МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТЕХНІКИ

КАФЕДРА ВИЩОЇ ТА ПРИКЛАДНОЇ МАТЕМАТИКИ

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Спеціальність 113 «Прикладна математика»

на тему:

«АЛГОРИТМИ ПОБУДОВИ МІНІМАЛЬНИХ ОПУКЛИХ ОБОЛОНОК ТА ЇХ АНАЛІЗ»

Студента групи 401ПМ Гурбанмирадова Арслана

Керівник роботи:

к.ф.-м.н., старший викладач Приставка Ю.В.

Завідувач кафедри

к.ф.-м.н., доцент Ічанська Н.В.

Полтава 2021

РЕФЕРАТ

Дипломну роботу виконано на 45 аркушах, вона містить перелік посилань на використані джерела з 15 найменувань. У роботі наведено 24 рисунки.

Об’єкт дослідження – мінімальні опуклі оболонки.

Предмет дослідження – алгоритми знаходження мінімальних опуклих оболонок.

Мета роботи – розробка програми для побудови мінімальної опуклої оболонки.

Методи: теоретичні — метод пошуку, метод аналізу наукових джерел, порівняння та узагальнення; емпіричні — спеціальні методи моделювання, систематизації, метод зіставлення різних поглядів на аспекти досліджуваної проблеми.

При розробці програми використовувалась мова програмування «Python» з доповненням «MatPlotLib».

Ключові слова: МІНІМАЛЬНА ОПУКЛА ОБОЛОНКА, МЕТОД ГРЕХЕМА, МЕТОД ДЖАРВІСА, МЕТОД ЧЕНА, PYTHON, MAPLE.

ABSTRACT

Bachelor's degree work: 46 p., 24 pictures, 15 sources.

The object of research is minimal convex shells.

The subject of research is algorithms for finding minimal convex shells.

The purpose of the work is to develop a program for constructing the minimal convex hull.

Methods: theoretical – search method, method of analysis of scientific sources, comparison and generalization; empirical – special methods of modeling, systematization, method of comparing different views on aspects of the problem.

Key words: MINIMUM CONVEX COVER, GRAHAM'S METHOD, JARVIS'S METHOD, CHEN'S METHOD, PYTHON, MAPLE.

Зміст

Вступ

Знаходження мінімальної опуклої оболонки (МОО, від англ. convex hull) множини вершин графа є фундаментальною проблемою у багатьох сферах сучасних наукових досліджень . Розв’язання цієї задачі передбачає формування найменшого опуклого набору, який містить всі вузли, наявні у графі. Відомо, що МОО є поширеним інструментом в системах автоматизованого проектування та пакетах комп’ютерної графіки. Наприклад, криві Без’є (Bezier curve), які застосовуються в програмах Adobe Photoshop, GIMP та CorelDraw для моделювання гладких ліній, повністю лежать в опуклій оболонці своїх контрольних вузлів . Ця властивість значно спрощує знаходження точок перетину кривих та дозволяє здійснювати їх трансформації (перенесення, масштабування, обертання та інші) за допомогою відповідних контрольних вузлів. Формування деяких шрифтів та анімаційних ефектів у пакеті Adobe Flash також відбувається за допомогою квадратичних кривих Без’є у складі сплайнів. Слід зазначити про застосування опуклих оболонок у географічних інформаційних системах (Geographical Information Systems), алгоритмах маршрутизації та при визначенні оптимальних шляхів обходу перешкод. З їх використанням запропоновані методи розв’язання складних задач оптимізації.

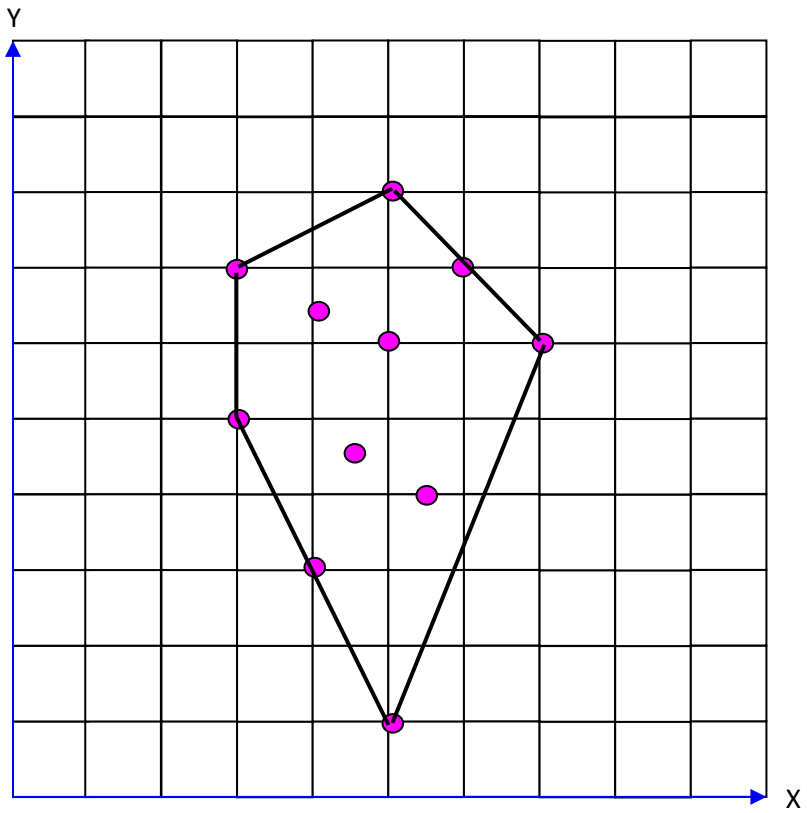
Останні десятиріччя пов’язані з бурхливим зростанням обсягу даних в наукових дослідженнях, які оброблюються інформаційними системами. За оцінками IBM, щодня в світі створюється близько 15 петабайт нової інформації. Тому в сучасній науці виник окремий напрям Big Data, пов’язаний з дослідженням великих наборів даних. Однак, більшість відомих алгоритмів формування МОО мають часову складність O (nlogn) , що обумовлює їх непридатність при формуванні розв’язків для графів великої розмірності. Тому виникає необхідність розробки ефективних алгоритмів зі складністю, близькою до лінійної O(n) . Попри інтенсивні дослідження, які тривали протягом останніх 40 років, проблема розробки ефективних алгоритмів формування МОО досі залишається відкритою. Основним досягненням є формування низки методів, які передбачають визначення екстремальних точок вихідного графа та

побудову системи з’єднань між ними . До них належать алгоритм Джарвіса (Jarvis march), Грехема (Graham Scan), швидкої оболонки (QuickHull), «Розділяй і володарюй» («Divide and conquer») та багато інших. Мова програмування «Python», яка була використовувана мною під час реалізації методів, це — інтерпретована об'єктно-орієнтована мова програмування високого рівня з динамічною семантикою. Розроблена в 1990 році Гвідо ван Россумом. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднання існуючих компонентів. Пайтон підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор «Python» та стандартні бібліотеки доступні як у скомпільованій так і у вихідній формі на всіх основних платформах. В мові програмування Пайтон підтримується декілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована. Мова має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис, динамічна обробка типів, а також те, що це інтерпретована мова, роблять її ідеальною для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Постановка задачі

Нехай на площині дано множина точок . Потрібно побудувати опуклий многокутник щоб для будь-якої точки із заданої множини було вірно одне з трьох тверджень :



а) точка є вершиною побудованого багатокутника

б) точка належить одній із сторін багатокутника ;

в) точка лежить всередині багатокутника.

(на малюнку показаний приклад опуклої оболонки) .[1,2]

2.2. Поняття опуклої оболонки. Властивості

У математиці, опукла оболонка для множини точок *X* дійсному векторному просторі V –це мінімальна опукла множина, що містить *X*.

В обчислювальній геометрії, прийнято використовувати термін "опукла оболонка" для границі мінімальної опуклої множини, що містить дану не порожню скінченну множину точок на площині. Якщо точки колінеарні, опукла оболонка являє собою ламану лінію.

Нехай на площині задано кінцева множина точок А. Оболонкою цієї множини називається будь-яка замкнута лінія Н без само перетинів така, що всі точки з А лежать всередині цієї кривої. Якщо крива Н є опуклою (наприклад, будь-яка дотична до цієї кривої не перетинає її більше ні в одній точці), то відповідна оболонка також називається опуклою. Нарешті, мінімальний опуклою оболонкою (далі коротко МОО) називається опукла оболонка мінімальної довжини (мінімального периметра).

Пошук опуклої оболонки множини точок є важливою задачею, яка часто є частиною більше загальної задачі. Є багато алгоритмів для знаходження опуклої оболонки.

2.2. Застосування мінімальної опуклої оболонки до розв'язання задач різного типу

Однією з найпоширеніших проблем в математиці та комп’ютерній науці є побудова опуклих оболонок. Задача побудови опуклої оболонки має давню історію. Вона є однією з перших задач обчислювальної геометрії, з якої почала зароджуватися ця наука.

Будучи однією з найстаріших задач в області обчислювальної геометрії, вона заклала основу для побудови алгоритмів для багатьох геометричних проблем і проблем побудови графів. Хоча алгоритми (з трудоємкістю O(n^4)) для обчислення опуклої оболонки існували з часів середньовіччя, перший ефективний алгоритм не з'являвся до роботи Чанда і Капура (1970), заснованого

на ідеї «завертання подарунка». Широка різноманітність алгоритмів була розроблена з тих пір, у тому числі оптимальні алгоритми (Чан, 1996). Крім того, удосконалення цих алгоритмів призвело до подальшого розвитку в інших областях математики та інформатики.

