

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

Лабораторна робота №2
з дисципліни «Комп'ютерний зір»

«Аналіз вмісту 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. В довільному редакторі зображень створити зображення зі зв'язною ділянкою певного кольору (наприклад, біла ділянка на чорному фоні). Виконати подальші експерименти, що містять ділянками різною форми.
2. Обчислити та вивести площу цієї ділянки.
3. Знайти та вивести центроїд, головну вісь та ексцентриситет цієї ділянки (візуалізувати знайдені центроїд та головну вісь, як пряму, що проходить через центроїд).
4. Виконати трасування краю (контур) цієї ділянки (реалізувавши алгоритм Восса). Візуалізувати його, обчислити його довжину та вивести її значення.
5. Оцінити кривизну контуру в різних точках. Вивести максимальне, мінімальне та середнє значення кривизни (абсолютного її значення).

Показати точки із максимальною та мінімальною кривизною (наприклад, відмітивши їх певними кольорами).

6. Побудувати та візуалізувати дистанційне перетворення для вищеописаної ділянки (оптимізована реалізація є необов'язковою).
7. Знайти та візуалізувати лінії на зображенні за допомогою стандартного перетворення Хафа (обрати зображення із наявними чіткими рівними лініями).

Додаткові завдання (за бажанням):

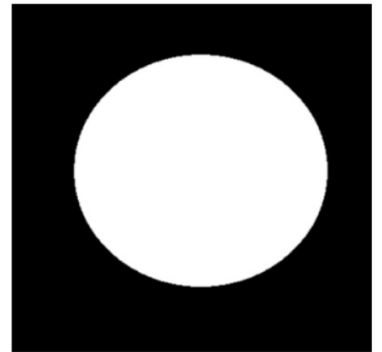
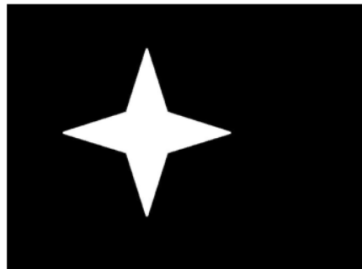
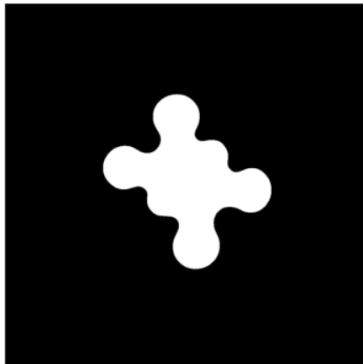
1. Порівняти реалізовані алгоритми із наявними в OpenCV (зокрема, `cv::findContours`, `cv::distanceTransform`, `cv::HoughLines`, `cv::HoughCircles`).

Порядок виконання роботи

Завдання 1

В довільному редакторі зображень створити зображення зі зв'язною ділянкою певного кольору (наприклад, біла ділянка на чорному фоні). Виконати подальші експерименти, що містять ділянками різною форми.

Для даного завдання використаємо такий графічний редактор, як Paint. В нього доволі простий інтерфейс та просте розуміння інструментів. Отримали таких три зображення, з якими будемо працювати надалі:



Загалом це завдання не містить нічого складного, тому заціклювати увагу на цьому не будемо.

Завдання 2

Обчислити та вивести площу цієї ділянки.

У цьому завданні я створюю дві функції: `get_area` та `print_areas_info`, які обчислюють і виводять площу білої ділянки(нашої фігури) зображень у пікселях та відсотках.

Функція `get_area` приймає на вхід зображення у вигляді масиву пікселів `img`. Вона створює маску `mask`, яка позначає пікселі, що мають значення інтенсивності понад 127 (тобто, значення, близькі до білого кольору). Після цього функція повертає суму значень маски (кількість білих пікселів), яка визначає площу білої ділянки в пікселях, та середнє значення маски (частка білих пікселів від загальної кількості), що використовується для обчислення відсоткової площі.

Функція `print_areas_info` приймає список зображень `imgs` і проходить по кожному з них. Для кожного зображення вона викликає функцію `get_area`, обчислює площу білих ділянок і виводить цю інформацію. Виведення містить номер зображення, площу білих ділянок у пікселях та її частку в процентах з точністю до п'яти знаків після коми.

Результати виконання:

```
Img1
white area (in pixels): 103880
white area (in percents): 0.09907%
```

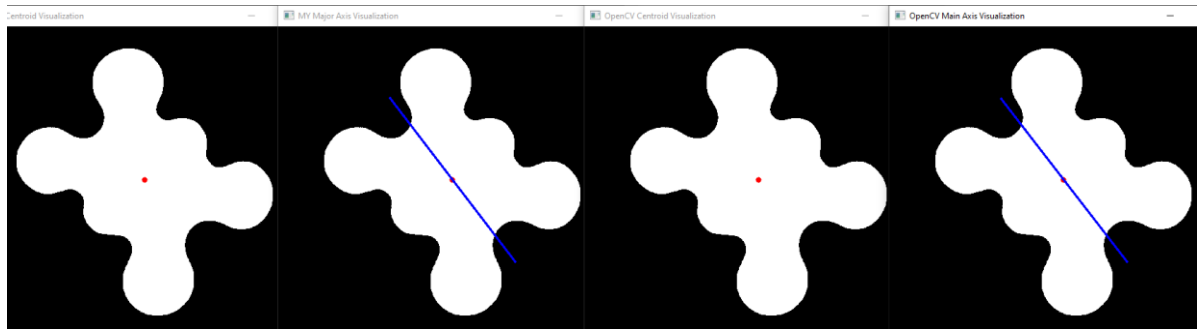
```
Img2
white area (in pixels): 25096
white area (in percents): 0.07567%
```

```
Img3
white area (in pixels): 57285
white area (in percents): 0.36727%
```

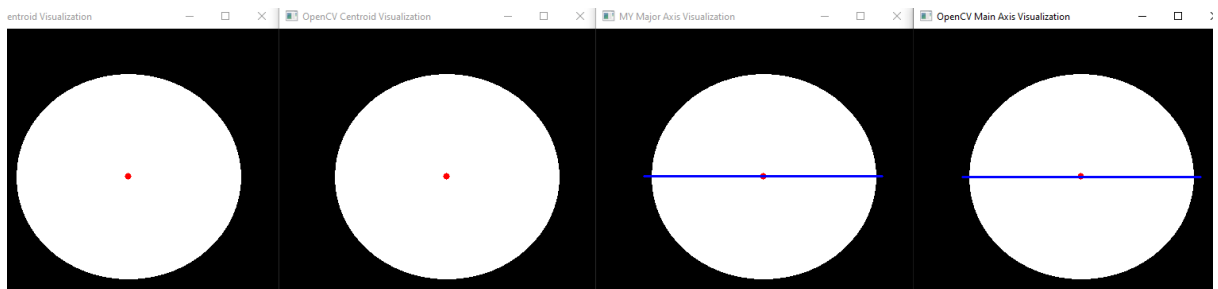
Загалом порівняння з OpenCV не матиме ніякої нагальності, так як код легкий для розуміння і не важко візуально впевнитись у вірності розрахунків.

Завдання 2

Знайти та вивести центроїд, головну вісь та ексцентриситет цієї ділянки (візуалізувати знайдені центроїд та головну вісь, як пряму, що проходить через центроїд).

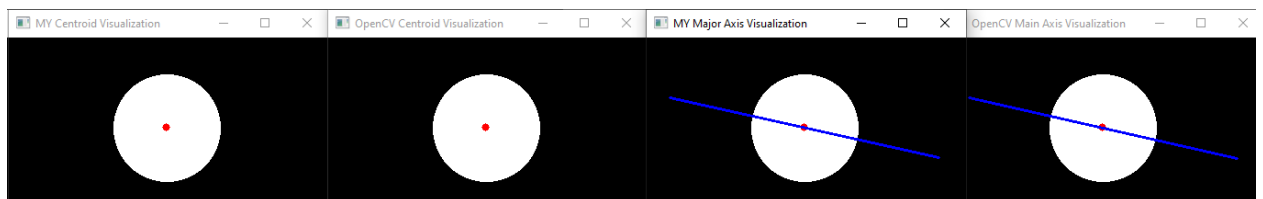


```
My Centroid: (253, 222)
OpenCV Centroid: (253, 222)
My major axis start: (162.118, 103.58), end: (345.501, 341.005)
OpenCV major axis start: (345.503, 341.004), end: (162.116, 103.582)
OpenCV eccentricity: 0.638968
My eccentricity: 0.638972
```



```
My Centroid: (210, 185)
OpenCV Centroid: (210, 185)
My major axis start: (60.7503, 185.761), end: (360.75, 185.752)
OpenCV major axis start: (60.7503, 185.761), end: (360.75, 185.752)
OpenCV eccentricity: 0.410923
My eccentricity: 0.410958
```

Як бачимо, дане коло не є ідеальним, щоб перевірити усі випадки створимо коло яке є більш ідеальним.



```
My Centroid: (171, 97)
OpenCV Centroid: (171, 97)
My major axis start: (25.5493, 65.5373), end: (318.44, 130.46)
OpenCV major axis start: (25.75, 64.6447), end: (318.239, 131.353)
OpenCV eccentricity: 0.0193026
My eccentricity: 0
```

Центроїд (геометричний центр) об'єкта на зображенні визначається як середнє значення координат усіх пікселів, що належать до об'єкта. Центроїд часто використовується як контрольна точка в задачах виявлення об'єктів, відстеження та сегментації.

