Convex Hull Report

CS 4412 Advanced Algorithms

Tanner Webb

Introduction:

This report is about a project dealing with the convex hull. A convex hull is a shape that takes place by mapping out a population of points and discovering the outside edges. In this way think about those connect the dot puzzles you would fill out when you were a kid. Sounds easy right? Unfortunately, this process is much harder than what it seems for you need to create an algorithm to tell a computer how to find these points and map all the outside edges. This project we needed to implement this into a program using python which tells a computer how to map the convex hull based on a x and y coordinated of points and discovering their slope from one another.

For this project we were given a random generator program to place points on a x and y plane. Based on the number of points plotted, we need to think about the most useful algorithm to apply to the convex hull to get the best time complexity. The divide and conquer algorithm are very useful when trying to tackle the convex hull. The reason being is that why not cut those points in half and speed up the process. This allows all the outer points to be discovered now in two hulls instead of one. All you need to do next is fund the upper and lower tangents of each divided hull and merge them.

How the project went:

I got to realizing that I was trying to implement the divide and conquer algorithm to find the convex hull. However, I started on the path to doing a divide and conquer on my project but ended up on the path of a Jarvis method without realizing it. I went back through and tried to make it to figure out the upper and lower tangents but got stuck one a few different ideas on how to code this up. I thought maybe I could determine the upper and lower tangent points by slope, but I learned that that is very inaccurate for the convex hull points can take any form and directions.

I did end up doing Jarvis method, which was still extremely hard. I needed to implement a function that would calculate based on slopes from individual points inside the hulls. I got most of it typed up and was going to compare to Jarvis to divide and conquer but never got it finished. I thought that maybe at the end of the project descriptions it said here are some other methods we can implement so I thought maybe it would be okay to use Jarvis and then try to implement divide and conquer. I do feel like there will be a difference between the two because I researched on their time complexities. Divide and conquer uses O(n log n) while Jarvis uses O(nh) leading to Jarvis being the more accurate.

The difference between the two was difficult, because the algorithm I implemented has a mixture of both and is designed to find the upper and lower tangents automatically and join them. For example, when I found the outside points for my left and right hulls (the outside points) I merged those together and ran it back though the algorithm to find the new upper and lower tangents. I merged them which is what you need to do for divide and conquer. Then ran it through my algorithm I created again, and it combined it both hulls together and linked the upper and lower tangents of each hull to complete the big convex hull.

I wanted to include this in my report for any questions on why I did things in my program and where I might of went wrong. I thought what I did looked good and it searches extremely fast. Faster then any algorithm I previously tested on this project.

Analysis of observations and differences seen:

I have noticed the algorithm does well at finding the convex hull. There are many algorithms that claim to be faster than the divide and conquer or vise vera. I have noticed though when getting to a high number of complexity of points such as 1000000, they take about equal amount of time. I talked with other kids in the class and they all got about the same time and we all coded this project differently. This can be true for we can see that the that the worst time the divide and conquer algorithm can do is possible O(n log n) which is also its best complexity. Versus the Jarvis we can see the best complexity is O(nh) and the worst is O(n^2) which is bad. I think this is because it takes more time to calculate the triplets and it needs to walk through each point and check its origins as well as direction.

Raw and mean of experimental outcomes, plots, and patterns:

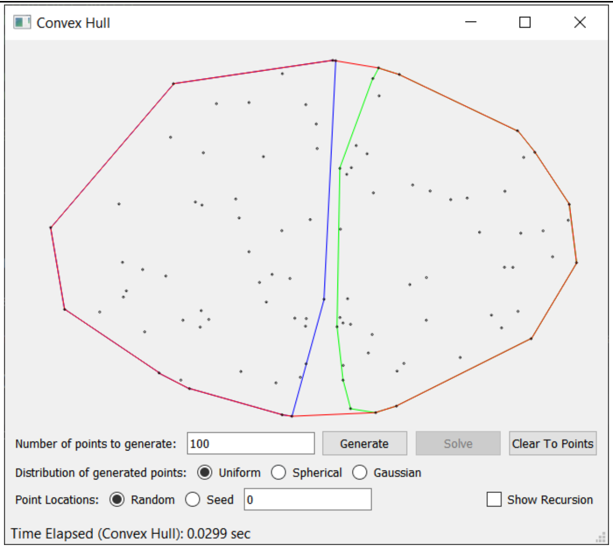
I tested many numbers of points ranging anywhere from 1 to 1000000. However, its difficult to say that my program kept erring on lower numbers rather then higher. I do believe this is due to the calculation process of so many angles and how the slopes can make it hard to predict the upper and lower tangents of the hull. For example, if I choose 10 points and I divide by half, ill get 5 points for each hull I am calculating. Sometimes it would not recognize the upper tangent for the points are either scattered or plotted away from the other group of points. However, I did get this fixed, but it took extra time to calculate why this was happening. The higher the points did take longer but I did not get as many errors and followed a more accurate path, not scrambled.

The other thing I did notice was that we could also shape how the points were distributed, Such as uniform, spherical, and gaussian. My algorithm likes gaussian better for it had faster times and it had more precise points gathered. I think this has to do with the time it takes to calculate the distance between each point. Versus if the points are spread out like in uniform and spherical, they are spread out and can be in weird direction when trying to calculate if an outside point. Perhaps a thing to think about when looking into this is time complexity because if the points.

are farther apart out time complexity will increase.