Важливість задачі побудови опуклої оболонки полягає не тільки у величезній кількості додатків, де вона застосовується (розпізнавання образів, обробка зображень, бази даних, задачі розкрою та компонування матеріалів, математична статистика), але також і через користь опуклої оболонки як інструменту вирішення безлічі завдань обчислювальної геометрії.

Це завдання дозволяє вирішити цілий ряд інших, іноді з першого погляду не пов'язаних з ним питань: побудова діаграм Вороного, побудова тріангуляції і т.д. Побудова опуклої оболонки кінцевої множини точок на площині досить широко досліджена і має безліч застосувань. Досить широко алгоритми побудови опуклої оболонки використовуються в геонформатиці та геоінформаційних системах.

Метод побудови оболонок набув широкого поширення в системах тривимірної машинної графіки. Свою назву він отримав від використання простих по конструкції тривимірних опуклих фігур – оболонок , які охоплюють об'єкт або його складові частини і дозволяють порівняно просто виявляти частину простору, де знаходиться об'єкт.

Серед найпоширеніших сфер застосувань опуклих оболонок: пошук оптимального шляху, виявлення та запобігання зіткнень, лінійне програмування, побудова обчислювальної сітки, розпізнавання образів.

Розглянемо застосування алгоритмів побудови опуклих оболонок в реальному світі.

Пошук оптимального шляху

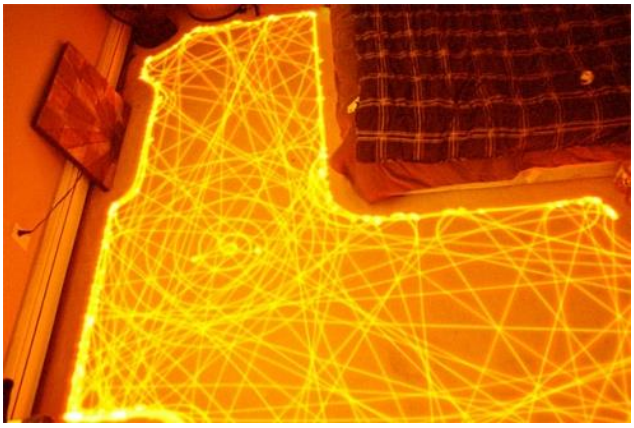
Пошук шляху є процесом переміщення штучного інтелектуального об’єкта з однієї точки в іншу. Це дуже важливе завдання, зокрема, в дизайні комп'ютерних ігор, оскільки дозволяє персонажу в грі пересуватися в реалістичній манері, уникаючи перешкод.

Загальним підходом до вирішення проблеми є випадок, коли дані складаються з геометричних об'єктів, і в наявності двійковий тип місцевості, тобто всі об'єкти є абсолютними перешкодами в тому сенсі, що крізь них не можна пройти, а вся місцевість, яку не займають об'єкти, вважається безперешкодною незалежно від зміни швидкості транспортного засобу або інших параметрів. У цьому випадку евклідової дистанції найкоротший шлях є найшвидшим шляхом. Основна ідея подолання перешкод полягає в наступному: спочатку будуємо опуклу оболонку всіх об'єктів. Кути усіх опуклих багатокутників визначають набір вершин. Усі вершини поєднують ребрами. Після цього здійснюється певна фільтрація цих меж і, нарешті, відбувається пошук найкоротшого правильного шляху між джерелом і місцем призначення уздовж набору ребер з використанням стандартних алгоритмів на графах.

Основна відмінність методів цього класу полягає у тому, яким саме чином виконується фільтрація. Мета фільтрації полягає у видаленні ребер, які ніяк не можуть належати до найкоротшого шляху і таким чином значно знизити кількість шляхів, які повинні бути розглянуті. Перший крок, як правило, полягає у видаленні ребер, які (геометрично) перетинаються з будь-якою іншою опуклою оболонкою, тобто ребер, які проходять через перешкоду. Після цього застосовуються різні правила, на основі яких видаляються занадто далекосяжні шляхи тощо.

Розглянемо декілька прикладів застосування цього у реальному світі.

Робот-пилосос є інноваційним пристроєм, оснащений штучним інтелектом і призначений для автоматичного прибирання приміщень. Під час прибирання роботи автоматично рухається по заданій поверхні, прибираючи її Зустрівши на шляху перешкоду, робот приймає рішення про спосіб її подолання на основі спеціальних алгоритмів. Здатність виявляти стіни, столи, стільці та інші предмети здається майже неймовірною. Але насправді, здатність прокладати шлях навколо перешкод заснована на здатності визначити межі навколо цих перешкод.



У двовимірному просторі, опукла оболонка об'єкту - це найменший багатокутник, що щільно прилягає навколо зовнішніх кутів об'єкта. Переміщуючись по кімнаті, робот може виявити ці кути і межі, що охоплюють перешкоду. Володіючи цією інформацією, він може обчислити оптимальний шлях (мінімальну відстань) між будь-якими двома точками в тій або іншій кімнаті. Це особливо важливо для визначення необхідних закономірностей при прибиранні поверхні, і визначення шляху для повернення до своєї заряджувальної станції.

Ще одним прикладом використання опуклих оболонок при пошуку оптимального шляху є програма НАСА з дослідження планети Марс за допомогою марсоходів.

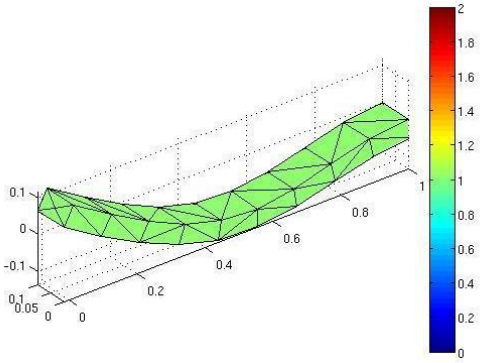
Для поліпшення навігації, дослідження та автономії марсохода, одним із основних завдань дослідження було поліпшення здатності марсохода під час навігації у навколишньому середовищі, зберігаючи точне значення своєї позиції. При цьому розглядалися 4 відповідні області: генерація динамічної послідовності, планування автономного шляху, візуальна локалізація і оцінка стану. Усі дослідження проводилися з прототипом марсохода Роккі 7.

Зокрема, при дослідженні у області планування автономного шляху були здійснені наступні завдання. Роккі 7 використовував локальний сенсорний планувальник шляху, що має назву Rover-Bug. Він був розроблений для пристроїв, які мають обмеження на діапазон сенсора, поле зору і обробку. Для забезпечення руху використовувалися два основні режими роботи – рух до об’єкту і рух відносно меж. Rover-Bug працює використовуючи карту місцевості, щоб побудувати карту опуклих оболонок навколо усіх перешкод в певному діапазоні чутливості. Ці оболонки потім об'єднуються і збільшуються для уявлення конфігурації простору відсканованої місцевості. Потім будується графік, щоб визначити, чи існує безперешкодний шлях до оболонки сканованої області в напрямку мети. Якщо він існує, марсохід рухається у цьому напрямку і процес повторюється для кінця сегмента контуру сканованої області. Якщо такого шляху не існує, для найбільш перспективних сторін поточного зображення застосовуються стерео знімки і процес повторюється. У деяких випадках безперешкодні шляхи приводять марсохід до краю сканованої оболонки, яка все ще ускладнена перешкодою. У цьому випадку марсохід почне використовувати свої вбудовані камери для запуску режиму руху відносно меж, доки шлях до цілі не міститиме перешкод. [3, 4]

Побудова обчислювальної сітки

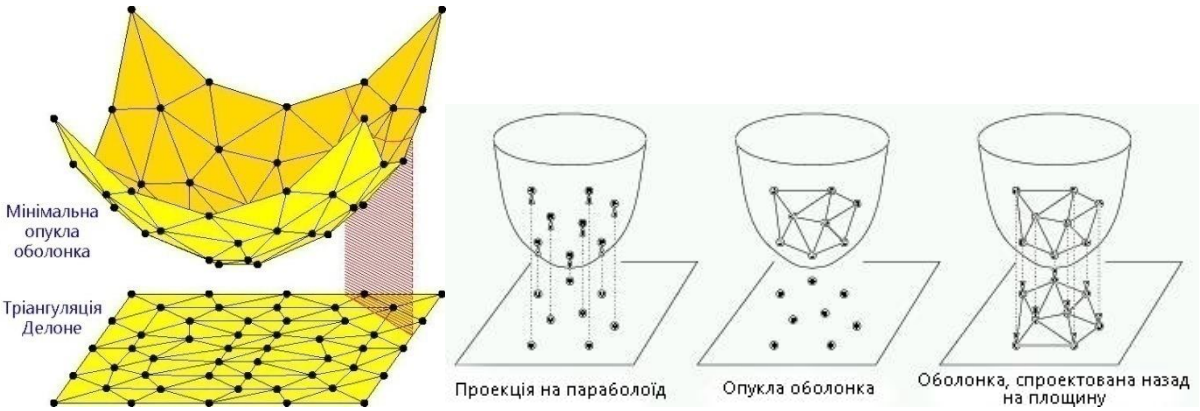
Перед тим, як побудувати будь-яку складну будівлю, інженер повинен з великою точністю знати, чи буде вона протистояти силам, що діють на неї. Деякі будівлі, такі як хмарочоси, занадто великі і дорогі, щоб ризикувати побудовою без абсолютної впевненості у їх структурній цілісності. Замість цього, інженери використовують процес, що має назву аналізу кінцевих

елементів, коли тестується математична модель будівлі проти діючих на неї сил. Модель будується шляхом розбиття складної форми будівлі у набір простіших форм, що мають назву обчислювальної сітки кінцевих елементів.