Головна вісь об'єкта вказує на його орієнтацію в просторі. У нашій реалізації ми визначили головну вісь, аналізуючи коваріаційну матрицю координат пікселів, що належать до об'єкта.

Ексцентриситет визначає, наскільки об'єкт відхиляється від ідеальної круглої форми. Він обчислюється на основі власних значень коваріаційної матриці. Цей показник корисний для класифікації об'єктів, оскільки різні форми можуть вказувати на різні класові характеристики.

Порівняння з функціями OpenCV:

Загалом в OpenCV немає конкретних функцій для обчислення центроїду, головної вісі та ексцентриситету цієї ділянки. Але є деякі функції, які можуть полегшити підрахунок, в нашому випадку `moments()`. Власна реалізація обчислює центроїд, головну вісь та ексцентриситет шляхом обходу пікселів бінарного зображення, підрахунку сум, що стосуються координат пікселів, та обчислення коваріаційної матриці. У вбудованій використовуємо функцію `moments()`, яка автоматично обчислює моменти зображення. Для головної осі OpenCV спрощує задачу, використовуючи функцію `eigen()`, щоб знайти власні значення та вектори без ручних обчислень.

Моя реалізація може бути менш оптимізованою, оскільки вимагає ручного перебору пікселів, що може призводити до повільнішої роботи при великих зображеннях. OpenCV оптимізує обчислення, використовуючи алгоритми, які можуть обробляти зображення швидше та ефективніше, завдяки внутрішнім оптимізаціям.

Щодо зображень то результати цих двох методів практично збігаються, що підтверджує коректність реалізації алгоритму.

Завдання 4

Виконати трасування краю (контур) цієї ділянки (реалізувавши алгоритм Восса). Візуалізувати його, обчислити його довжину та вивести її значення.

У цьому завданні реалізовано алгоритм трасування контуру на основі методу Мура (алгоритм Восса), який використовується для знаходження межі об'єкта на бінарному зображенні. Спочатку ми перетворюємо зображення на маску, де пікселі об'єкта мають значення 255, а фон – 0. Потім визначається початковий піксель контуру, з якого починається трасування.

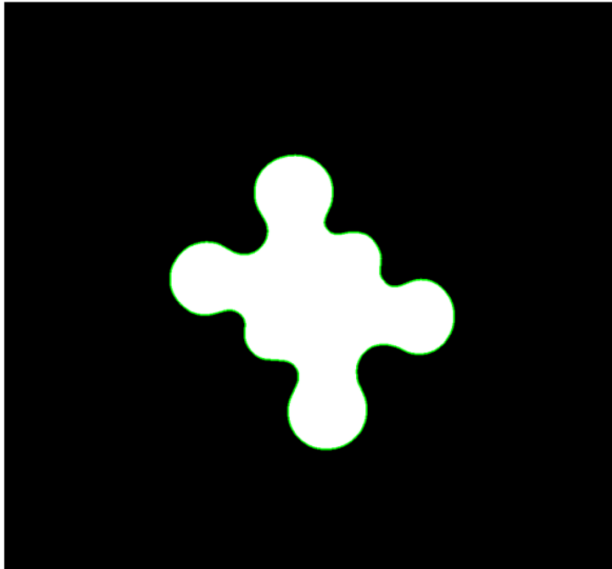
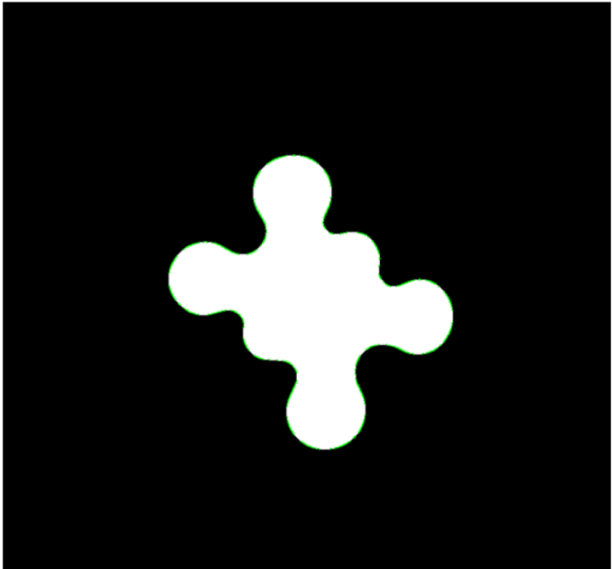

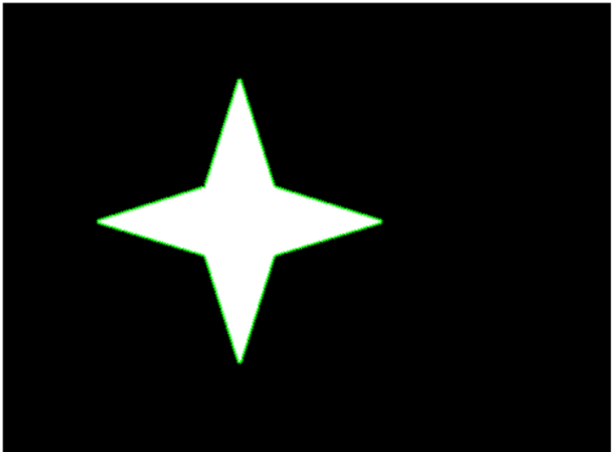
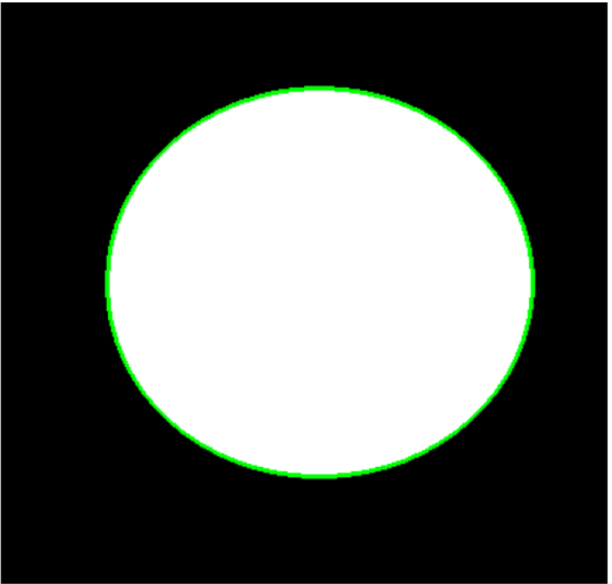
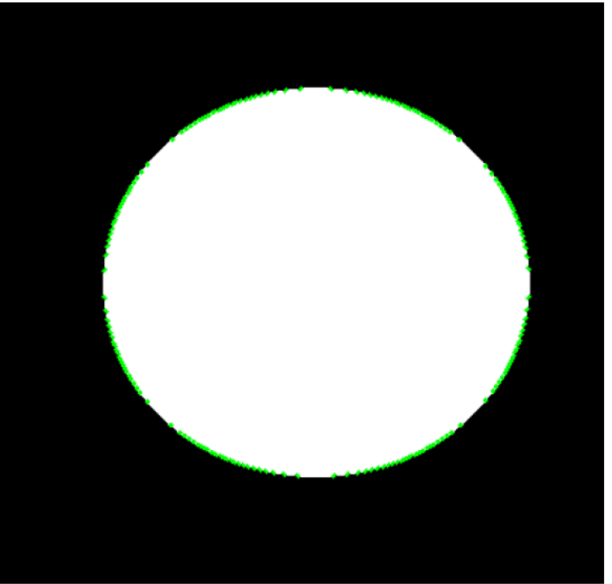
Алгоритм рухається вздовж пікселів, що складають контур, за допомогою заданого напрямку, а також перевіряє сусідні пікселі (8 можливих напрямків). Починаючи з певного пікселя, він обирає наступний сусідній піксель, який також є частиною об'єкта (знову зі значенням 255), тим самим формуючи контур. При кожному кроці обраний піксель додається до масиву контуру і маркується в зображенні як "пройдений" (із значенням 128), щоб уникнути повторного проходу. Алгоритм завершується, коли піксель повертається до початкового значення, що означає завершення циклу обхідного маршруту.

