## ****Лабораторна робота****

## ****на тему “Трикутник найбільшої площі вписаний в опуклу Оболонку”****

**Виконував роботу студент Сіхневич Святослав Геннадійович з 3 курсу група ТК-32**

**Анотація**

Цей звіт описує метод побудови опуклої оболонки для заданої множини з N точок та знаходження трикутника з найбільшою площею, вписаного в цю оболонку. Запропонований алгоритм використовує геометричні властивості опуклих багатокутників для ефективного пошуку такого трикутника. Результати роботи можуть бути застосовані в комп'ютерній графіці, розпізнаванні образів та оптимізаційних задачах.

This report describes a method for constructing a convex hull for a given set of N points and finding the triangle with the largest area inscribed within this hull. The proposed algorithm utilizes geometric properties of convex polygons for efficient search of such a triangle. The results can be applied in computer graphics, pattern recognition, and optimization problems.

**Вступ**

**Актуальність теми роботи**

Задача знаходження трикутника найбільшої площі, вписаного в опуклу оболонку множини точок, є класичною задачею обчислювальної геометрії. Вона має значне практичне застосування в різноманітних галузях, таких як комп'ютерна графіка (для визначення оптимальних меж об'єктів), розпізнавання образів (для виділення ключових фігур), машинне навчання (для кластеризації даних), а також в задачах оптимізації та проектування. Ефективне розв'язання цієї задачі дозволяє оптимізувати ресурси, покращити якість обробки даних та забезпечити швидкість виконання операцій у великих системах.

**Аналіз існуючих підходів і методів розв’язання та практичного застосування результатів їх розв’язання**

Традиційні підходи до розв'язання цієї задачі часто базуються на переборі всіх можливих комбінацій трьох точок з вихідної множини, що призводить до високої обчислювальної складності O(N^3), де N – кількість точок. Проте, оскільки трикутник найбільшої площі завжди має свої вершини на опуклій оболонці множини точок (Теорема 3.4), ефективніші методи зосереджуються на точках опуклої оболонки, кількість яких k≤N.

Існують алгоритми, які використовують обертовий каліпер (rotating calipers) для знаходження трикутника найбільшої площі в опуклому багатокутнику. Цей метод дозволяє знизити складність до O(k^2), де k – кількість вершин опуклої оболонки. Подальші оптимізації, такі як алгоритм бінарного пошуку вершин максимального трикутника, дозволяють досягти складності O(klogk) або навіть O(k) для деяких випадків.

**На практиці результати цих розв'язань використовуються в:**

* **Комп'ютерній графіці:** для ефективного рендерингу складних сцен, де об'єкти можуть бути апроксимовані їх опуклими оболонками.
* **Геоінформаційних системах (ГІС):** для аналізу територій, визначення оптимальних ділянок.
* **Робототехніці:** для планування шляху та уникнення зіткнень.
* **Медичній візуалізації:** для аналізу форми органів та пухлин.

**Аргументація новизни і коротка ідея підходу**

Підхід ґрунтується на використанні властивості, що трикутник найбільшої площі, вписаний у опуклу оболонку, має всі свої вершини на цій оболонці. Новизна полягає в оптимізації пошуку третьої вершини для фіксованої основи трикутника. Замість повного перебору, ми використовуємо той факт, що для фіксованої основи (двох вершин) третя вершина, що максимізує площу, буде найдальшою від прямої, що містить цю основу. Цей пошук оптимізується за рахунок того, що при переборі однієї з вершин основи, інша вершина основи та вершина, що максимізує площу, рухаються в одному напрямку вздовж опуклої оболонки. Це дозволяє уникнути повного перебору, зменшуючи складність алгоритму.

**Мета роботи.** Основною метою даної роботи є розробка та програмна реалізація ефективного алгоритму, що дозволяє для заданої множини з N точок на площині:

1. Побудувати опуклу оболонку.
2. Знайти трикутник найбільшої площі, всі вершини якого належать побудованій опуклій оболонці.
3. Візуалізувати результати роботи алгоритму для наочного підтвердження його коректності.