Обчислювальна сітка кінцевих елементів, що представляє балку, яка прогинається під дією зовнішніх сил

Опукла оболонка відіграє важливу роль у побудові сітки кінцевих елементів. При вибірці вузлових точок в n-вимірному просторі, можна спрогнозувати побудову цих точок в n+1-вимірному параболоїді. Як тільки опукла оболонка цих n+1-вимірних точок знайдена, нижня сторона опуклої оболонки може бути спроектована назад в n-вимірний простір для побудови n-вимірної сітки кінцевих елементів. [5]



Розпізнавання образів

Численні дослідження зосереджені на швидкому наближенні різних геометричних структур в комп'ютерній графіці. Крім того, побудова точної і наближеної опуклої оболонки використовується в якості попередньої обробки або проміжного кроку для вирішення багатьох проблем в комп'ютерній графіці.

Так, обмежувальні рамки використовуються для наближеного знаходження об'єкта і як дуже простий дескриптор його форми. Наприклад, в обчислювальній геометрії та її додатках коли потрібно знайти перетин множин об'єктів, початковою перевіркою буде перетин між їх обмежувальними рамками. Оскільки це зазвичай набагато легша операція, ніж перевірка фактичного перетину (тому що вона вимагає тільки порівняння координат), це дозволяє швидко виключати з перевірки пари, які знаходяться далеко одна від одної. Опукла оболонка в цих ситуаціях дуже добре заміняє обмежувальну рамку. [6]

Прикладом розпізнавання образів за допомогою опуклих оболонок є алгоритми роботи дорожніх камер, що часто використовуються для фіксації порушень дорожнього руху водіями. У деяких містах, кількість порушень правил дорожнього руху може бути достатньо великою, тому перебирати кожне зображення вручну, щоб визначити точний номерний знак досить незручно і неефективно. Алгоритми розпізнавання образів використовуються для автоматичного визначення не тільки номерного знаку, а й стану логотипу та реєстраційного номеру.

Для виділення літер з усього зображення до кожної з них застосовується опукла оболонка. З цих оболонок, інший алгоритм визначає порожній простір всередині оболонки і зіставляє його з формою шаблону, що відповідає певній літері або числу. Алгоритми для цього були розроблені в 1970-х роках (Туссен, 1978) і вперше реалізовані в технології управління рухом у кінці 1980-х (Гонсалес та ін., 1989).

У даний час відома досить велика кількість алгоритмів побудови опуклої оболонки, проте існує проблема, пов'язана з недостатньою кількістю робіт, присвячених аналізу та огляду цих алгоритмів, особливо в тривимірному просторі.

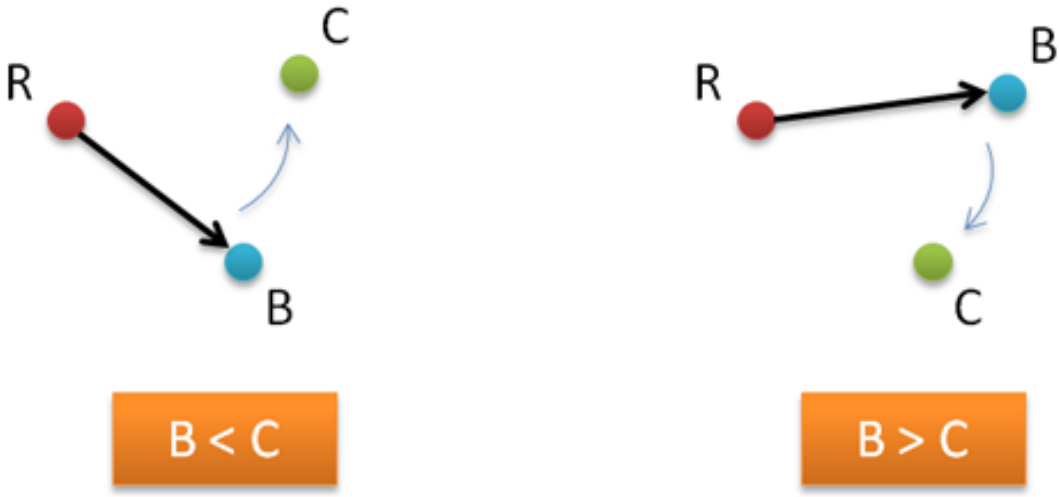
2.4. Методи знаходження мінімальної опуклої оболонки

2.4.1.Метод Грехема

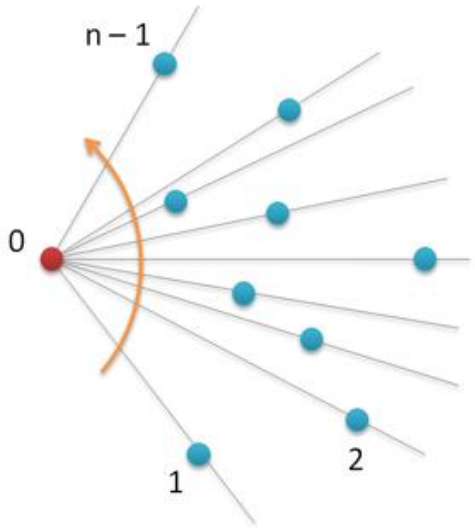
Алгоритм Грехема — метод знаходження опуклої оболонки для скінченної множини точок на площині за час O($n \log n$). Названий на честь Рональда Грехема, який опублікував первісний варіант алгоритму в 1972 році. У цьому алгоритмі задача про опуклу оболонку вирішується за допомогою стека, сформованого з точок-кандидатів. Всі точки вхідної множини заносяться в стек, а потім точки, які не є вершинами опуклої оболонки, з часом видаляються з нього. По завершенні роботи алгоритму в стеку залишаються тільки вершини оболонки в порядку їх обходу проти годинникової стрілки.

Цей алгоритм складається з трьох кроків. На першому кроці шукається будь-яка точка в A , гарантовано що входить до МОО . Неважко збагнути, що такою точкою буде , наприклад , точка з найменшою x - координатою (най лівіша точка в A). Цю точку (будемо називати її стартової) переміщаємо в початок списку , вся подальша робота буде проводитися з рештою точками . За деяких міркувань , вихідний масив точок A нами змінюватись не буде , для всіх маніпуляцій з точками будемо використовувати непряму адресацію : заведемо список P , в якому будуть зберігається номери точок (їх позиції в масиві A). Отже, перший крок алгоритму полягає в тому , щоб першою точкою в P виявилася точка з найменшою x - координатою .

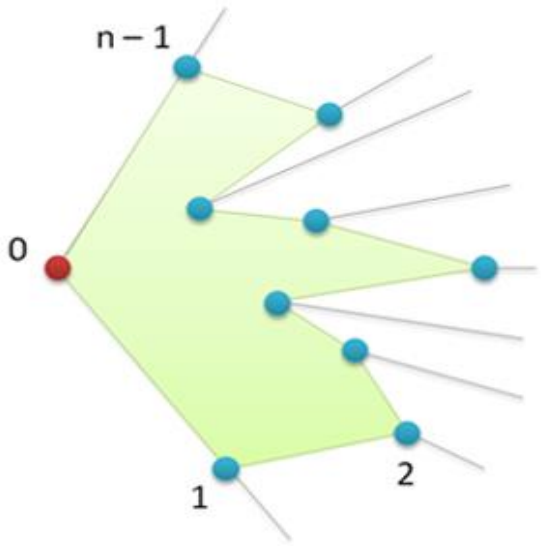
Другий крок в алгоритмі Грехема - сортування всіх точок (крім P [0]-ої), хто найлівіший, щодо стартової точки R = $A_{p[0]}$.Будемо говорити, що B<C, якщо точка C знаходиться по ліву сторону від вектора RB.



Для виконання такого упорядкування можна застосовувати будь-який алгоритм сортування, заснований на попарному порівнянні елементів, наприклад, швидке сортування, або сортування вставками .Результат сортування можна проілюструвати наступним малюнком.



Якщо ми тепер з'єднуємо точки в отриманому порядку, то отримаємо багатокутник, який, однак, не є опуклим.



Переходимо до третьої дії. Все, що нам залишилося зробити, так це зрізати кути. Для цього потрібно пройтися по всіх вершин і видалити ті з них, в яких виконується правий поворот (кут в такій вершині виявляється більше розгорнутого). Заводимо список S і поміщаємо в нього перші дві вершини (вони, знову ж, гарантовано входять в МОО).Потім переглядаємо всі інші вершини, і відслідковуємо напрямок повороту в них з точки зору останніх двох вершин в списку S: якщо цей напрямок негативно, то можна зрізати кут видаленням з стека останньої вершини. Як тільки поворот виявляється позитивним, зрізання кутів завершується, поточна вершина заноситься в список. У підсумку в списку S виявляється шукана послідовність вершин, причому в потрібній нам орієнтації, визначальна МОО заданої множини точок A.

Недоліки:

• Теоретично алгоритм Грехема є оптимальним в гіршому випадку, однак він не оптимальний в середньому.

• Алгоритм не є відкритим, для його роботи необхідно апріорне знання всього набору точок.

• Невідомі узагальнення алгоритму на простору розмірності більше двох.

Переваги:

• Алгоритм Грехема має гарантовану лінійно-логарифмічну трудоємкість.