Для обчислення довжини отриманого контуру використовується евклідова відстань між усіма суміжними пікселями, що складають контур.

Код також використовує метод OpenCV для трасування контуру, який автоматизує пошук та побудову контуру. Метод `cv2.findContours` знаходить всі зовнішні контури об'єкта в бінарному зображенні, а потім обирається найдовший контур. Довжина контуру обчислюється аналогічно – шляхом обчислення відстаней між суміжними точками на контурах.

Для візуалізації результатів кожен піксель контуру позначається зеленим кольором на копії маски зображення. Обидва методи трасування порівнюються за візуальним результатом і обчисленою довжиною контуру, яка виводиться на зображенні для обох методів.

Результати виконання:

<p>Image 1 - Custom Contour Length: 1864.73 pixels</p> 	<p>Image 1 - OpenCV Contour Length: 1864.14 pixels</p> 
<p>Image 2 - Custom Contour Length: 1068.26 pixels</p> 	<p>Image 2 - OpenCV Contour Length: 1067.68 pixels</p> 
<p>Image 3 - Custom Contour Length: 891.48 pixels</p> 	<p>Image 3 - OpenCV Contour Length: 873.89 pixels</p> 

Як бачимо, в деяких випадках, як от на зображенні з колом видно, що метод OpenCV відображає контур із меншими деталями, ніж реалізація алгоритму Мура. Це пов'язано з тим, що `cv2.findContours` використовує параметр `cv2.CHAIN_APPROX_SIMPLE`, який оптимізує збереження контурних точок. Цей режим зберігає лише ключові точки та відкидає зайві пікселі на прямих відрізках, щоб зменшити кількість точок. Таким чином, контур OpenCV виглядає менш гладким та детальним, особливо на круглих або криволінійних контурах.

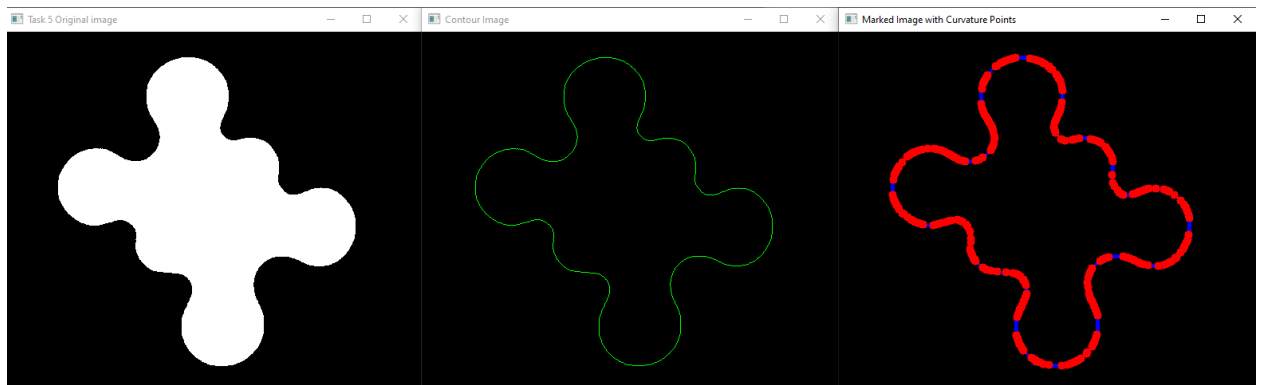
В алгоритмі Мура, натомість, кожен піксель контуру враховується, що забезпечує більш точне відображення, зокрема для круглих об'єктів. Це також пояснює невелику різницю в обчисленій довжині контуру між двома методами: оскільки метод OpenCV використовує менше точок, довжина контуру може бути трохи меншою через втрату деяких деталей.

Порівняння з функціями OpenCV:

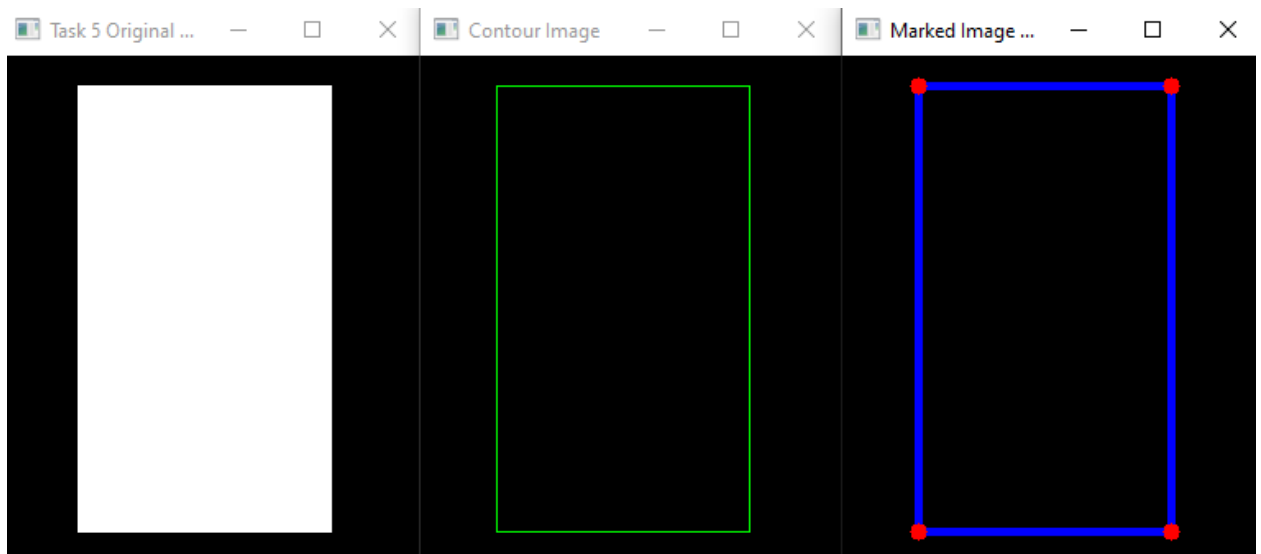
Метод Восса, реалізований вручну, забезпечує контроль над логікою проходження контуру, що може бути корисним у специфічних задачах або при потребі в індивідуальних налаштуваннях алгоритму. Водночас метод OpenCV більш оптимізований для загального застосування і забезпечує швидший, надійніший результат для стандартних випадків. Він також автоматично враховує особливості зображення та контурів, що зменшує потребу в ручному налаштуванні.

Завдання 5

Оцінити кривизну контуру в різних точках. Вивести максимальне, мінімальне та середнє значення кривизни (абсолютного її значення). Показати точки із максимальною та мінімальною кривизною (наприклад, відмітивши їх певними кольорами).



```
Maximum curvature: 1
Minimum curvature: 0
Average curvature: 0.31956
```



```
Maximum curvature: 1
Minimum curvature: 0
Average curvature: 0.00470588
Max Curvature Points:
Point: (46, 18)
Point: (46, 289)
Point: (200, 289)
Point: (200, 18)
```

Визначення кривизни контуру в різних точках є важливим етапом аналізу форм і структур в комп'ютерному зорі. Кривизна дозволяє зрозуміти, як форма змінюється у просторі, що може бути критично важливим для розпізнавання об'єктів, сегментації зображень та інших завдань обробки зображень. У нашій реалізації ми використовуємо три точки з контуру (попередню, поточну та наступну) для обчислення кривизни, що дозволяє виявити зміни в напрямку контуру. Вектори, що виходять з цих точок, використовуються для обчислення зміни напрямку контуру. Зміна напрямку обчислюється за допомогою формули, яка включає обчислення площі паралелограма, що описується цими векторами. Чисельник формули визначає абсолютне значення цієї площі, а знаменник масштабує значення, щоб взяти до уваги довжину вектора. Це дозволяє отримати значення кривизни, яке не залежить від масштабування контуру. Максимальна та мінімальна кривизна вказують на точки з найзначнішими та найменшими змінами форми, відповідно.

У ході обчислення ми отримуємо максимальне, мінімальне та середнє значення кривизни, що дає змогу оцінити загальні характеристики контуру. Відзначення точок з максимальними та мінімальними значеннями кривизни кольоровими маркерами на зображенні дозволяє візуально проаналізувати ці особливості. Це може допомогти в виявленні ключових особливостей форм, які можуть бути важливими для інших алгоритмів обробки зображень. У результаті, обчислення кривизни забезпечує детальнішу інформацію про геометрію об'єкта, що може бути корисним для різних застосувань у комп'ютерному зорі.

Завдання 6

Побудувати та візуалізувати дистанційне перетворення для вищеописаної ділянки (оптимізована реалізація є необов'язковою).