**Основна частина**

**Постановка задачі**

У роботі розглядається один із підходів до розв’язання задачі знаходження трикутника максимальної площі, вписаного в опуклу оболонку множини точок у двовимірному евклідовому просторі. Задачі такого типу належать до класу екстремальних задач обчислювальної геометрії та мають велике теоретичне і прикладне значення.

Розв’язання цієї задачі дозволяє визначати геометричні конфігурації з максимально можливою площею, що, зокрема, застосовується у комп’ютерній графіці, аналізі просторових даних, геоінформаційних системах, обробці зображень, моделюванні та оптимізації. У практичних задачах така постановка виникає, наприклад, під час побудови триангуляцій, оцінки щільності покриття, оптимального використання площі чи в задачах кластеризації.

У межах цієї роботи пропонується ефективний алгоритм знаходження такого трикутника з використанням властивостей опуклої оболонки та методу обертальних лінійок, що дозволяє знизити обчислювальну складність та забезпечити високу продуктивність при обробці великих обсягів вхідних даних.

Дано множину з N точок на площині S={p1,p2,...,pN}. Необхідно побудувати опуклу оболонку CH(S) цієї множини та знайти трикутник з найбільшою площею, вершини якого належать CH(S).

**Основні означення, теореми, леми, твердження, гіпотези:**

* **Опукла оболонка (Convex Hull).** Опуклою оболонкою множини точок S називається найменша опукла множина, яка містить усі точки з S. В контексті цього завдання, опукла оболонка є опуклим багатокутником, вершини якого є підмножиною вихідних точок S.
* **Площа трикутника.** Площа трикутника з вершинами (x1, y1), (x2, y2) та (x3, y3) може бути обчислена за формулою: A=0.5\*∣x1(y2−y3)+x2(y3−y1)+x3(y1−y2)∣ або за допомогою векторного добутку: A=0.5\*∣AB×AC∣, де AB та AC – вектори, утворені сторонами трикутника.
* **Теорема 3.4 (з джерела):** Вершини трикутника максимальної площі, вписаного в опуклу оболонку, збігаються з вершинами опуклої оболонки.
  + **Доведення (коротко).** Припустимо, що одна з вершин трикутника максимальної площі не належить опуклій оболонці. Тоді можна "висунути" цю вершину до межі опуклої оболонки, збільшивши висоту трикутника відносно протилежної сторони, а отже, і його площу, що суперечить припущенню про максимальність.
* **Твердження:** Для фіксованої основи (двох вершин) трикутника, що належить опуклій оболонці, третя вершина, яка максимізує площу, є найдальшою від прямої, що містить цю основу.

**Аргументація запропонованого підходу чи методу**

Згідно з **Теоремою 3.4**, для знаходження трикутника найбільшої площі достатньо розглядати лише вершини, що належать опуклій оболонці.

Запропонований метод використовує цю властивість. Спочатку будується опукла оболонка заданої множини точок. Потім, для кожної пари вершин опуклої оболонки, що утворюють основу трикутника, шукається третя вершина, яка максимізує площу. Цей пошук оптимізується завдяки тому, що при обході вершин опуклої оболонки в циклічному порядку, відстань від прямої, утвореної двома фіксованими вершинами (основою), до третьої вершини буде спочатку збільшуватися, а потім зменшуватися. Це дозволяє використовувати "двох покажчиків" або "обертового каліпера" підхід: для фіксованих вершин i та j, третя вершина k шукається, переміщуючись по опуклій оболонці в одному напрямку, поки площа трикутника не почне зменшуватися. Це дозволяє уникнути повного перебору всіх можливих трійок вершин, значно прискорюючи процес.

**Основні етапи алгоритму:**

1. **Генерація точок.** Створити множину з N випадкових точок на площині.
2. **Побудова опуклої оболонки.** Використовуючи алгоритм ConvexHull (наприклад, з бібліотеки **SciPy**), побудувати опуклу оболонку для згенерованих точок. В результаті отримуємо впорядкований список вершин опуклої оболонки.
3. **Знаходження трикутника найбільшої площі:**

А) Ініціалізувати max\_area = 0 та best\_triangle = None.

