", sharpenedImageBox);
        moveWindow("Sharpened (Box Filter)", startX + 2 * windowOffset,
20);
        waitKey(0);
        break;

    case 2:

        comp_vis::customMedianFilter(img, blurredImage,
kernelSize.width);
        sharpenedImageMedian = comp_vis::sharpenImage(img, blurredImage,
alpha);
        imshow("Original Image", img);
        moveWindow("Original Image", startX, 20);
        imshow("Blurred (Median Filter)", blurredImage);
```

```
        moveWindow("Blurred (Median Filter)", startX + windowOffset, 20);
        imshow("Sharpened (Median Filter)", sharpenedImageMedian);
        moveWindow("Sharpened (Median Filter)", startX + 2 *
windowOffset, 20);
        waitKey(0);
        break;

    case 3:

        comp_vis::customGaussianFilter(img, blurredImage, kernelSize,
sigma);
        sharpenedImageGaussian = comp_vis::sharpenImage(img,
blurredImage, alpha);
        imshow("Original Image", img);
        moveWindow("Original Image", startX, 20);
        imshow("Blurred (Gaussian Filter)", blurredImage);
        moveWindow("Blurred (Gaussian Filter)", startX + windowOffset,
20);
        imshow("Sharpened (Gaussian Filter)", sharpenedImageGaussian);
        moveWindow("Sharpened (Gaussian Filter)", startX + 2 *
windowOffset, 20);
        waitKey(0);
        break;

    case 4:

        comp_vis::customSigmaFilter(img, blurredImage,
sigmaFilterDiameter, sigmaColor, sigmaSpace);
        sharpenedImageSigma = comp_vis::sharpenImage(img, blurredImage,
alpha);
        imshow("Original Image", img);
        moveWindow("Original Image", startX, 20);
        imshow("Blurred (Sigma Filter)", blurredImage);
        moveWindow("Blurred (Sigma Filter)", startX + windowOffset, 20);
        imshow("Sharpened (Sigma Filter)", sharpenedImageSigma);
        moveWindow("Sharpened (Sigma Filter)", startX + 2 * windowOffset,
20);
        waitKey(0);
        break;

    case 5:

        comp_vis::customBoxFilter(img, blurredImage, kernelSize);
        sharpenedImageBox = comp_vis::sharpenImage(img, blurredImage,
alpha);

        comp_vis::customMedianFilter(img, blurredImage,
kernelSize.width);
        sharpenedImageMedian = comp_vis::sharpenImage(img, blurredImage,
alpha);

        comp_vis::customGaussianFilter(img, blurredImage, kernelSize,
sigma);
        sharpenedImageGaussian = comp_vis::sharpenImage(img,
blurredImage, alpha);

        comp_vis::customSigmaFilter(img, blurredImage,
sigmaFilterDiameter, sigmaColor, sigmaSpace);
        sharpenedImageSigma = comp_vis::sharpenImage(img, blurredImage,
alpha);

        imshow("Original Image", img);
        moveWindow("Original Image", startX, 20);
```

```
        imshow("Sharpened (Box Filter)", sharpenedImageBox);
        moveWindow("Sharpened (Box Filter)", startX + windowOffset, 20);

        imshow("Sharpened (Median Filter)", sharpenedImageMedian);
        moveWindow("Sharpened (Median Filter)", startX + 2 *
windowOffset, 20);

        imshow("Sharpened (Gaussian Filter)", sharpenedImageGaussian);
        moveWindow("Sharpened (Gaussian Filter)", startX + 3 *
windowOffset, 20);

        imshow("Sharpened (Sigma Filter)", sharpenedImageSigma);
        moveWindow("Sharpened (Sigma Filter)", 20, 500);


imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_sharp
enedImageBox.jpg", sharpenedImageBox);

imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_sharp
enedImageMedian.jpg", sharpenedImageMedian);

imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_sharp
enedImageGaussian.jpg", sharpenedImageGaussian);

imwrite("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\output\\my_sharp
enedImageSigma.jpg", sharpenedImageSigma);

        waitKey(0);
        break;

    default:
        cout << "Invalid choice!" << endl;
        return;
    }
}


void task7(int subtask)
{
    switch (subtask) {
    case 1:
    {
        Mat image =
imread("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\input\\image_74.j
pg");
        if (image.empty()) {
            cout << "Could not read the image" << endl;
            return;
        }

        Mat grayImage;
        cvtColor(image, grayImage, COLOR_BGR2GRAY);
        int blockSize = 2;
        int ksize = 3;
        double k = 0.04;
        double threshold = 100;

        Mat harrisCorners;
        cornerHarris(grayImage, harrisCorners, blockSize, ksize, k,
BORDER_DEFAULT);