У даному завданні реалізовано власний метод обчислення дистанційного перетворення з використанням метрики для ділянки зображення. Спершу створюється двовимірний масив, де всі непорожні (не нульові) елементи ініціалізуються як нескінченність, а елементи, рівні нулю, позначаються як такі, що не потребують обчислення (відстань 0). Далі використовується двопрхідний алгоритм. Перший прохід виконується з верхнього лівого кута зображення до нижнього правого, де для кожного пікселя обчислюється мінімальна відстань до найближчого нульового елемента на основі обраного ядра (метрики). Другий прохід здійснюється у зворотному напрямку — з нижнього правого кута до верхнього лівого, — щоб забезпечити точне обчислення відстаней у всіх напрямках.

Різні метрики (Манхеттенська, Евклідова та Чебишева) задаються як ядра із різними вагами. В кожному ядрі вказані зсуви координат та ваги для обчислення відстані до сусідніх пікселів у залежності від обраної метрики. Далі код послідовно застосовує власний алгоритм дистанційного перетворення до кожного зображення та метрики й порівнює результати з результатами вбудованої функції `cv2.distanceTransform` з бібліотеки OpenCV, що реалізує аналогічне перетворення.

Для кожного зображення створюються три графічні підплоти: початкове зображення, результат власного методу дистанційного перетворення та результат перетворення з OpenCV. Це дозволяє візуально порівняти ефективність та точність реалізації.

Результати виконання:

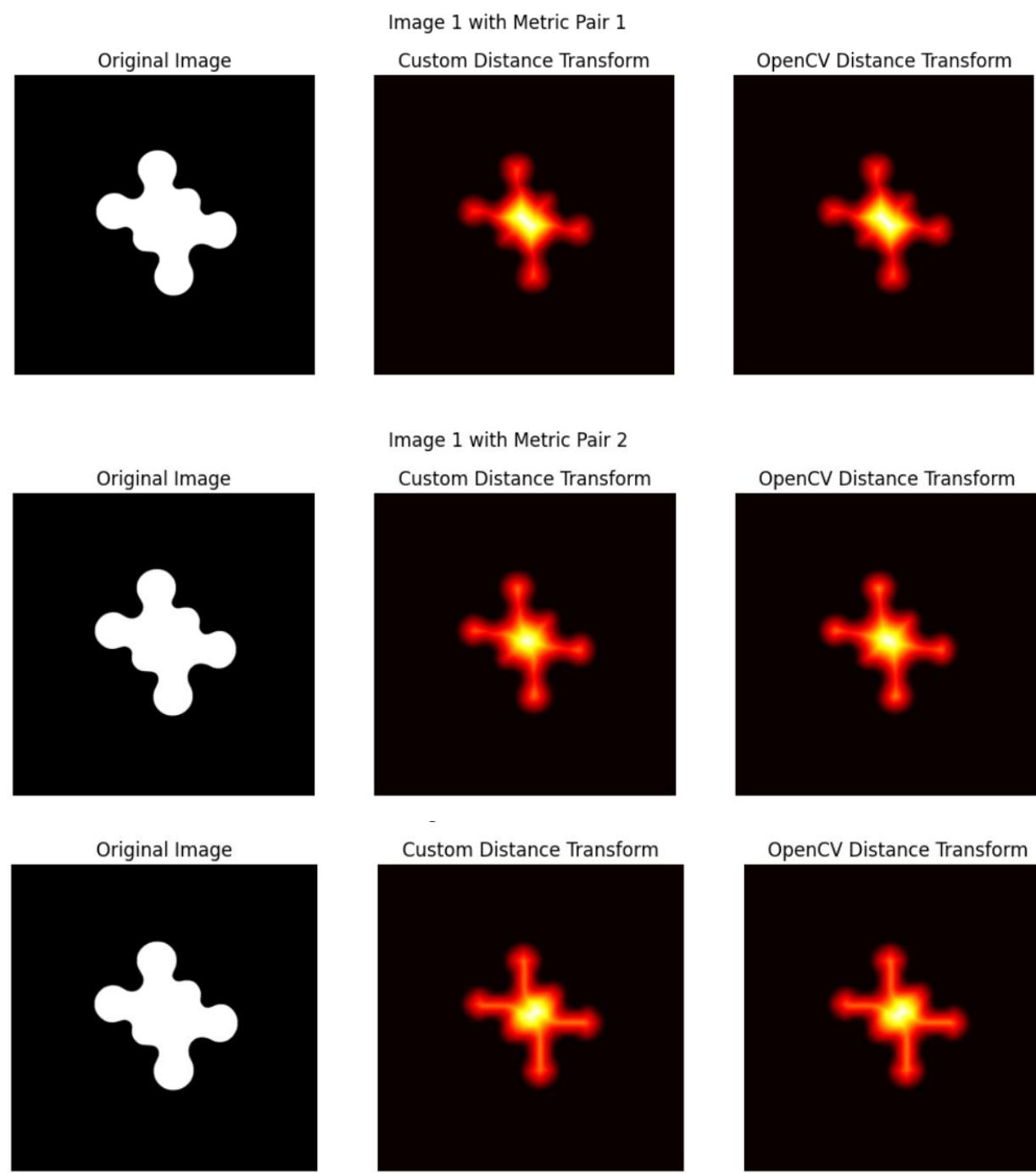
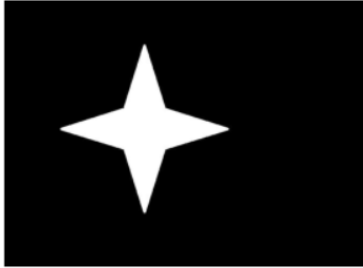
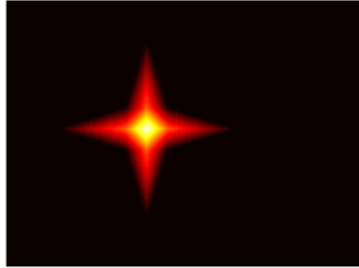


Image 2 with Metric Pair 1

Original Image



Custom Distance Transform



OpenCV Distance Transform

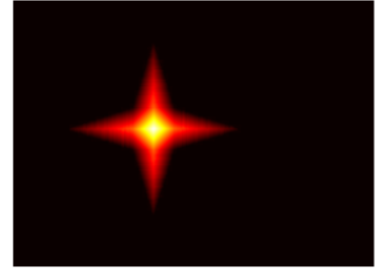
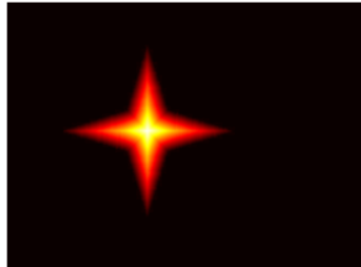