2.4.2.Метод Джарвіса

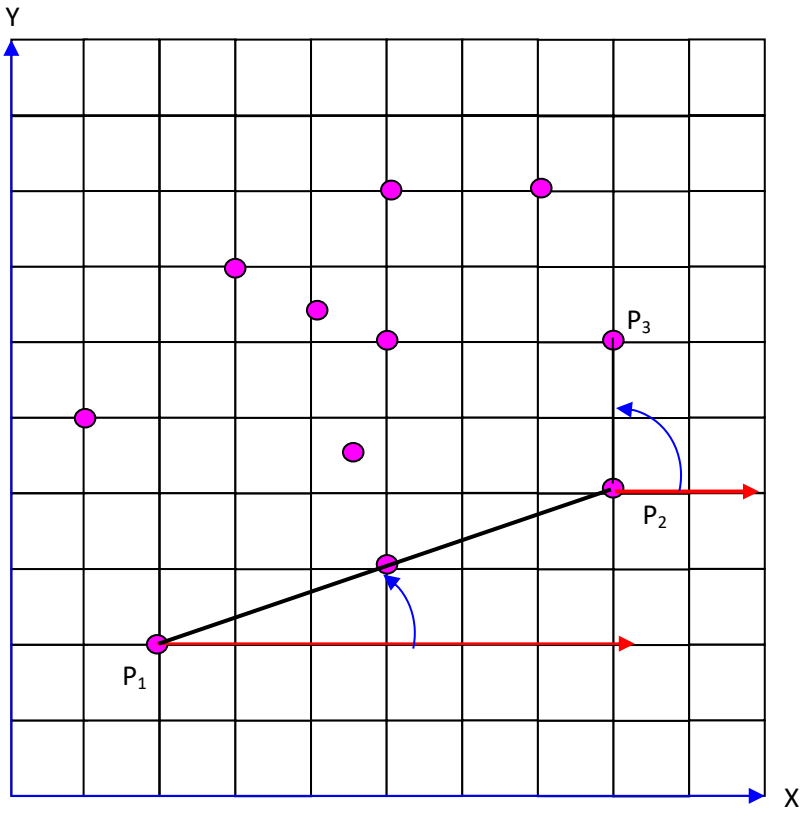
Цей алгоритм , запропонований Джарвісом в 1973, дуже легкий для розуміння. Він також відомий під назвою «метод загортання подарунка » , так як цей алгоритм оброблює точки опуклої оболонки одну за одною, як якщо б ми обертали безліч точок аркушем паперу . Джарвіс звернув увагу на те , що багатокутник , яким є опукла оболонка , з однаковим успіхом можна задати впорядкованим безліччю , як його ребер , так і його вершин. Якщо задано безліч точок , то досить важко швидко визначити , є чи ні деяка точка крайньою. Однак якщо дано дві точки , то чи не безпосередньо можна перевірити , чи не з'єднує їх відрізок ребром опуклої оболонки.

Теорема: Відрізок l , що визначається двома точками , є ребром опуклої оболонки тоді і тільки тоді , коли всі інші точки заданої множини лежать на l або з одного боку від нього.

Джарвіс використав цю ідею , ми розглянемо запропонований ним алгоритм .

1 . Знайдемо точку (позначимо її P1) , яка свідомо буде належати опуклій оболонці (наприклад , точку з мінімальною координатою по у) і запам'ятаємо її

2 . Для всіх інших точок визначимо полярні кути щодо P1 і виберемо з них ту , чий полярний кут є найменшим (якщо таких точок декілька, то виберемо найбільш віддалену від P1) . Це буде друга точка опуклої оболонки - P2 , запам'ятаємо її, наприклад в масиві.



3 . Повторимо крок 2 для другої і наступних точок опуклої оболонки .

Оцінимо час роботи цього методу. Для кожної вершини опуклої оболонки знаходяться полярні кути всіх інших точок (за O (n)) , і серед цих полярних кутів шукається мінімум (ще за O (n)) . Якщо оболонка складається з h точок , то час роботи одно O (hn) . Найгіршим буде випадок , коли всі точки вихідного безлічі належать опуклій оболонці . У цьому випадку час роботи методу складе O (n2) .

Переваги:

• Алгоритм Джарвіса може бути застосований у просторах розмірності більше двох.

• При апріорному знанні малості числа вершин оболонки в теорії дає оптимальну трудомісткість.

Недоліки:

• Висока квадратична трудомісткість в гіршому випадку

Оцінимо складність алгоритму Джарвіса . Перший крок лінійний за n. З другим все цікавіше. У нас є вкладений цикл , число зовнішніх ітерацій дорівнює числу вершин ч в МОО , число внутрішніх ітерацій не перевищує n. Отже , складність всього алгоритму дорівнює O (HN) . Незвичайним в цій формулі є те , що складність визначається не тільки довжиною вхідних даних , але і довжиною виходу (output-sensitive algorithm). А далі як карти точки ляжуть . У гіршому випадку всі крапки з входять в МОО (тобто вже саме по собі опуклий багатокутник) , тоді h =n і складність підскакує до квадратичної . У кращому випадку (за умови , що точки з чи не лежать на одній прямій) h = 3 і складність стає лінійною. Залишилося заздалегідь зрозуміти, який у нас випадок , що зробити не так просто , можна тільки виходити з характеру завдання - якщо точок багато і вони рівномірно заповнюють деяку область , то (можливо) Джарвіс буде швидше , якщо ж дані зібрані на межі області , то швидше буде Грехем .

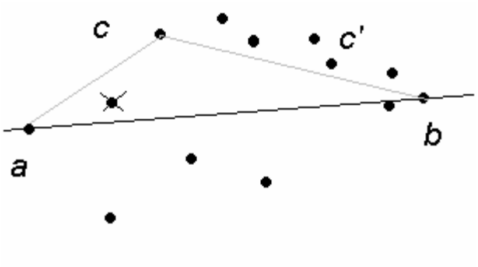
2.4.3.Метод Чена

Алгоритм Чана , запропонований Тімоті М. Чаном у 1996, алгоритм побудови опуклої оболонки кінцевої множини точок на площині. Є комбінацією двох більш повільних алгоритмів (сканування по Грехему O (n log n) і загортання по Джарвісу O (nh)). Недоліком сканування по Грехему є необхідність сортування всіх точок по полярному кутку, що займає досить багато часу O (n log n). Загортання по Джарвісу вимагає перебору всіх точок для кожної з h точок опуклої оболонки, що в гіршому випадку займає O (n^2).

2.3.4.Метод швидкої побудови

Метод швидкої побудови-алгоритм побудови опуклої оболонки на площині. Використовує ідею швидкого сортування Хоара. Для побудови опуклої оболонки були створені алгоритми, що нагадують швидке сортування. Такі алгоритми називаються швидкими методами побудови оболонки. одними з перших такий алгоритм запропонував Бікату.

Суть алгоритму полягає в тому , що вихідна множина S з N точок розбивається на дві підмножини , кожне з яких буде містити одну з двох ламаних , які при зєднанні утворюють опуклу оболонку. Для початку потрібно визначити дві точки , які будуть сусідніми вершинами опуклої оболонки. Можна взяти саму ліву (a) інайправішу (b) вершини .



Після чого потрібно знайти точку максимально віддалену відпрямий ab .Алгоритм вибору з грунтується на тому факті , що ця точка визначає трикутник ,потрапляючи в який , будуть відкидатися точки . Для того , щоб

максимізувати число точок , що підлягають видаленню , розумним було б знайти точку з такою , щоб вона максімізувала площу трикутника ABC. Одночасно з тим , точка с повинна лежати по ліву сторону від відрізка AB . Таким чином поєднавши ці дві умови , визначаємо , що за точка з вибирається та точка з S \ { a, b} , яка максимізує значення Δ (a, b , c).Всі точки , що лежать у трикутнику ABC виключаються з подальшого розгляду , для того , щоб на кожному кроці виключати якомога більше точок , крапка з вибирається за кри - терию найбільшою площі трикутника ABC. Решта точки будуть ділитися на два підмножини: точки , які лежати лівіше ac і точки , які лежать правіше cb . кожне з них містить ламані які в поєднанні з a, b і c з дають опуклу оболонку. З кожним з них проробляємо те ж саме. У підмножині точок, що лежать лівіше CB вибираємо c ' за цими ж ознаками , за якими обирали точку c, максимально віддалену від cb, яка ділить його на три частини. З них одна викидається , а інші діляться знову . Це реалізується рекурсивної процедурою , яка для даного їй безлічі повертає відповідаючу частина опуклої оболонки . Кінець алгоритму .

У разі , коли потужність кожного , з підмножин , на яке ділиться безліч , що неперевершує деякої константи помноженої на потужність множини , отримуємо складність алгоритму , як і у швидкій сортуванні O (n log n) . Але в гіршому випадку може зайняти час O(n^2).

Переваги:

• Алгоритм можна розпаралелювати.

• Алгоритм узагальнюємо на довільну розмірність.