        Mat harrisCornersNorm;
        normalize(harrisCorners, harrisCornersNorm, 0, 255, NORM_MINMAX,
CV_32FC1);
```

```
        Mat outputImage = image.clone();
        for (int y = 0; y < harrisCornersNorm.rows; y++) {
            for (int x = 0; x < harrisCornersNorm.cols; x++) {
                if (harrisCornersNorm.at<float>(y, x) > threshold) {

                    circle(outputImage, Point(x, y), 3, Scalar(0, 255,
0), 1);
                }
            }
        }

        imshow("Origin", image);
        imshow("Opencv Harris Corners", outputImage);

        Mat myharrisCorners;
        comp_vis::harrisCornerDetector(image, harrisCorners);

        imshow("MY Harris Corners", harrisCorners);



        waitKey(0);
    }
    break;
    case 2:
    {
        Mat image =
imread("D:\\Study\\4_curse_1_sem\\CV\\comp_vis\\images\\input\\image7.jpg
",IMREAD_GRAYSCALE);

        if (image.empty()) {
            cout << "Could not read the image" << endl;
            return;
        }

        vector<KeyPoint> keypoints;
        int threshold = 40;


        Ptr<FastFeatureDetector> fast =
FastFeatureDetector::create(threshold);
        fast->detect(image, keypoints);

        Mat outputImage;
        drawKeypoints(image, keypoints, outputImage, Scalar(0, 255, 0));

        imshow("ORIGINAL IMAGE", image);
        imshow("FAST Corners Opencv", outputImage);

        int threshold_1 = 40;

        vector<Point> corners = comp_vis::fastDetector(image,
threshold_1);
        Mat outputImageWithCorners;
        cvtColor(image, outputImageWithCorners, COLOR_GRAY2BGR);

        for (const auto& point : corners) {
            circle(outputImageWithCorners, point, 4, Scalar(0, 255, 0),
1);
        }

        imshow("MY FAST Corners", outputImageWithCorners);
```

```
            waitKey(0);
        }
        break;
        default:
            cout << "Invalid subtask choice." << endl;
            break;
        }

    }
```

Task.hpp:

```
#pragma once
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

void task1(int subtask);
void task3(int subtask);
void task5(int subtask);
void task7(int subtask);
```

app.cpp:

```
#include <opencv2/opencv.hpp>
#include "func.hpp"
#include "task.hpp"

using namespace cv;
using namespace std;


int main() {

    int subtask;
    int taskNumber;

    do {
        cout << "Select a task to execute:" << endl;
        cout << "1. Task 1: Histogram Equalization" << endl;
        cout << "3. Task 3: Linear local operator (given filter kernel)" <<
endl;
        cout << "5. Task 5: Unsharp macking" << endl;
        cout << "7. Task 7: Harris and FAST angle detectors" << endl;
        cout << "0. Exit" << endl;

        cin >> taskNumber;

        switch (taskNumber) {
        case 1: {

            do {
                cout << "Select what you want to display:" << endl;
                cout << "1. Display histograms" << endl;
                cout << "2. Display images" << endl;
                cout << "0. Go back to main menu" << endl;
                cin >> subtask;

                if (subtask != 0) {
```

```
            task1(subtask);
        }

    } while (subtask != 0);
    break;
}

case 3:
    do {
        cout << "Select what you want to display:" << endl;
        cout << "1. Display images blur" << endl;
        cout << "2. Display images Sobel_x" << endl;
        cout << "0. Go back to main menu" << endl;
        cin >> subtask;

        if (subtask != 0) {

            task3(subtask);
        }

    } while (subtask != 0);
    break;
case 4:

    break;
case 5: {

    do {
        cout << "Select what you want to display:" << endl;
        cout << "1. Display images box filter" << endl;
        cout << "2. Display images median filter" << endl;
        cout << "3. Display images gaussian filter" << endl;
        cout << "4. Display images sigma filter" << endl;
        cout << "5. Show all sharpened images" << endl;
        cout << "0. Go back to main menu" << endl;
        cin >> subtask;

        if (subtask != 0) {

            task5(subtask);
        }

    } while (subtask != 0);

}
      break;
case 7: {

    do {
        cout << "Select what you want to display:" << endl;
        cout << "1. Display images HARRIS" << endl;
        cout << "2. Display images FAST" << endl;
        cout << "0. Go back to main menu" << endl;
        cin >> subtask;

        if (subtask != 0) {

            task7(subtask);
        }

    } while (subtask != 0);

}
      break;
```

```
            case 0:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid task number!" << endl;
                break;
            }

    } while (taskNumber != 0);

    return 0;
}
```