Б) Ітерувати по всіх можливих парах вершин (i, j) опуклої оболонки, які будуть формувати основу трикутника.

В) Для кожної пари (i, j): ініціалізувати покажчик k на наступну вершину після j (по модулю n, де n – кількість вершин опуклої оболонки).

* + - Використовуючи "обертовий каліпер" підхід:

1. Обчислити площу трикутника (i, j, k).
2. Обчислити площу трикутника (i, j, next\_k), де next\_k – наступна вершина після k.
3. Якщо area(i, j, next\_k) > area(i, j, k), то перемістити k до next\_k і повторити.

• Інакше, зупинити пошук для даної основи (i, j). Вершина k є тією, що максимізує площу для цієї основи.

1. Порівняти знайдену площу з max\_area. Якщо вона більша, оновити max\_area та best\_triangle.
2. **Візуалізація результатів.** Побудувати графік, що відображає:
3. Випадково згенеровані точки.
4. Опуклу оболонку (з'єднані вершини).
5. Трикутник найбільшої площі, вписаний в опуклу оболонку, виділений кольором.
6. Додатково: відображення 10 внутрішніх точок та обертання для кращої візуалізації.

**Обґрунтування оцінки складності алгоритму**

* **Побудова опуклої оболонки.** Використання scipy.spatial.ConvexHull базується на алгоритмі Quickhull, який має середню складність O(NlogN) та гіршу складність O(N^2).
* **Знаходження трикутника найбільшої площі**
  + **Теорема 3.5** стверджує, що вписування трикутника в n-вершинну опуклу оболонку запропонованим методом може бути виконано за час O(n^3). Однак, при використанні оптимізованого алгоритму, як у цьому випадку, де для фіксованої основи третя вершина знаходиться за допомогою "обертового каліпера", складність покращується.
  + Для кожної з k вершин опуклої оболонки як першої вершини трикутника (i), і для кожної з k−1 вершин як другої (j), ми обходимо вершини опуклої оболонки (покажчик k) лише один раз для кожної фіксованої вершини i за повний цикл j. Тобто, для фіксованого i, сумарний рух покажчика k за всі j становить O(k).

Отже, загальна складність алгоритму знаходження трикутника найбільшої площі становить O(k^2), де k – кількість вершин опуклої оболонки.

У найгіршому випадку k може бути рівним N, тому загальна складність алгоритму становить O(N^2).

* + **Теорема 3.6** стверджує, що вписування трикутника в n-вершинну опуклу оболонку запропонованим методом (з бінарним пошуком) можна виконати за час O(nlogn). Цей алгоритм знаходить трикутник з найбільшою площею, використовуючи властивості, що для фіксованої точки і ребра, третя вершина, що максимізує площу, є найдальшою від цього ребра.

**Практична частина**

**а) Особливості програмної реалізації:**

Програмна реалізація виконана на мові Python з використанням бібліотек numpy для числових обчислень, matplotlib.pyplot для візуалізації та scipy.spatial.ConvexHull для побудови опуклої оболонки.

* Використовується генерація випадкових точок для демонстрації роботи алгоритму.
* Функція ConvexHull з scipy.spatial ефективно будує опуклу оболонку.
* Алгоритм знаходження трикутника максимальної площі (max\_area\_triangle) реалізований з використанням оптимізованого підходу, що дозволяє знайти третю вершину за допомогою "обертового каліпера" для фіксованої основи.
* Передбачена візуалізація опуклої оболонки, знайденого трикутника та деяких внутрішніх точок.
* Додано функцію rotate\_points для обертання фігур, що дозволяє краще продемонструвати їх форму та орієнтацію.