Image 2 with Metric Pair 2

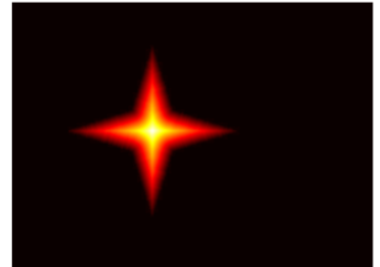
Original Image



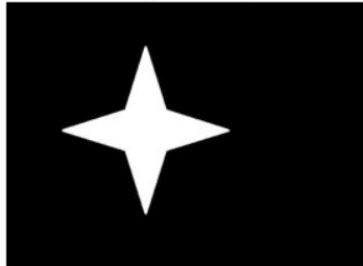
Custom Distance Transform



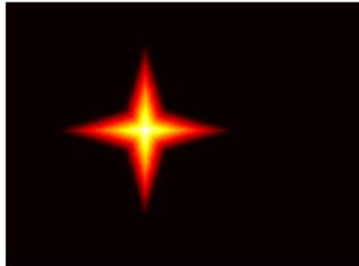
OpenCV Distance Transform



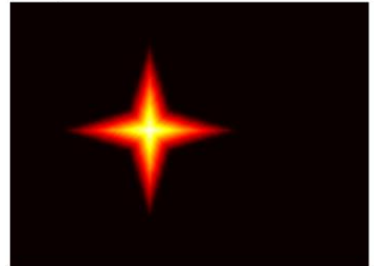
Original Image

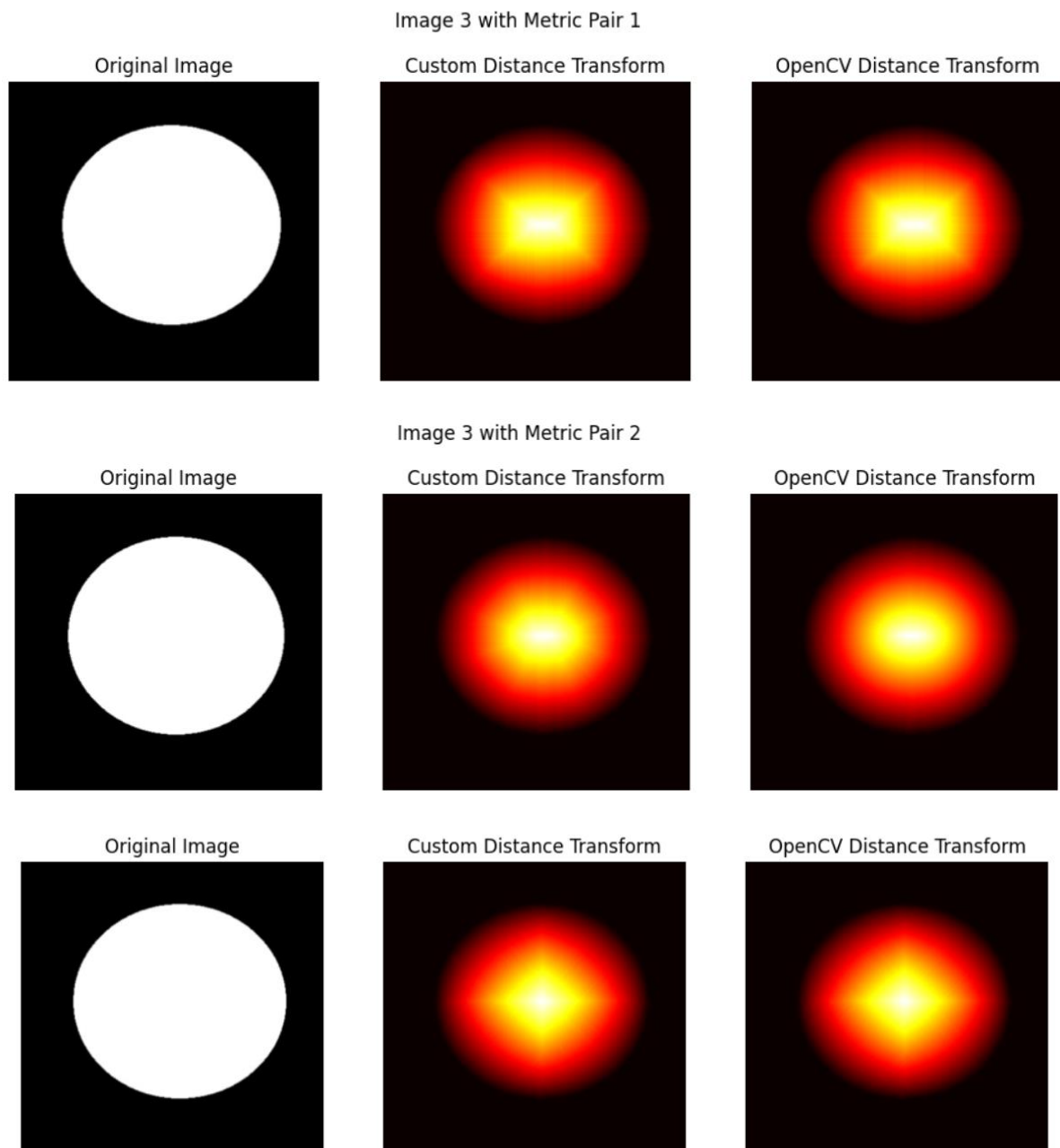


Custom Distance Transform



OpenCV Distance Transform



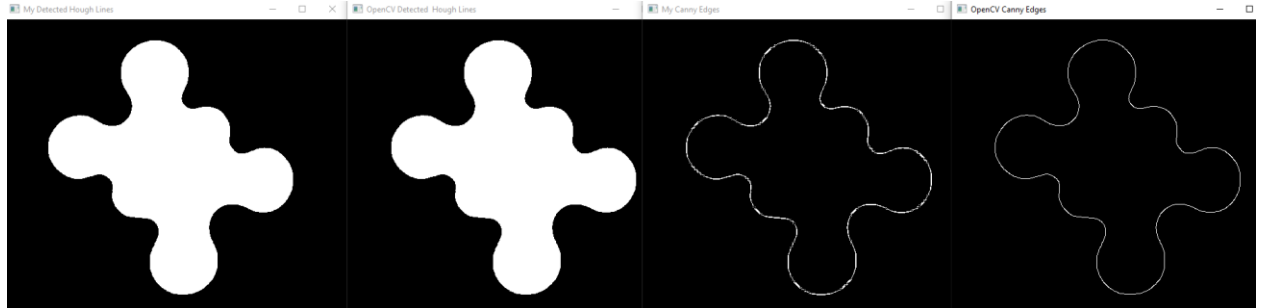


Порівняння з функціями OpenCV:

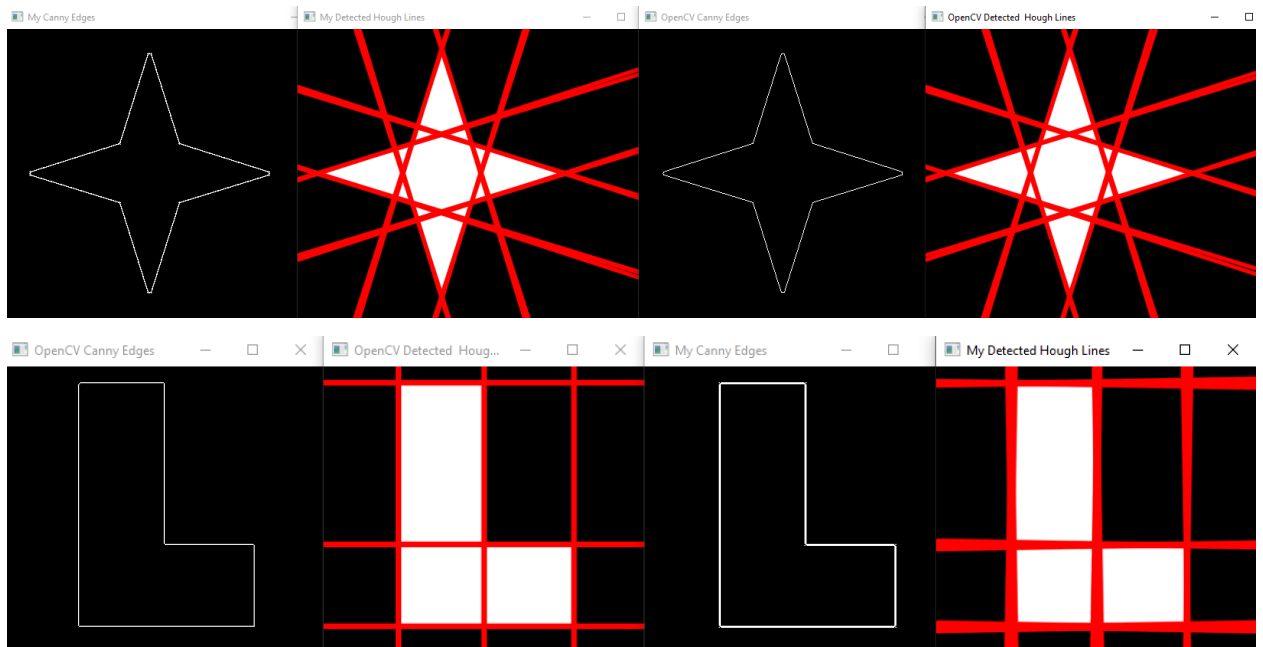
У порівнянні з функцією `cv2.distanceTransform`, що оптимізована для різних метрик та забезпечує швидке виконання на основі високопродуктивних алгоритмів, власний метод дистанційного перетворення має базову реалізацію і є більш обчислювально затратним. OpenCV використовує внутрішні оптимізації для коректної обробки навіть великих зображень, що робить його ефективнішим у практичному застосуванні. Проте власний метод допомагає краще зрозуміти принципи роботи дистанційного перетворення, зокрема вплив метрики на форму та значення відстаней.

Завдання 7

Знайти та візуалізувати лінії на зображенні за допомогою стандартного перетворення Хафа (обрати зображення із наявними чіткими рівними лініями).



Як і очікувалося ліній не виявлено.



Перетворення Хафа є потужним методом для виявлення геометричних фігур, зокрема прямих ліній, у зображеннях. Основна ідея цього підходу полягає в трансформації координатної системи зображення в параметричну, що дозволяє відзначати всі можливі прямі, що проходять через кожну точку зображення. амість того, щоб працювати у двовимірному просторі (координати x, y), ми переходимо до параметричного простору, де кожна лінія може бути описана через параметри ρ та θ .

У даному завданні було реалізовано методи виявлення країв та візуалізації ліній на зображеннях за допомогою алгоритму Канні та стандартного перетворення Хафа. Після виявлення країв було реалізовано стандартне перетворення Хафа для знаходження прямих ліній на зображенні. Кожен піксель, що є частиною краю, переводиться у параметричну форму лінії (ρ, θ) . Для кожного пікселя обчислюється кілька значень ρ для різних кутів θ , що дозволяє заповнити акумулятор, де зберігається кількість голосів для кожної пари (ρ, θ) . Після заповнення акумулятора, за допомогою заданого порога визначаються лінії, які є достатньо помітними.