Недоліки:

• Висока квадратична трудомісткість в гіршому випадку

Алгоритм Чена

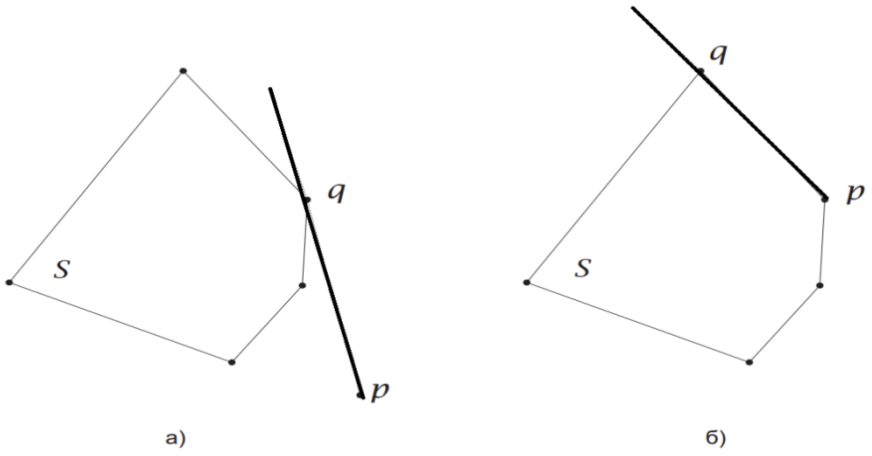
Нехай дано вихідна безліч точок P , для якого шуканої є його опукла оболонка CH (P). Покладемо , що зафіксований деякий цілочисельний параметр $m < 1 < n$.

1 . Розділимо безліч P на $\frac{n}{m}$ непересічних підмножин P_i так , щоб у кожній підмножині було не більше ніж m точок . Очевидно , що таке розбиття завжди можливо здійснити.

2 . Побудуємо опуклі оболонки CH (P_i) для множин P_i будь-яким з оптимальних в гіршому випадку алгоритмів (наприклад , алгоритмом Грехема).

3 . Знайдемо точку $P_{start} \in$ P , яка буде гарантовано включена в опуклу оболонку CH (P).

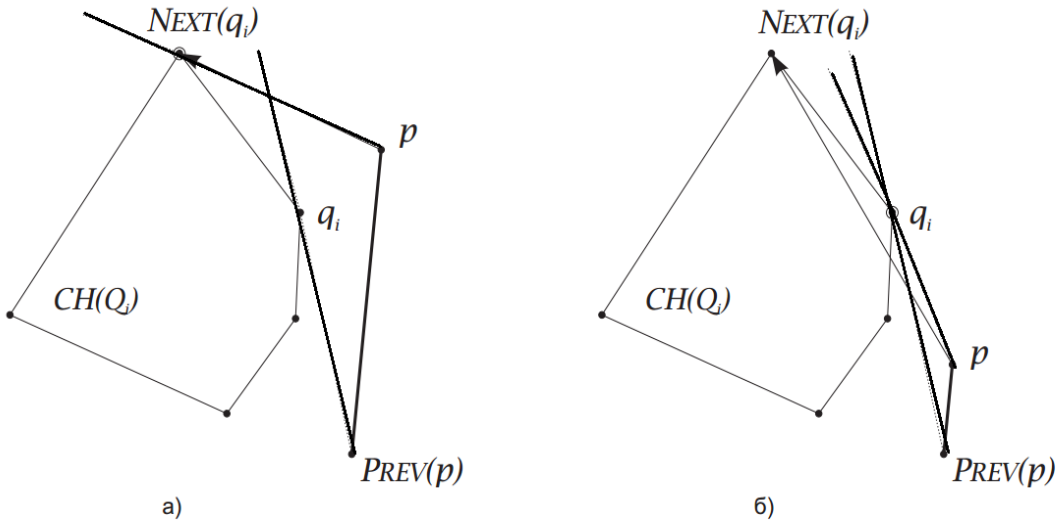
4 . Будемо виконувати кроки , знаходячи кожного разу таку точку , яка є наступною вершиною опуклої оболонки в порядку обходу проти годинникової стрілки. Коли чергова знайдена точка p співпаде з p_{start} будемо вважати , що алгоритм побудував опуклу оболонку CH (P).Для знаходження наступної вершини опуклої оболонки буде використовуватися інформація про вже побудованих опуклих оболонках CH (P_i) . Базовою операцією будемо вважати знаходження правої опорної точки опуклого багатокутника S щодо поточної точки p . Права опорна точка опуклого багатокутника S відносно точки p - це така точка q $\in$ S , що будь-яка точка багатокутника S лежить по ліву сторону від орієнтованого відрізка $[p,q]$ або на ньому.



На кожному кроці необхідно знайти всі праві опорні точки q_i опуклих багатокутників СН (P_i) відносно точки p і лінійним пошуком вибрати серед них таку точку r, щоб будь-яка точка q_i лежала ліворуч від орієнтованого відрізка $[p,r]$ або на ньому. Точка r буде наступною вершиною опуклої оболонки в порядку обходу проти годинникової стрілки.

Праву опорну точку опуклого багатокутника S відносно точки p можна знайти двійковим пошуком за час O (log m), де m - кількість вершин в багатокутнику . Але можна поліпшити цю асимптотику , якщо помітити , що на кожному наступному кроці , права опорна точка qi або залишиться такою ж, або прийме значення однієї з наступних вершин СН (P_i) у бік обходу проти годинникової стрілки , причому ніяка вершина не може бути пройдена більше ніж два рази. Таким чином , кожен раз , коли потрібно знайти праву опорну точку q_i щодо нової точки p , береться старе значення q_i і перевіряється наступна за q_i точка Next (q_i) : якщо виявляється, що точка q_i лежить лівіше орієнтованого відрізка $[p,Next(q_i)]$ або строго на ньому ,то в якості q_i береться Next (q_i) і операція повторюється , в іншому випадку опорною точкою вважається точка q_i.

До знаходження правих опорних точок на першому кроці , q_iпопередньо ініціалізується будь-якими з вершин відповідних багатокутників СН (P_i) .



Переваги:

• Алгоритм має трудоємність O (n log h) в середньому, при цьому в гіршому випадку він є оптимальним.

Недоліки:

• Алгоритм не є відкритим, тобто необхідно апріорне знання всієї множини точок.[7]

3. Практична частина

3.1.Інформація про середовище

При розробки програми для побудови мінімальної опуклої оболонки, використовувалась мова програмування «Python» з доповненням « MatPlotLib»-це бібліотека призначена для побудови графіків .Всі необхідні файли, ви можете знайти.

3.2.Опис головних функцій програми

```
def rotate(a,b,c):
  return (b[0]-a[0])*(c[1]-b[1])-(b[1]-a[1])*(c[0]-b[0])
```

функція «rotate» показує положення точки с відносно точки а і в, якщо точка лежить зліва, відносно прямої яка проведена через ці 2 точки, функція поверне додатнє число , якщо ж праворуч, поверне від’ємне число.

```
def jarvis(A):
    start = time.time()
    n = len(A)
    P = range(n)
    for i in range(1,n):
      if A[P[i]][0]<A[P[0]][0]:
        P[i], P[0] = P[0], P[i]
    H=[P[0]]
    del P[0]
    P.append(H[0])
    while True:
        right=0
        for i in range(1,len(P)):
            if rotate(A[H[-1]],A[P[right]],A[P[i]])<0:
                right=i
```

```
        if P[right]==H[0]:
            break
        else:
            H.append(P[right])
            del P[right]
    H.append(H[0])
    x=(time.time() - start)
    global jarvis_time
    jarvis_time= "%.6f" % (x)
    for i in range(len(H)):
        x_final.append(points[int(H[i])][0])
        y_final.append(points[int(H[i])][1])
```

фукнція «jarvis» э реалізацією методу Джарвіса, яка отримує на вхід множину точок а повертає множину точок які входять до мінімальної опуклої оболонки.

```
def graham(A):
    start = time.time()
    n = len(A)
    P = range(n)
    for i in range(1,n):
      if A[P[i]][0]<A[P[0]][0]:
        P[i], P[0] = P[0], P[i]
    for i in range(2,n):
      j = i
      while j>1 and (rotate(A[P[0]],A[P[j-1]],A[P[j]])<0):
        P[j], P[j-1] = P[j-1], P[j]
        j -= 1
    S = [P[0],P[1]]
    for i in range(2,n):
```

```
    while rotate(A[S[-2]],A[S[-1]],A[P[i]])<0:
      del S[-1]
    S.append(P[i])
  S.append(P[0])
  for i in range(len(S)):
    x_final.append(points[int(S[i])][0])
    y_final.append(points[int(S[i])][1])
```

функція «graham» є реалізованим алогритмом Грехема, який як минулий метод отримує множину точок, та повертає множину точок мінімальної опуклої оболонки.

```
def read():
  f=open('points.txt')
  p=f.read()
  p=p.split(",")
  n=int(p[0])/2
  del p[0]
  for i in range(n):
    points.append([])
    for j in range(2):
      points[i].append(int(p[0]))
      del p[0]
  for i in range(len(points)):
        x_list.append(points[i][0])
  for i in range(len(points)):
        y_list.append(points[i][1])
```

функція «read» зчитує координати точок з файлу, оброблює та перетворює в зручний вид.

```
def chan(pts):
```

```
    start = time.time()
    global hull
    global hulls
    global x
    for m in (1 << (1 << t) for t in xrange(len(pts))):
        hulls = [_graham_scan(pts[i:i + m]) for i in xrange(0, len(pts), m)]
        hull = [_min_hull_pt_pair(hulls)]
        for throw_away in xrange(m):
            p = _next_hull_pt_pair(hulls, hull[-1])
            if p == hull[0]:
                return [hulls[h][i] for h, i in hull]
            hull.append(p)
```