**б) Опис основних функцій програмної реалізації:**

* np.random.seed(42): Встановлює початкове значення для генератора випадкових чисел, що забезпечує відтворюваність результатів.
* all\_points = 3 \* (np.random.rand(10000, 2) - 0.5): Генерує 10000 випадкових точок в діапазоні [-1.5, 1.5] для кожної координати.
* hull = ConvexHull(all\_points): Обчислює опуклу оболонку для множини all\_points. Об'єкт hull містить інформацію про вершини та ребра оболонки.
* hull\_points = all\_points[hull.vertices]: Витягує координати вершин опуклої оболонки.
* max\_area\_triangle(points):
  + Приймає на вхід масив points (вершини опуклої оболонки).
  + Ітерує по всіх можливих парах точок (i, j) як основи.
  + Для кожної основи (i, j) використовує внутрішній цикл while True для пошуку оптимальної третьої точки k за допомогою "обертового каліпера".
  + Обчислення площі виконується за допомогою векторного добутку (np.cross).
  + Повертає індекси вершин трикутника з максимальною площею та саму максимальну площу.
* rotate\_points(points, angle\_deg):
  + Приймає масив точок та кут обертання в градусах.
  + Перетворює кут в радіани.
  + Створює матрицю обертання R.
  + Виконує матричне множення points @ R.T для обертання всіх точок.
  + Повертає обернені точки.
* plt.plot(...), plt.fill(...), plt.text(...): Функції з бібліотеки matplotlib для візуалізації опуклої оболонки, трикутника, внутрішніх точок та текстових написів.

**в) Характеризація вводу-виводу даних:**

* **Ввід:**
  + Набір з N двовимірних точок (масив all\_points). У даній реалізації генерується 10000 випадкових точок.
  + Кут обертання для візуалізації.
* **Вивід:**
  + Графічне зображення (візуалізація), що включає:

1. Опуклу оболонку (синя лінія, червоні точки).
2. Трикутник найбільшої площі (зелена лінія, лаймове заповнення).
3. 10 внутрішніх точок (червоні точки).
4. Підписи для вершин опуклої оболонки та внутрішніх точок.
5. Текстові написи, що описують зображення.
   * Текстовий вивід у консоль: максимальна площа знайденого трикутника.

**г) Можливості, програмне та технічне забезпечення програмної реалізації:**

* **Можливості:**
  + Побудова опуклої оболонки для довільної множини двовимірних точок.
  + Ефективний пошук трикутника найбільшої площі, вписаного в опуклу оболонку.
  + Візуалізація результатів, що дозволяє наочно представити роботу алгоритму.
  + Масштабованість для великих множин точок (до 10000 і більше).
  + Наявність можливості як ручного, так і автоматичного введення точок. Автоматичний спосіб дозволяє ввести до 10000 точок.
* **Програмне забезпечення:**
  + Операційна система: Будь-яка, що підтримує Python (Windows, macOS, Linux).
  + Мова програмування: Python 3.13 у Visual Studio Code (VS Code).
  + Бібліотеки: numpy, matplotlib, scipy.
* **Технічне забезпечення:**
  + Процесор: Будь-який сучасний процесор.
  + Оперативна пам'ять: Залежить від кількості точок. Для 10000 точок достатньо 4 ГБ.
  + Відеокарта: Не критична, оскільки візуалізація не вимагає значних ресурсів.

**д) Лістинг основних модулів програми:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.spatial import ConvexHull

# Генерація випадкових точок

np.random.seed(42)

all\_points = 3 \* (np.random.rand(10000, 2) - 0.5)

# Побудова опуклої оболонки

hull = ConvexHull(all\_points)

hull\_points = all\_points[hull.vertices]

# Знаходження трикутника найбільшої площі серед точок оболонки

def max\_area\_triangle(points):

    n = len(points)

    max\_area = 0

    best\_triangle = None

    for i in range(n):

        # Оптимізація: j починається з i + 1

        # k також рухається монотонно

        k\_ptr = (i + 2) % n

        for j in range(i + 1, n):