Порівняння з функціями OpenCV:

Власна реалізація не оптимізована і може дати гірші результати в порівнянні з вбудованою функцією OpenCV. Але дозволяє змінювати алгоритм відповідно до специфічних вимог проекту, наприклад, налаштовувати порогові значення або розмір акумулятора. OpenCV надає потужну і оптимізовану реалізацію перетворення Хафа, яка широко використовується в комп'ютерному зорі. Вона реалізована з урахуванням різних нюансів обробки зображень і намагається зменшити вплив шуму. Вбудоване перетворення Хафа виявляло лінії з кращою точністю, завдяки застосуванню оптимізованих алгоритмів для обробки країв та голосування в акумуляторі.

Також треба зазначити що на результат впливало те, що використовувалася власна реалізація методу Канні для виявлення країв. Вона використовується для попередньої обробки зображення, може мати гірші результати в порівнянні з вбудованою версією. Це може бути пов'язано з недосконалими налаштуваннями параметрів, такими як порогові значення, що призводить до втрат інформації про краї. Неправильний вибір порогових значень або розмірів матриці акумулятора може призвести до поганих результатів. Власна реалізація може не забезпечувати достатньої чутливості до важливих ліній.

Висновок:

У цій роботі було розглянуто завдання створення та обробки зображень із застосуванням базових алгоритмів комп'ютерного зору та геометричного аналізу. Реалізовані методи включали завантаження зображень, обчислення їхніх характеристик, а також візуалізацію та збереження результатів. Основним завданням було побудувати застосунок, здатний ідентифікувати зв'язні ділянки на зображенні, обчислити площу та параметри форми цих ділянок, зокрема центроїд, головну вісь та ексцентриситет, а також візуалізувати знайдені характеристики.

Окрім того, було виконано трасування краю зв'язної ділянки, зокрема реалізовано алгоритм Восса для побудови та візуалізації контуру, обчислено його довжину, а також проведено аналіз кривизни у різних точках контуру. Під час цього етапу було знайдено та відмічено на зображенні точки з максимальною і мінімальною кривизною, що дозволило оцінити властивості контуру. Виконано дистанційне перетворення для заданої ділянки, що є корисним інструментом для подальших задач, таких як аналіз форми чи обробка зображень у медичних або географічних системах.

Для виявлення лінійних елементів зображення було застосовано стандартне перетворення Хафа, яке дозволило ефективно знайти й візуалізувати чіткі рівні лінії на зображенні, відповідні загальним геометричним елементам.

Результати показали ефективність алгоритмів у вирішенні поставлених задач і відповідність отриманих результатів очікуванням. Додатково було проведено порівняння реалізованих алгоритмів з їхніми реалізаціями в OpenCV, що допомогло визначити сильні та слабкі сторони кожного підходу, а також дало уявлення про можливі оптимізації.

Додаток А:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from google.colab.patches import cv2_imshow

paths = [f'img{i+1}.jpg' for i in range(3)]
imgs = [cv2.imread(path, cv2.IMREAD_GRAYSCALE) for path in paths]
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, len(imgs), figsize=(15, 5))

for ax, img in zip(axes, imgs):
    ax.imshow(img, cmap='gray')
    ax.axis('off')

plt.show()
def get_area(img):
    mask = img > 127
    return mask.sum(), mask.mean()

def print_areas_info(imgs):
    for i, img in enumerate(imgs):
        print(f'Img{i+1}')
        print('White area (in pixels): %s \nWhite area (in percents):'
%.5f%%\n\n' % get_area(img))
def moore_neighborhood_contour(mask):
    img = mask.copy()
    height, width = img.shape
    contour = []
    start_pixel = None

    start_pixel = np.argwhere(img == 255)
    if start_pixel.size == 0:
        return np.array(contour)
    start_pixel = tuple(start_pixel[0])
    contour.append(start_pixel)

    p = start_pixel
    current_direction = 0
    directions = np.array([
        (-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -
1)
    ], dtype=int)

    def get_next_pixel(p, start_direction):
        for i in range(8):
            dir_index = (start_direction + i) % 8
            new_yx = np.array(p) + directions[dir_index]
```

```

        if (0 <= new_yx[0] < height and 0 <= new_yx[1] < width and
img[tuple(new_yx)] == 255):
            return tuple(new_yx), dir_index
        return None, start_direction

    while True:
        c, current_direction = get_next_pixel(p, (current_direction + 6) %
8)

        if c == start_pixel:
            break
        if c:
            contour.append(c)
            img[c] = 128
            p = c

    return np.array(contour)

def calculate_contour_length(contour):
    return np.sum(np.sqrt((np.diff(contour, axis=0) ** 2).sum(axis=1)))

def opencv_contour(binary_image):
    contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    longest_contour = max(contours, key=cv2.contourArea)
    edge_pixels = np.array(list(map(lambda pt: tuple(pt[0]),
longest_contour)))
    edge_length = calculate_contour_length(edge_pixels)
    return edge_pixels, edge_length

def process_image(image):
    mask = np.where(image > 127, 255, 0).astype(np.uint8)

    custom_contour = moore_neighborhood_contour(mask)
    custom_contour_length = calculate_contour_length(custom_contour)

    custom_contour_image = cv2.cvtColor(mask.copy(), cv2.COLOR_GRAY2BGR)
    for point in custom_contour:
        cv2.circle(custom_contour_image, (point[1], point[0]), 1, (0, 255,
0), -1)

    opencv_contour_pixels, opencv_contour_length =
opencv_contour(mask.copy())

    opencv_contour_image_colored = cv2.cvtColor(mask.copy(),
cv2.COLOR_GRAY2BGR)
    for pixel in opencv_contour_pixels:
        cv2.circle(opencv_contour_image_colored, (pixel[0], pixel[1]), 1,
(0, 255, 0), -1)

    return custom_contour_image, custom_contour_length,
opencv_contour_image_colored, opencv_contour_length

```

```

results = [process_image(image) for image in imgs]

fig, axes = plt.subplots(3, 2, figsize=(12, 18))

for i, (custom_img, custom_length, opencv_img, opencv_length) in
enumerate(results):
    axes[i, 0].imshow(custom_img)
    axes[i, 0].set_title(f"Image {i+1} - Custom Contour Length:
{custom_length:.2f} pixels")
    axes[i, 0].axis('off')

    axes[i, 1].imshow(opencv_img)
    axes[i, 1].set_title(f"Image {i+1} - OpenCV Contour Length:
{opencv_length:.2f} pixels")
    axes[i, 1].axis('off')

plt.tight_layout()
plt.show()

def distance_transform_custom(img, kernel):
    dist_transform = np.where(img == 0, 0, np.inf)
    rows, cols = img.shape

    for i in range(rows):
        for j in range(cols):
            if dist_transform[i, j] != 0:
                for (di, dj), weight in kernel:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols:
                        dist_transform[i, j] = min(dist_transform[i, j],
dist_transform[ni, nj] + weight)

    for i in range(rows - 1, -1, -1):
        for j in range(cols - 1, -1, -1):
            if dist_transform[i, j] != 0:
                for (di, dj), weight in kernel:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols:
                        dist_transform[i, j] = min(dist_transform[i, j],
dist_transform[ni, nj] + weight)

    return dist_transform

manhattan_metric = [((-1, 0), 1), ((1, 0), 1), ((0, -1), 1), ((0, 1), 1)]

euclidean_metric = [
    ((-1, -1), np.sqrt(2)), ((-1, 0), 1), ((-1, 1), np.sqrt(2)),
    ((0, -1), 1), ((0, 1), 1),
    ((1, -1), np.sqrt(2)), ((1, 0), 1), ((1, 1), np.sqrt(2))
]

```