функція «» реалізовує алгоритм Чена для побудови мінімальної опуклої оболонки. Використовуючи функції:

```
def _keep_left(hull, r):
    while len(hull) > 1 and turn(hull[-2], hull[-1], r) != TURN_LEFT:
            hull.pop()
    return (not len(hull) or hull[-1] != r) and hull.append(r) or hull

def _graham_scan(points):
    points.sort()
    lh = reduce(_keep_left, points, [])
    uh = reduce(_keep_left, reversed(points), [])
    return lh.extend(uh[i] for i in xrange(1, len(uh) - 1)) or lh

def _rtangent(hull, p):
    l, r = 0, len(hull)
    l_prev = turn(p, hull[0], hull[-1])
    l_next = turn(p, hull[0], hull[(l + 1) % r])
```

```
    while l < r:
        c = (l + r) / 2
        c_prev = turn(p, hull[c], hull[(c - 1) % len(hull)])
        c_next = turn(p, hull[c], hull[(c + 1) % len(hull)])
        c_side = turn(p, hull[l], hull[c])
        if c_prev != TURN_RIGHT and c_next != TURN_RIGHT:
            return c
        elif c_side == TURN_LEFT and (l_next == TURN_RIGHT or
                                      l_prev == l_next) or \
             c_side == TURN_RIGHT and c_prev == TURN_RIGHT:
            r = c              # Tangent touches left chain
        else:
            l = c + 1          # Tangent touches right chain
            l_prev = -c_next    # Switch sides
            l_next = turn(p, hull[l], hull[(l + 1) % len(hull)])
    return l

def _min_hull_pt_pair(hulls):
    h, p = 0, 0
    for i in xrange(len(hulls)):
        j = min(xrange(len(hulls[i])), key=lambda j: hulls[i][j])
        if hulls[i][j] < hulls[h][p]:
            h, p = i, j
    return (h, p)

def _next_hull_pt_pair(hulls, pair):
    p = hulls[pair[0]][pair[1]]
    next = (pair[0], (pair[1] + 1) % len(hulls[pair[0]]))
    for h in (i for i in xrange(len(hulls)) if i != pair[0]):
        s = _rtangent(hulls[h], p)
        q, r = hulls[next[0]][next[1]], hulls[h][s]
```

```
        t = turn(p, q, r)
        if t == TURN_RIGHT or t == TURN_NONE:
            next = (h, s)
    return next
def rand():
    global k1, k2
    n=raw_input("input n= ")
    k1=int(raw_input("input min= "))
    k2=int(raw_input("input max= "))
    f=open('points.txt', 'w')
    f.write(n+',')
    for i in range(int(n)):
        k=random.randint(k1,k2)
        f.write(str(k)+',')
    f.close()
```

функція «rand» використовується для рандомного створення точок, вона запитує у користувача кількість точок, та межі для їх координат.

4.Тестування

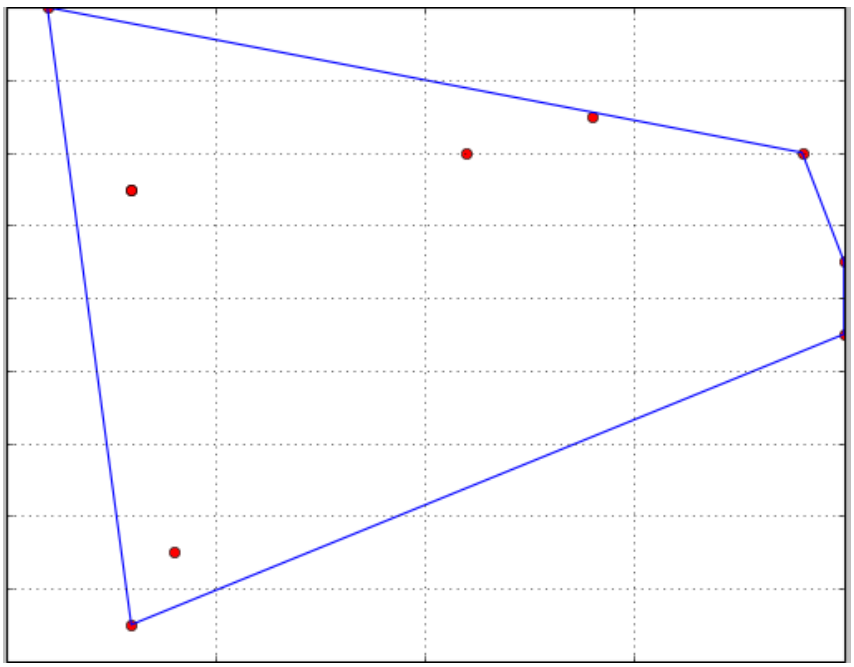
4.1.Використання математичного пакету Maple для побудови мінімальної опуклої оболонки

Для перевірки правильності нашої програми, ми побудуємо мінімальну опуклу оболонку, для однієї множини точок, у нашій програмі та програмному пакеті Maple. Система Maple призначена для символьних обчислень, хоча має ряд засобів і для чисельного розв'язання диференціальних рівнянь і знаходження інтегралів.

Маємо точки (9,6),(-6,-5),(10,1),(-7,5),(4,7),(-7,5),(10,3),(1,6),(-9,10),(-7,-7).

Запустимо программу , отримаємо точки з МОО

```
>>>
[ -9 10 ]
[ -7 -7 ]
[ 10 1 ]
[ 10 3 ]
[ 9 6 ]
[ -9 10 ]
0.0460000038147 second
```



Тепер введемо координати точок в Maple, та знайдемо МОО, за допомогою команд:

```
> with (simplex) :
> convexhull ({[9, 6], [−6,−5], [10, 1], [−7, 5], [4, 7], [−7, 5], [10, 3],
    [1, 6], [−9, 10], [−7,−7]});
```

Отримуємо:

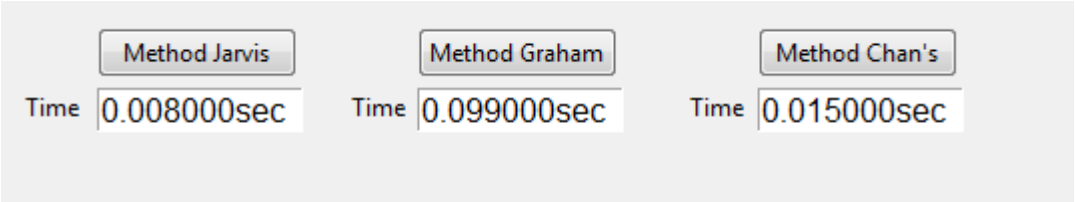
```
[[−9, 10], [−7, −7], [10, 1], [10, 3], [9, 6]]
```

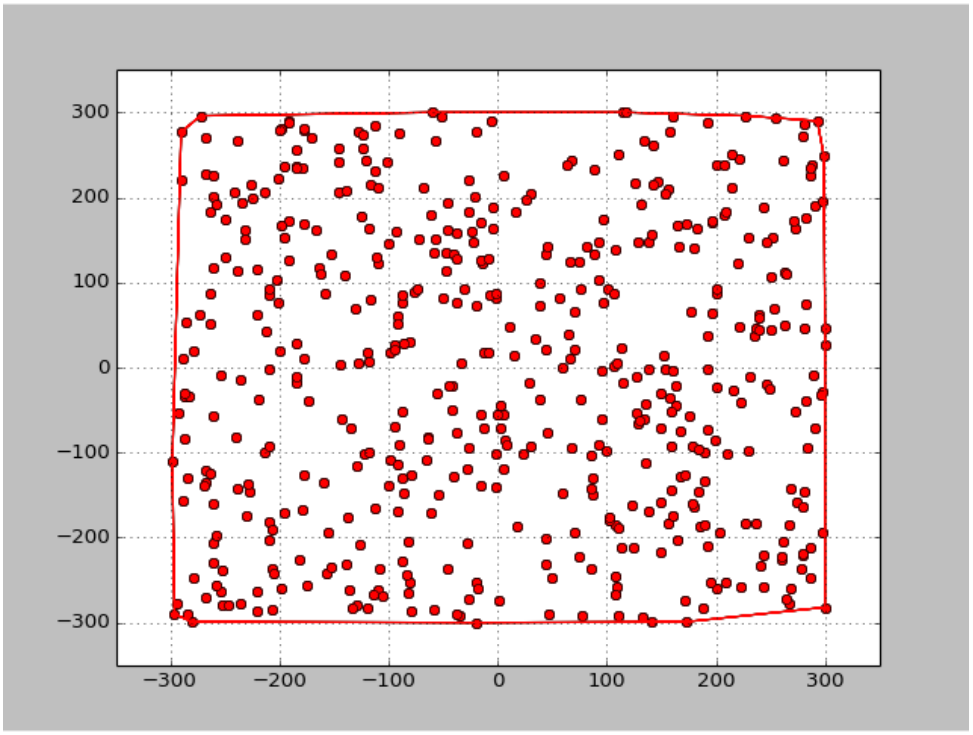
Що сходиться з результатами нашої програми, отже реалізували алгоритм ми правильно.

4.2.Порівняння часової характеристики алгоритмів

Порівння часової характеристики почнемо з 1000 точок,координати в інтервалі $[-300; 300]$.