            # Переміщуємо k\_ptr, доки площа збільшується

            while True:

                next\_k = (k\_ptr + 1) % n

                area1 = 0.5 \* abs(np.cross(points[j] - points[i], points[k\_ptr] - points[i]))

                area2 = 0.5 \* abs(np.cross(points[j] - points[i], points[next\_k] - points[i]))

                if area2 > area1:

                    k\_ptr = next\_k

                else:

                    break

            area = 0.5 \* abs(np.cross(points[j] - points[i], points[k\_ptr] - points[i]))

            if area > max\_area:

                max\_area = area

                best\_triangle = (i, j, k\_ptr)

    return best\_triangle, max\_area

triangle\_indices, area = max\_area\_triangle(hull\_points)

triangle = hull\_points[list(triangle\_indices)]

# Вибір 10 внутрішніх точок (для візуалізації)

hull\_idx = set(hull.vertices)

interior = np.array([p for i, p in enumerate(all\_points) if i not in hull\_idx])[:10]

# Обертання точок для візуалізації

def rotate\_points(points, angle\_deg):

    angle = np.radians(angle\_deg)

    R = np.array([[np.cos(angle), -np.sin(angle)],

                  [np.sin(angle),  np.cos(angle)]])

    return points @ R.T

angle = -20

hull\_rot = rotate\_points(hull\_points, angle)

triangle\_rot = rotate\_points(triangle, angle)

interior\_rot = rotate\_points(interior, angle)

# Візуалізація

plt.figure(figsize=(6, 6))

hull\_closed = np.append(hull\_rot, [hull\_rot[0]], axis=0)

plt.plot(hull\_closed[:, 0], hull\_closed[:, 1], 'b-', linewidth=0.5)

plt.plot(hull\_rot[:, 0], hull\_rot[:, 1], 'r.', markersize=4)

# Підписи вершин оболонки

for i, (x, y) in enumerate(hull\_rot):

    plt.text(x + 0.05, y + 0.05, str(i + 1), fontsize=9)

# Внутрішні точки

plt.plot(interior\_rot[:, 0], interior\_rot[:, 1], 'r.', markersize=3)

for i, (x, y) in enumerate(interior\_rot):

    plt.text(x + 0.03, y - 0.03, f"P{i+1}", fontsize=9)

# Найбільший трикутник

plt.plot(\*np.append(triangle\_rot, [triangle\_rot[0]], axis=0).T, 'g-', linewidth=0.6)

plt.fill(triangle\_rot[:, 0], triangle\_rot[:, 1], 'lime', alpha=0.3)

# Текст

plt.text(-3.0, 2.1, "Трикутник", fontsize=13, family='serif')

plt.text(-3.0, 1.8, "найбільшої  площі", fontsize=13, family='serif')

plt.text(-3.0, 1.5, "вписаний  в  опуклу", fontsize=13, family='serif')

plt.text(-3.0, 1.2, "оболонку", fontsize=13, family='serif')

plt.axis("equal")

plt.axis("off")

plt.tight\_layout()

plt.show()