```

]

chebishev_metric = [
    ((-1, -1), 1), ((-1, 0), 1), ((-1, 1), 1),
    ((0, -1), 1), ((0, 1), 1),
    ((1, -1), 1), ((1, 0), 1), ((1, 1), 1)
]

metric_pairs = [
    (manhattan_metric, cv2.DIST_L1),
    (euclidean_metric, cv2.DIST_L2),
    (chebishev_metric, cv2.DIST_C)
]

for i, img in enumerate(imgs):
    for j, metric_pair in enumerate(metric_pairs):
        dist_transform = distance_transform_custom(img, metric_pair[0])

        opencv_dt = cv2.distanceTransform(img.astype(np.uint8),
metric_pair[1], 0)

        plt.figure(figsize=(12, 4))
        plt.suptitle(f"Image {i+1} with Metric Pair {j+1}")

        plt.subplot(1, 3, 1)
        plt.title("Original Image")
        plt.axis('off')
        plt.imshow(img, cmap='gray')

        plt.subplot(1, 3, 2)
        plt.title("Custom Distance Transform")
        plt.axis('off')
        plt.imshow(dist_transform, cmap='hot')

        plt.subplot(1, 3, 3)
        plt.title("OpenCV Distance Transform")
        plt.axis('off')
        plt.imshow(opencv_dt, cmap='hot')

    plt.show()

```

Додаток Б:

Func.cpp

```
#include "func.hpp"
#include <stdint>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <set>

namespace comp_vis {

    Point2f calc_centroid(const Mat& image) {
        float sumX = 0, sumY = 0;
        int count = 0;

        for (int y = 0; y < image.rows; ++y) {
            for (int x = 0; x < image.cols; ++x) {
                if (image.at<uchar>(y, x) == 255) {
                    sumX += x;
                    sumY += y;
                    count++;
                }
            }
        }

        return Point2f(sumX / count, sumY / count);
    }

    void visualize_centroid(Mat& image, const Point2f& centroid) {
        if (centroid.x >= 0 && centroid.y >= 0) {
            circle(image, Point(static_cast<int>(centroid.x),
                                static_cast<int>(centroid.y)), 4, Scalar(0, 0, 255), -1);
        }
    }

    void calc_major_axis(const Mat& image, const Point2f& centroid, Point2f&
axis_start, Point2f& axis_end, float& eccentricity) {
        float sumXX = 0, sumYY = 0, sumXY = 0;
        int count = 0;

        for (int y = 0; y < image.rows; ++y) {
            for (int x = 0; x < image.cols; ++x) {
                if (image.at<uchar>(y, x) == 255) {
                    sumXX += (x - centroid.x) * (x - centroid.x);
                    sumYY += (y - centroid.y) * (y - centroid.y);
                    sumXY += (x - centroid.x) * (y - centroid.y);
                    count++;
                }
            }
        }

        if (count == 0) {
            axis_start = Point2f(0, 0);
            axis_end = Point2f(0, 0);
            eccentricity = 0;
            return;
        }

        float covXX = sumXX / count;
        float covYY = sumYY / count;
        float covXY = sumXY / count;
```



```

    float trace = covXX + covYY;
    float determinant = covXX * covYY - covXY * covXY;

    float eigenValue1 = trace / 2 + sqrt(trace * trace / 4 -
determinant);
    float eigenValue2 = trace / 2 - sqrt(trace * trace / 4 -
determinant);

    float a = sqrt(eigenValue1);
    float b = sqrt(eigenValue2);

    eccentricity = sqrt(1 - (b * b) / (a * a));

    float theta = 0.5 * atan2(2 * covXY, covXX - covYY);
    float cosTheta = cos(theta);
    float sinTheta = sin(theta);

    float length = 150;
    axis_start = centroid + Point2f(-length * cosTheta, -length *
sinTheta);
    axis_end = centroid + Point2f(length * cosTheta, length * sinTheta);
}

void visualize_major_axis(Mat& image, const Point2f& axis_start, const
Point2f& axis_end) {
    line(image, Point(static_cast<int>(axis_start.x),
static_cast<int>(axis_start.y)),
        Point(static_cast<int>(axis_end.x),
static_cast<int>(axis_end.y)),
        Scalar(255, 0, 0), 2);
}

void cannyEdgeDetection(const Mat& src, Mat& dst, double lowerThreshold,
double upperThreshold) {
    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;
        }
    }
}

void houghTransform(const Mat& edges, vector<pair<float, float>>& lines,
int threshold) {
    int width = edges.cols;
    int height = edges.rows;
    int maxRho = sqrt(width * width + height * height);
    int numAngles = 180;
    vector<vector<int>> accumulator(2 * maxRho, vector<int>(numAngles,
0));

```

```

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (edges.at<uchar>(y, x) == 255) {
                    for (int theta = 0; theta < numAngles; theta++) {
                        float rad = theta * CV_PI / 180.0;
                        int rho = cvRound(x * cos(rad) + y * sin(rad)) +
maxRho;

                        if (rho >= 0 && rho < 2 * maxRho) {
                            accumulator[rho][theta]++;
                        }
                    }
                }
            }

            for (int rho = 0; rho < 2 * maxRho; rho++) {
                for (int theta = 0; theta < numAngles; theta++) {
                    if (accumulator[rho][theta] >= threshold) {
                        lines.emplace_back(rho - maxRho, theta * CV_PI / 180.0);
                    }
                }
            }
        }

void drawHoughLines(Mat& img, const vector<pair<float, float>>& lines) {
    for (const auto& line : lines) {
        float rho = line.first;
        float theta = line.second;
        Point pt1, pt2;
        double a = cos(theta), b = sin(theta);
        double x0 = a * rho, y0 = b * rho;
        pt1.x = cvRound(x0 + 1000 * (-b));
        pt1.y = cvRound(y0 + 1000 * (a));
        pt2.x = cvRound(x0 - 1000 * (-b));
        pt2.y = cvRound(y0 - 1000 * (a));
        cv::line(img, pt1, pt2, Scalar(0, 0, 255), 3, LINE_AA);
    }
}

double curvature(Point p1, Point p2, Point p3) {
    double dx1 = p2.x - p1.x;
    double dy1 = p2.y - p1.y;
    double dx2 = p3.x - p2.x;
    double dy2 = p3.y - p2.y;

    double num = abs(dx1 * dy2 - dy1 * dx2);
    double denom = pow(dx1 * dx1 + dy1 * dy1, 1.5);

    if (denom == 0) return 0;

    return num / denom;
}

void calculateCurvature(const vector<vector<Point>>& contours,
vector<Point>& maxCurvaturePoints, vector<Point>& minCurvaturePoints,
double& maxCurvature, double& minCurvature, double& avgCurvature,
int& pointCount) {

    const double epsilon = 1e-5;
    double sumCurvature = 0;
    maxCurvature = -std::numeric_limits<double>::infinity();
    minCurvature = std::numeric_limits<double>::infinity();
    pointCount = 0;

```

```

    for (const auto& contour : contours) {
        for (size_t i = 0; i < contour.size(); ++i) {
            Point p1 = contour[(i == 0) ? contour.size() - 1 : i - 1];
            Point p2 = contour[i];
            Point p3 = contour[(i == contour.size() - 1) ? 0 : i + 1];
            double curv = curvature(p1, p2, p3);
            sumCurvature += curv;
            pointCount++;

            if (curv > maxCurvature) {
                maxCurvature = curv;
                maxCurvaturePoints.clear();
                maxCurvaturePoints.push_back(contour[i]);
            }
            else if (curv == maxCurvature) {
                maxCurvaturePoints.push_back(contour[i]);
            }

            if (curv < minCurvature - epsilon) {
                minCurvature = curv;
                minCurvaturePoints.clear();
                minCurvaturePoints.push_back(contour[i]);
            }
            else if (abs(curv - minCurvature) < epsilon) {
                minCurvaturePoints.push_back(contour[i]);
            }
        }
    }

    if (pointCount > 0) {
        avgCurvature = sumCurvature / pointCount;
    }
    else {
        avgCurvature = 0;
    }
}

```

```

Mat markCurvaturePoints(const Mat& contourImage, const vector<Point>&
maxPoints, const vector<Point>& minPoints) {
    Mat markedImage = contourImage.clone();

```

```

    for (const auto& pt : minPoints) {
        circle(markedImage, pt, 2, Scalar(255, 0, 0), -1);
    }

```

```

    for (const auto& pt : maxPoints) {
        circle(markedImage, pt, 5, Scalar(0, 0, 255), -1);
    }

```

```

    return markedImage;
}

```

```

void outputMaxCurvaturePoints(const vector<Point>& maxCurvaturePoints) {
    cout << "Max Curvature Points:" << endl;
    for (const auto& pt : maxCurvaturePoints) {
        cout << "Point: (" << pt.x << ", " << pt.y << ")" << endl;
    }
}

```

```
}
```

Func.hpp

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

using namespace cv;
using namespace std;

namespace comp_vis {

    Point2f calc_centroid(const Mat& image);
    void visualize_centroid(Mat& image, const Point2f& centroid);
    void calc_major_axis(const Mat& image, const Point2f& centroid, Point2f&
axis_start, Point2f& axis_end, float& eccentricity);
    void visualize_major_axis(Mat& image, const Point2f& axis_start, const
Point2f& axis_end);
    void cannyEdgeDetection(const Mat& src, Mat& dst, double lowerThreshold = 50,
double upperThreshold = 150);
    void houghTransform(const Mat& edges, vector<pair<float, float>>& lines, int
threshold);
    void drawHoughLines(Mat& img, const vector<pair<float, float>>& lines);
    double curvature(Point p1, Point p2, Point p3);
    void calculateCurvature(const vector<vector<Point>>& contours, vector<Point>&
maxCurvaturePoints, vector<Point>& minCurvaturePoints,
double& maxCurvature, double& minCurvature, double& avgCurvature, int&
pointCount);
    Mat markCurvaturePoints(const Mat& contourImage, const vector<Point>&
maxPoints, const vector<Point>& minPoints);
    void outputMaxCurvaturePoints(const vector<Point>& maxCurvaturePoints);

}
```