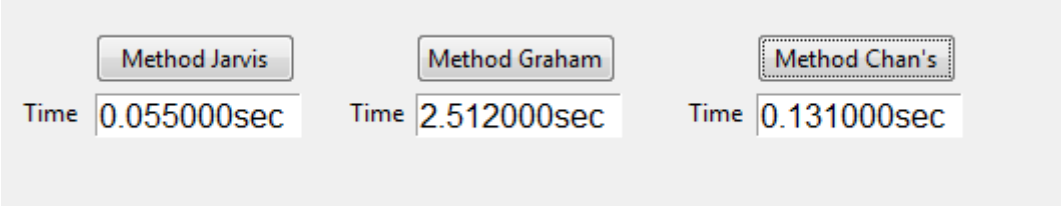
Отримуємо:

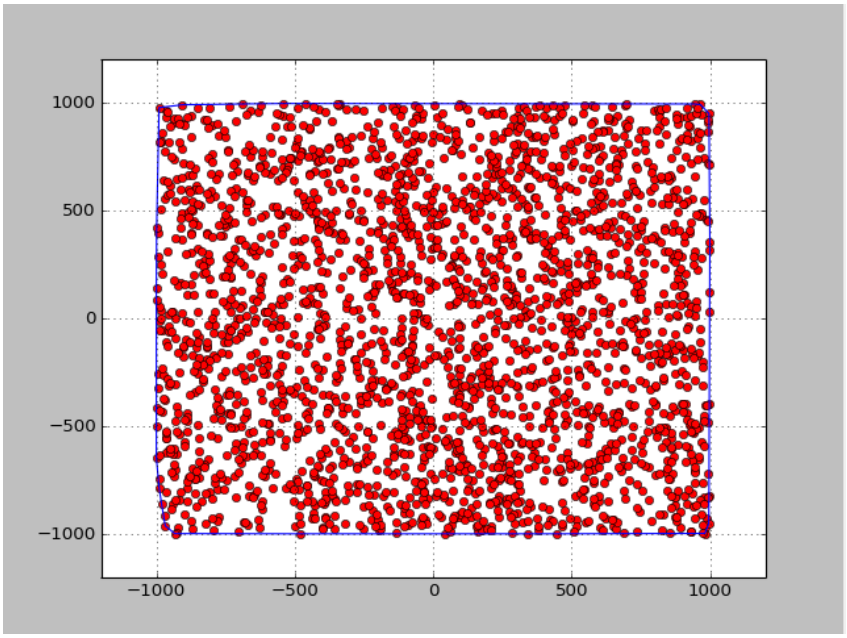




Наступним кроком 5000 точок, в інтервалі $[-1000; 1000]$.

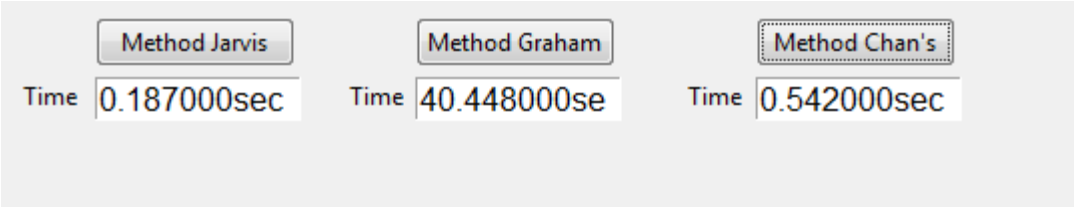
Отримали:

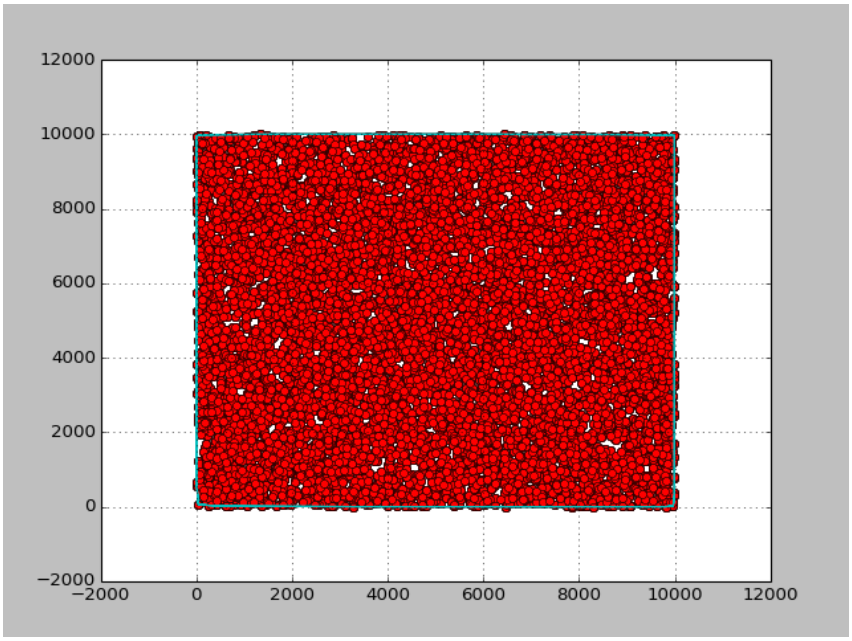




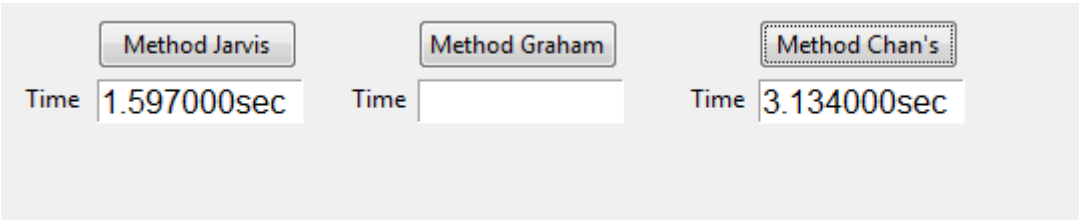
Далі для порівняння часу, візьмемо 20000,координати в інтервалі $[0;10000]$.

Отримали:



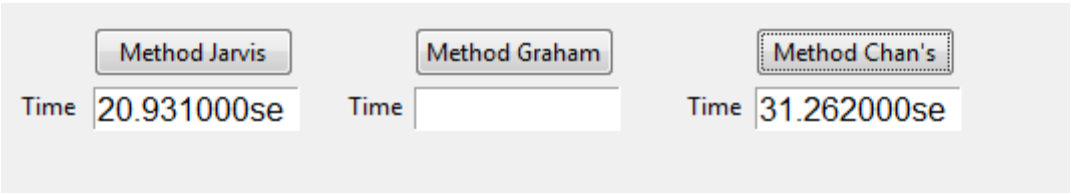


Далі,ми візьмемо 100000 точок. Отримали:



Час роботи методу Грехема 7 хвилин.

І останнім кроком візьмемо 999999 точок, та розглянемо результати.



Час за який працює метод Грехема для 999999 точок, не вдалось знайти.

Отже, найшвидшим є метод Джарвіса та Чена, їхній час приблизно однаковий, метод Грехема повільніший, але в найгіршому випадку, він покаже час на рівні з алгоритмом Джарвіса.

Висновки

У математиці та комп’ютерній науці однією з найпоширеніших проблем є побудова опуклих оболонок. Задача побудови опуклих оболонок та їх аналіз було програмно реалізовано в даній роботі.

Важливість задачі побудови опуклої оболонки полягає не тільки у величезній кількості додатків, де вона застосовується (розпізнавання образів, обробка зображень, бази даних, задачі розкрою та компонування матеріалів, математична статистика), але також і через користь опуклої оболонки як інструменту вирішення безлічі завдань обчислювальної геометрії.

У роботі також було проаналізовано та досліджено основні методи побудови мінімальних опуклих оболонок. Розроблено програму для побудови мінімальної опуклої оболонки. При розробки даної програми використовувалась мова програмування «Python» з доповненням « MatPlotLib».

У розробленій програмі були застосовані наступні алгоритми знаходження мінімальних оболонок: метод Джарвіса, метод Грехема та метод Чена. Для тестування розробленої програми у ній було побудовано мінімальну опуклу оболонку для однієї множини точок та в програмному пакеті Maple.

Внаслідок тестування даної програми було з’ясовано, що, найшвидшим є метод Джарвіса та Чена, їхній час приблизно однаковий, метод Грехема повільніший, але в найгіршому випадку, він покаже час на рівні з алгоритмом Джарвіса.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Ивановский С.А., Преображенский А.С., Симончик С.К. Алгоритмы вычислительной геометрии. Выпуклые оболочки: простые алгоритмы // Компьютерные инструменты в образовании. - СПб.: Изд-во ЦПО "Информатизация образования", 2007, N1, С. 4-19.
2. Фелпс Р. , Лекции о теоремах Шоке,1968.
3. Эдварде Р. Функциональний анализ. Теория и приложения,1969.
4. Anany Levitin. Introduction to the Design and analysis of algorithm,2006.
5. Framerwork definition [Електронний ресурс] – Режим доступу до ресурсу: https: // techterms.com/definition/framerwork
6. Laubach S. and Burdick J. An Autonomous Sensor-Based Path-Planner for Planetary Microrovers. In IEEE International Conference on Robotics and Automation, Detroit MI, 1999.
7. Laubach S. Theory and Experiments in Autonomous Sensor-Based Motion Planning with Applications for Flight Planetary Microrovers. PhD thesis, California Institute of Technology, May 1999.
8. Mobile application [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Mobile_app
9. Mobile app development [Електронний ресурс] – Режим доступу до ресурсу: https://en/wikipedia.org/wiki/Mobile_app_development
10. Patters of development [Електронний ресурс] – Режим доступу до ресурсу: https://web.iit.edu/cac/studend-resources/writing-guides/writing-process/patterns- development-writing
11. React – a JavaScript library [Електронний ресурс] – Режим доступу до ресурсу: https://reactjs.org
12. React Native – a JavaScript library [Електронний ресурс] – Режим доступу до ресурсу: https://facebook.github.io/react-native/
13. Three-dimensional unstructured mesh generation: Part 1. Fundamental aspects of triangulation and point creation. Yao Zheng, Roland W. Lewis, David T.