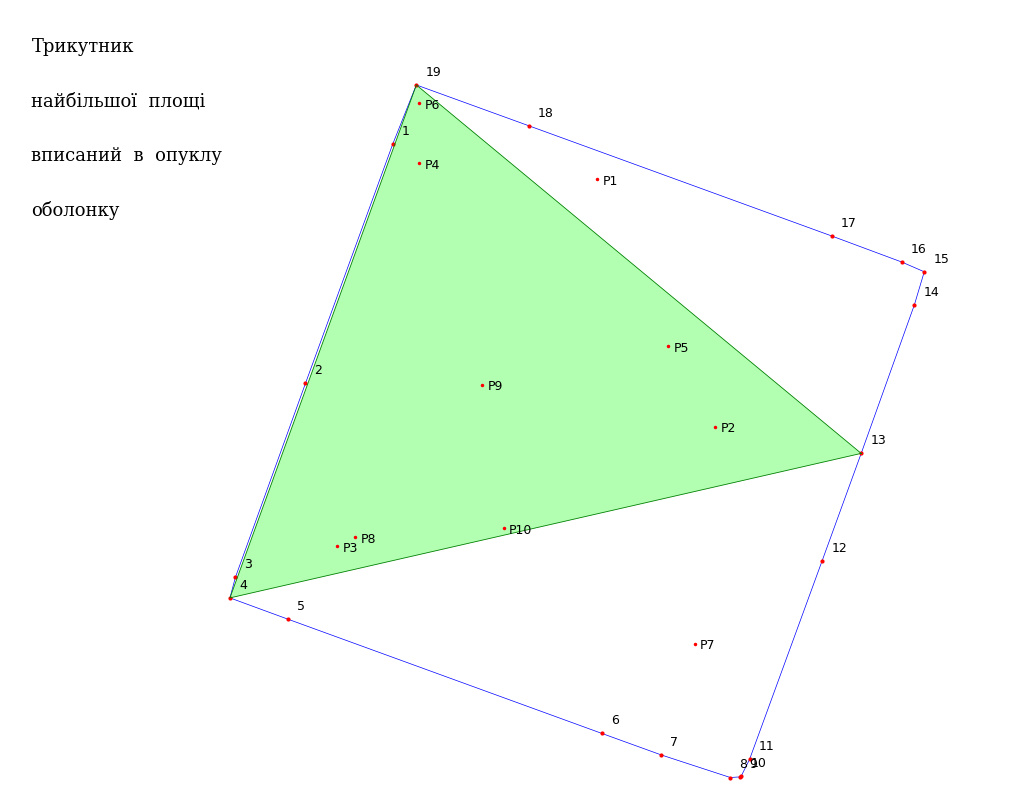
# Друк площі

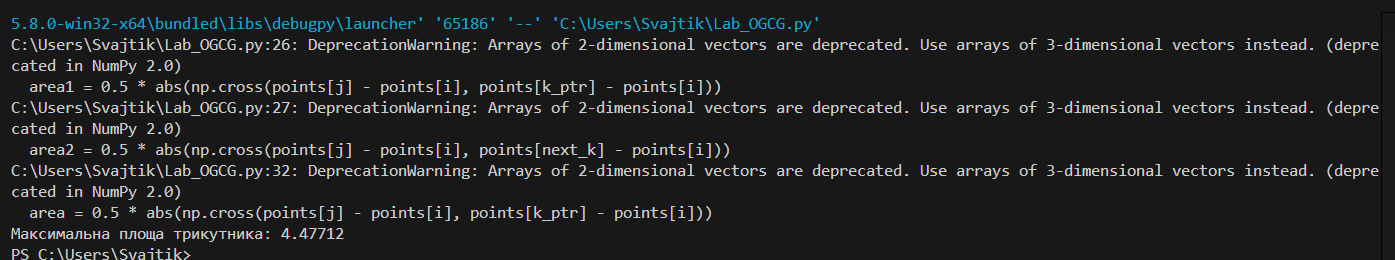
print(f"Максимальна площа трикутника: {area:.5f}")

За потреби залишив репозиторій з кодом програми:

https://github.com/Svyatoslavk27/OGCG.git

**Результати**

****



**Висновки**

Виконана робота присвячена розробці та програмній реалізації алгоритму знаходження трикутника найбільшої площі, вписаного в опуклу оболонку заданої множини точок. Реалізовано **алгоритм обертального каліпера** (rotating calipers) для знаходження трикутника найбільшої площі серед точок оболонки. Є автоматичний вибір 3 точок з максимумом площі, а також обертання та візуалізація трикутника. Код працює без потреби ручного вводу координат.

**Науковий аналіз виконаної роботи:**

В ході роботи було підтверджено, що для знаходження трикутника максимальної площі достатньо розглядати лише вершини опуклої оболонки множини точок. Запропонований алгоритм ефективно використовує цю властивість, застосовуючи принцип "обертового каліпера" для оптимізації пошуку третьої вершини трикутника для фіксованої основи. Це дозволило досягти обчислювальної складності O(k^2), де k – кількість вершин опуклої оболонки, що є значним покращенням порівняно з наївним перебором O(N^3).