Task.cpp

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

using namespace cv;
using namespace std;

void task3(int subtask)
{
    Mat input =
imread("D:\\Study\\4_course_1_sem\\CV\\lab_2\\images\\input\\4.jpg",
IMREAD_GRAYSCALE);
    if (input.empty()) {
        cout << "Could not read the image" << endl;
        return;
    }
    Mat binary;
    threshold(input, binary, 128, 255, THRESH_BINARY);

    int width = binary.cols;
```

```

int height = binary.rows;

/*  cout << "Pixel values of the input binary image:" << endl;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            cout << static_cast<int>(binary.at<uchar>(y, x)) << " ";
        }
        cout << endl;
    }*/

Mat output;
cvtColor(binary, output, COLOR_GRAY2BGR);

imshow("Task 3 Original image", binary);
moveWindow("Task 3 Original image", 0, 0);

switch (subtask) {
case 1: {
    Point2f centroid = comp_vis::calc_centroid(binary);

    comp_vis::visualize_centroid(output, centroid);

    imshow("MY Centroid Visualization", output);
    moveWindow("MY Centroid Visualization", width, 0);

    Point2f axis_start, axis_end;
    float my_eccentricity;
    comp_vis::calc_major_axis(binary, centroid, axis_start, axis_end,
my_eccentricity);
    comp_vis::visualize_major_axis(output, axis_start, axis_end);
    imshow("MY Major Axis Visualization", output);
    moveWindow("MY Major Axis Visualization", width, height);

    // TASK 3 OPENCV
    Mat centr_cv = binary.clone();
   .cvtColor(binary, centr_cv, COLOR_GRAY2BGR);
    Moments m = moments(binary, true);

    if (m.m00 == 0) {
        cout << "No white area found!" << endl;
        return;
    }

    Point p(m.m10 / m.m00, m.m01 / m.m00);
    circle(centr_cv, p, 4, Scalar(0, 0, 255), -1);
    imshow("OpenCV Centroid Visualization", centr_cv);
    moveWindow("OpenCV Centroid Visualization", 2 * width, 0);

    Mat covMatrix = (Mat_<doubledoubledoublefloat halfLength = 150.0;
    Point2f startPoint = centroid - halfLength * mainAxis;
    Point2f endPoint = centroid + halfLength * mainAxis;

    line(centr_cv, startPoint, endPoint, Scalar(255, 0, 0), 2);

    imshow("OpenCV Main Axis Visualization", centr_cv);

```

```

        moveWindow("OpenCV Main Axis Visualization", 2 * width, height);

        cout << "My Centroid: (" << static_cast<int>(centroid.x) << ", " <<
static_cast<int>(centroid.y) << ")" << endl;
        cout << "OpenCV Centroid: (" << p.x << ", " << p.y << ")" << endl;
        cout << "My major axis start: (" << axis_start.x << ", " <<
axis_start.y << "), end: (" << axis_end.x << ", " << axis_end.y << ")" <<
endl;
        cout << "OpenCV major axis start: (" << startPoint.x << ", " <<
startPoint.y << "), end: (" << endPoint.x << ", " << endPoint.y << ")" <<
endl;

        float a = sqrt(eigenValues.at<double>(0));
        float b = sqrt(eigenValues.at<double>(1));
        float eccentricity_cv = sqrt(1 - (b * b) / (a * a));
        cout << "OpenCV eccentricity: " << eccentricity_cv << endl;
        cout << "My eccentricity: " << my_eccentricity << endl;

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

```

```

void task5(int subtask) {

    Mat input =
    imread("D:\\Study\\4_curse_1_sem\\CV\\lab_2\\images\\input\\11.jpg",
    IMREAD_GRAYSCALE);
    if (input.empty()) {
        cout << "Could not read the image" << endl;
        return;
    }

    Mat binary;
    threshold(input, binary, 128, 255, THRESH_BINARY);

    imshow("Task 5 Original image", binary);
    moveWindow("Task 5 Original image", 0, 0);

    switch (subtask) {
    case 1: {

        //Task 5
        vector<vector<Point>> contours;

        findContours(binary, contours, RETR_EXTERNAL, CHAIN_APPROX_NONE);

        Mat contourImage = Mat::zeros(binary.size(), CV_8UC3);
        drawContours(contourImage, contours, -1, Scalar(0, 255, 0), 1);

        imshow("Contour Image", contourImage);

        vector<Point> maxCurvaturePoints, minCurvaturePoints;
        double maxCurvature = 0, minCurvature = DBL_MAX;
        double avgCurvature = 0;
        int pointCount = 0;
    }
    }
}

```

```

        comp_vis::calculateCurvature(contours, maxCurvaturePoints,
minCurvaturePoints, maxCurvature, minCurvature, avgCurvature, pointCount);

        cout << "Maximum curvature: " << maxCurvature << endl;
        cout << "Minimum curvature: " << minCurvature << endl;
        cout << "Average curvature: " << avgCurvature << endl;

        comp_vis::outputMaxCurvaturePoints(maxCurvaturePoints);

        Mat markedImage = comp_vis::markCurvaturePoints(contourImage,
maxCurvaturePoints, minCurvaturePoints);
        imshow("Marked Image with Curvature Points", markedImage);

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

void task7(int subtask) {

    Mat input =
imread("D:\\Study\\4_course_1_sem\\CV\\lab_2\\images\\input\\6.jpg",
IMREAD_GRAYSCALE);
    if (input.empty()) {
        cout << "Could not read the image" << endl;
        return;
    }

    Mat binary;
    threshold(input, binary, 128, 255, THRESH_BINARY);

    imshow("Task 7 Original image", binary);
    moveWindow("Task 7 Original image", 0, 0);

    switch (subtask) {
    case 1: {

        Mat my_edges;
        comp_vis::cannyEdgeDetection(binary, my_edges);
        imshow("My Canny Edges", my_edges);

        vector<pair<float, float>> my_lines;
        comp_vis::houghTransform(my_edges, my_lines, 70);
        Mat my_imgWithLines;
        cvtColor(binary, my_imgWithLines, COLOR_GRAY2BGR);
        comp_vis::drawHoughLines(my_imgWithLines, my_lines);
        imshow("My Detected Hough Lines", my_imgWithLines);

        // Task 7 OpenCV
        Mat edges;
        Canny(binary, edges, 50, 150);
        imshow("OpenCV Canny Edges", edges);

        vector<Vec2f> lines;

```



```

HoughLines(edges, lines, 1, CV_PI / 180, 60);

Mat imgWithLines;
cvtColor(binary, imgWithLines, COLOR_GRAY2BGR);

for (size_t i = 0; i < lines.size(); i++) {
    float rho = lines[i][0], theta = lines[i][1];
    Point pt1, pt2;
    double a = cos(theta), b = sin(theta);
    double x0 = a * rho, y0 = b * rho;
    pt1.x = cvRound(x0 + 1000 * (-b));
    pt1.y = cvRound(y0 + 1000 * (a));
    pt2.x = cvRound(x0 - 1000 * (-b));
    pt2.y = cvRound(y0 - 1000 * (a));
    line(imgWithLines, pt1, pt2, Scalar(0, 0, 255), 3, LINE_AA);
}

imshow("OpenCV Detected Hough Lines", imgWithLines);
moveWindow("OpenCV Detected Hough Lines", 0, 0);

cout << "Press any key to return to menu..." << endl;
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 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 << "3. Task 3: Find and output the centroid, major axis, and
eccentricity" << endl;
        cout << "5. Task 5: Estimate the curvature of the contour at different
points" << endl;
        cout << "7. Task 7: Find and visualize lines in an image using a
standard Hough transform " << endl;
        cout << "0. Exit" << endl;

        cin >> taskNumber;

        switch (taskNumber) {

        case 3: {

                do {
                        cout << "1. Display centroid, major axis, and
eccentricity" << endl;
                        cout << "0. Go back to main menu" << endl;

                        cin >> subtask;

                        if (subtask != 0) {
                                task3(subtask);
                        }

                } while (subtask != 0);
                break;

        }

        case 5: {

                do {
                        cout << "1. Display the maximum, minimum the average value
of curvature(its absolute value)." << endl;
                        cout << "0. Go back to main menu" << endl;

                        cin >> subtask;

                        if (subtask != 0) {
                                task5(subtask);
                        }

                } while (subtask != 0);
                break;

        }

        case 7: {

                do {
                        cout << "1. Display lines in an image using a standard
Hough transform" << 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;
}
```