Gethinb. ELSEVIER Comput. Methods Appl. Mech. Engrg. 134 (1996) 249—268.

14. Types of mobile applications [Електронний ресурс] – Режим доступу до ресурсу: https: // thinkmobile.com/blog/popular-types-of-apps/
15. Zahid Hossain V., Ashraful Amin M. On Constructing Approximate Convex Hull. American Journal of Computational Mathematics, 2013.

Додаток

```
from Tkinter import *
import ttk
import matplotlib.pyplot as plt
import time
import random
class Application:
    def __init__(self, root):
        self.root = root
        self.root.title('Convex Hulls')
        ttk.Frame(self.root, width=540, height=270).pack()
        self.init_widgets()


    def init_widgets(self):
        but1=ttk.Button(self.root, command=self.insert_1, text='Method Jarvis',
width='15').place(x=50, y=150)
        but2=ttk.Button(self.root, command=self.insert_2, text='Method Graham',
width='15').place(x=210, y=150)
        but4=ttk.Button(self.root, command=self.insert_3, text='Method Chan\'s',
width='15').place(x=380, y=150)
        but3=ttk.Button(self.root, command=self.read_points, text='Read Points',
width='15').place(x=50, y=20)
        self.txt1 = Text(self.root, width='11', height='1',font="Arial 12")
        self.txt1.place(x=50, y=180)a
        self.l1 = ttk.Label(text="Time", style="BW.TLabel")
        self.l1.place(x=12, y=180)
        self.l2 = ttk.Label(text="Time", style="BW.TLabel")
        self.l2.place(x=175, y=180)
        self.l3 = ttk.Label(text="Time", style="BW.TLabel")
        self.l3.place(x=344, y=180)
        self.txt2 = Text(self.root, width='11', height='1',font="Arial 12")
        self.txt2.place(x=210, y=180)
```

```
    self.txt3 = Text(self.root, width='11', height='1',font="Arial 12")
    self.txt3.place(x=380, y=180)
    self.txt4 = Text(self.root, width='11', height='1',font="Arial 12")
    self.txt4.place(x=580, y=280)
def read_points(self):
    read()
def insert_1(self):
    try:
        jarvis(points)
        self.txt1.insert(INSERT,str(jarvis_time)+'sec')
        plt.plot(x_list, y_list,'ro')
        plt.plot(x_final,y_final,'-')
        plt.axis([k1-k2/5,k2+k2/5,k1-k2/5,k2+k2/5])
        plt.grid(True)
        plt.show()
    except:
        self.txt3.insert(INSERT,'Error')
def insert_2(self):
    graham(points)
    self.txt2.insert(INSERT,str(graham_time)+'sec')
    plt.plot(x_list, y_list,'ro')
    plt.plot(x_final,y_final,'-')
    plt.axis([k1-k2/7,k2+k2/7,k1-k2/7,k2+k2/7])
    plt.grid(True)
    plt.show()
def insert_3(self):
    final()
    self.txt3.insert(INSERT,str(chan_time)+'sec')
    plt.plot(x_list, y_list,'ro')
    plt.plot(x_final,y_final,'-')
    plt.axis([k1-k2/6,k2+k2/6,k1-k2/6,k2+k2/6])
```

```
        plt.grid(True)
        plt.show()
points=[]
x_final=[]
y_final=[]
x_list=[]
y_list=[]
def jarvis(A):
    start = time.time()
    n = len(A)
    P = range(n)
    for i in range(1,n):
      if A[P[i]][0]<A[P[0]][0]:
        P[i], P[0] = P[0], P[i]
    H=[P[0]]
    del P[0]
    P.append(H[0])
    while True:
        right=0
        for i in range(1,len(P)):
            if rotate(A[H[-1]],A[P[right]],A[P[i]])<0:
                right=i
        if P[right]==H[0]:
            break
        else:
            H.append(P[right])
            del P[right]
    H.append(H[0])
    x=(time.time() - start)
    global jarvis_time
    jarvis_time= "%.6f" % (x)
```

```
    for i in range(len(H)):
        x_final.append(points[int(H[i])][0])
        y_final.append(points[int(H[i])][1])
def rotate(a,b,c):
  return (b[0]-a[0])*(c[1]-b[1])-(b[1]-a[1])*(c[0]-b[0])
def read():
    f=open('points.txt')
    p=f.read()
    p=p.split(",")
    n=int(p[0])/2
    del p[0]
    for i in range(n):
      points.append([])
      for j in range(2):
        points[i].append(int(p[0]))
        del p[0]
    for i in range(len(points)):
            x_list.append(points[i][0])
    for i in range(len(points)):
            y_list.append(points[i][1])
def graham(A):
    start = time.time()
    n = len(A)
    P = range(n)
    for i in range(1,n):
      if A[P[i]][0]<A[P[0]][0]:
        P[i], P[0] = P[0], P[i]
    for i in range(2,n):
      j = i
      while j>1 and (rotate(A[P[0]],A[P[j-1]],A[P[j]])<0):
        P[j], P[j-1] = P[j-1], P[j]
```

```
        j -= 1
    S = [P[0],P[1]]
    for i in range(2,n):
      while rotate(A[S[-2]],A[S[-1]],A[P[i]])<0:
        del S[-1]
      S.append(P[i])
    S.append(P[0])
    for i in range(len(S)):
        x_final.append(points[int(S[i])][0])
        y_final.append(points[int(S[i])][1])
    x=(time.time() - start)
    global graham_time
    graham_time= "%.6f" % (x)
#---------------------------------------------------------------------------
TURN_LEFT, TURN_RIGHT, TURN_NONE = (1, -1, 0)
def turn(p, q, r):
    return cmp((q[0] - p[0])*(r[1] - p[1]) - (r[0] - p[0])*(q[1] - p[1]), 0)
def _keep_left(hull, r):
    while len(hull) > 1 and turn(hull[-2], hull[-1], r) != TURN_LEFT:
            hull.pop()
    return (not len(hull) or hull[-1] != r) and hull.append(r) or hull
def _graham_scan(points):
    points.sort()
    lh = reduce(_keep_left, points, [])
    uh = reduce(_keep_left, reversed(points), [])
    return lh.extend(uh[i] for i in xrange(1, len(uh) - 1)) or lh
def _rtangent(hull, p):
    l, r = 0, len(hull)
    l_prev = turn(p, hull[0], hull[-1])
    l_next = turn(p, hull[0], hull[(l + 1) % r])
    while l < r:
```

```
        c = (l + r) / 2
        c_prev = turn(p, hull[c], hull[(c - 1) % len(hull)])
        c_next = turn(p, hull[c], hull[(c + 1) % len(hull)])
        c_side = turn(p, hull[l], hull[c])
        if c_prev != TURN_RIGHT and c_next != TURN_RIGHT:
            return c
        elif c_side == TURN_LEFT and (l_next == TURN_RIGHT or
                                      l_prev == l_next) or \
             c_side == TURN_RIGHT and c_prev == TURN_RIGHT:
            r = c              # Tangent touches left chain
        else:
            l = c + 1          # Tangent touches right chain
            l_prev = -c_next    # Switch sides
            l_next = turn(p, hull[l], hull[(l + 1) % len(hull)])
    return l
def _min_hull_pt_pair(hulls):
    h, p = 0, 0
    for i in xrange(len(hulls)):
        j = min(xrange(len(hulls[i])), key=lambda j: hulls[i][j])
        if hulls[i][j] < hulls[h][p]:
            h, p = i, j
    return (h, p)
def _next_hull_pt_pair(hulls, pair):
    p = hulls[pair[0]][pair[1]]
    next = (pair[0], (pair[1] + 1) % len(hulls[pair[0]]))
    for h in (i for i in xrange(len(hulls)) if i != pair[0]):
        s = _rtangent(hulls[h], p)
        q, r = hulls[next[0]][next[1]], hulls[h][s]
        t = turn(p, q, r)
        if t == TURN_RIGHT or t == TURN_NONE:
            next = (h, s)
```

```
    return next


def chan(pts):
    start = time.time()
    global hull
    global hulls
    global x
    for m in (1 << (1 << t) for t in xrange(len(pts))):
        hulls = [_graham_scan(pts[i:i + m]) for i in xrange(0, len(pts), m)]
        hull = [_min_hull_pt_pair(hulls)]
        for throw_away in xrange(m):
            p = _next_hull_pt_pair(hulls, hull[-1])
            if p == hull[0]:
                return [hulls[h][i] for h, i in hull]
            hull.append(p)
        x=(time.time() - start)
        global chan_time
        chan_time= "%.6f" % (x)
def final():
    chan(points)
    for i in range(len(hull)):
        x_final.append(int(hulls[hull[i][0]][hull[i][1]][0]))
        y_final.append(int(hulls[hull[i][0]][hull[i][1]][1]))
    x_final.append(x_final[0])
    y_final.append(y_final[0])
def rand():
    global k1, k2
    n=raw_input("input n= ")
    k1=int(raw_input("input min= "))
    k2=int(raw_input("input max= "))
    f=open('points.txt', 'w')
```

```
    f.write(n+',')
    for i in range(int(n)):
        k=random.randint(k1,k2)
        f.write(str(k)+',')
    f.close()
#-------------------------------------------------------------------------------
if __name__ == '__main__':
    rand()
    root = Tk()
    Application(root)
    root.mainloop()
```