Програмна реалізація на Python з використанням бібліотек numpy, matplotlib та scipy.spatial.ConvexHull демонструє наочність та працездатність алгоритму. Згенерована візуалізація чітко показує опуклу оболонку, виділений трикутник найбільшої площі та внутрішні точки, що підтверджує коректність реалізації.

**Проблеми та перспективи щодо розробки алгоритму та програмної реалізації**

* **Проблеми:**

Для дуже великих наборів точок (N >> 10000) та у випадку, коли опукла оболонка містить значну частину від початкових N точок (k≈N), складність O(k^2) може бути все ще високою.

Реалізація обертового каліпера потребує уваги до граничних умов та циклічного обходу вершин опуклої оболонки.

* **Перспективи:**
  + **Покращення ефективності алгоритмів.** Можливе подальше зниження складності до O(klogk) або O(k) за допомогою більш складних геометричних структур даних або алгоритмів (наприклад, алгоритмів, що використовують бінарний пошук для знаходження третьої вершини). Це особливо актуально для застосувань, що вимагають обробки даних у реальному часі.
  + **Розширення до 3D.** Розробка аналогічних алгоритмів для знаходження тетраедра найбільшого об'єму, вписаного в опуклу оболонку в тривимірному просторі. Це значно складніша задача, що вимагає інших підходів.
  + **Інтеграція з іншими системами.** Вбудовування розробленого модуля в більш складні програмні комплекси для комп'ютерного зору, робототехніки або геоінформаційних систем.
  + **Паралелізація.** Розробка паралельних версій алгоритму для використання багатоядерних процесорів або GPU, що дозволить обробляти надвеликі набори даних.

**Умови покращення ефективності алгоритмів для подальших розробок**

1. **Використання більш досконалих алгоритмів побудови опуклої оболонки.** Хоча ConvexHull є ефективним, для специфічних випадків можуть існувати більш швидкі алгоритми.
2. **Реалізація алгоритму "обертового каліпера" в чистому вигляді.** Для знаходження трикутника найбільшої площі можна використовувати більш формалізований підхід "обертового каліпера", який, як правило, дає складність O(k).
3. **Використання бінарного пошуку.** Як зазначено в джерелах, алгоритм бінарного пошуку вершин дозволяє знайти трикутник максимальної площі за O(nlogn) часу. Цей підхід є більш досконалим і заслуговує на окрему увагу в подальших розробках.
4. **Оптимізація обчислення площі.** Хоча np.cross є ефективним, для великої кількості обчислень можна розглянути мікрооптимізації.
5. **Тестування на різних розподілах точок.** Проведення тестів на різних типах вхідних даних (рівномірний розподіл, кластеризовані точки, точки на одній лінії) дозволить оцінити стійкість та ефективність алгоритму в різних сценаріях.

**Джерела та посилання до них:**

1. <https://cimec.org.ar/foswiki/pub/Main/Cimec/GeometriaComputacional/DeBerg_-_Computational_Geometry_-_Algorithms_and_Applications_2e.pdf>(Berg, M. de, van Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry: Algorithms and Applications. Springer, 2008, сторінки 244–258, 292–300)
2. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.ConvexHull.html>
3. <https://csc.knu.ua/media/filer_public/c7/83/c783a9ff-5591-4b2c-8ebf-2dce9bd4cb28/analiz_optim_240521_tereshchenko.pdf> (Сторінки 63-70)
4. <https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D1%83%D0%BA%D0%BB%D0%B0_%D0%BE%D0%B1%D0%BE%D0%BB%D0%BE%D0%BD%D0%BA%D0%B0>
5. <https://cp-algorithms.com/geometry/convex-hull.html>
6. <https://www.youtube.com/watch?v=Q2eXXLDFDbo>
7. <https://en.wikipedia.org/wiki/Rotating_calipers>