Gaussian	Round 1	Round 2	Round 3	Round 4	Round 5
10	0.0052 sec	0.0056 sec	0.0070 sec	0.0060 sec	0.0072 sec
100	0.0124 sec	0.0060 sec	0.0129 sec	0.0110 sec	0.0060 sec
1000	0.0519 sec	0.0549 sec	0.0409 sec	0.0608 sec	0.0618 sec
10000	0.5585 sec	0.5774 sec	0.5256 sec	0.7659 sec	0.5555 sec

Screenshots of fully working examples:



Number of points to generate: 100

Distribution of generated points: [✓ Uniform] Spherical Gaussian

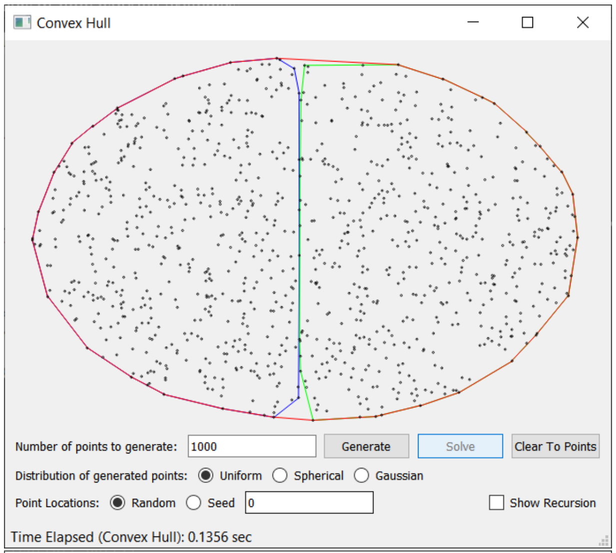
Point Locations: [✓ Random] Seed 0

☐ Show Recursion

Time Elapsed (Convex Hull): 0.0299 sec

Picture 1.1

Shows the convex hull of 100 uniform points.



Number of points to generate: 1000

Distribution of generated points: [✓ Uniform] Spherical Gaussian

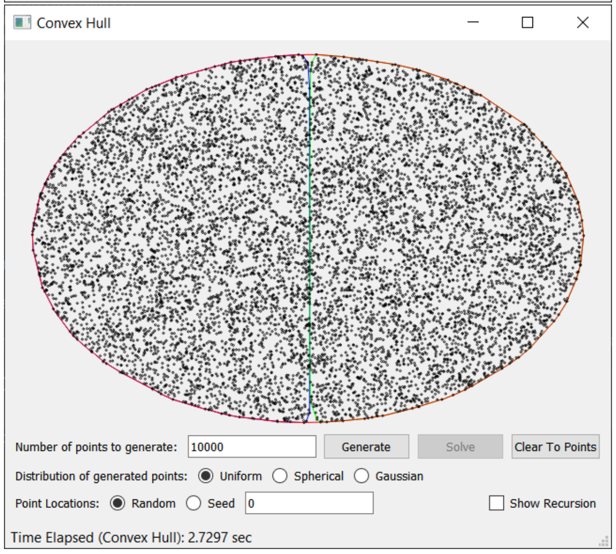
Point Locations: [✓ Random] Seed 0

☐ Show Recursion

Time Elapsed (Convex Hull): 0.1356 sec

Picture 1.2

Shows the convex hull of 1000 uniform points.



Number of points to generate: 10000

Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0

☐ Show Recursion

Time Elapsed (Convex Hull): 2.7297 sec

Picture 1.3

Shows the convex hull of 10000 uniform points.

Pseudocode for divide and conquer for convex hull:

1. Sorting and Dividing O (n log n)
   a. We sort the Q points by x coordinates.
   b. We now divide the points into two sets O(n)

Steps 3 and 4 are I will follow the Jarvis March Algorithm AKA wrapping algorithm O(nh)

2. Find direction *repeated for each left and right hull*
   a. We want to plan on focusing on a counterclockwise direction for our paths.
   b. Pass in three points into a function (origin, pt1, pt2)
   c. Use the determine the slope and combined slopes to calculate a return value.
   d. Sent number to function that can show if that origin is in the convex hull.

3. Add Convex Hull Points *while for each left and right hull*
   a. Retrieve number from orientation, only -1 matters for counterclockwise.
   b. Step 3 above
   c. If -1 append that point into an array to be sorted.
   d. If 0 end process for no more to be tested
   e. Return the appended points that are sorted based on counterclockwise positions.

4. Plot the Left and Right hulls separately and connect lines.
   a. For loop to work through all points
   b. Blue for Left_hull and Green for Right_hull

5. Merge the Steps retrieved from step 4. O(n)
   a. Retrieve convex hull values for left_hull
   b. Retrieve convex hull values for right_hull
   c. Merge these two sets to get a full_convexhull

6. Send full_convexhull int Jarvis March Algorithm
   a. Will go through the process in steps 3 and 4.
   b. We will retrieve the return value using Calc_hull to store all points it found from the algorithm.
   c. These points to include the upper and lower tangents to connect the convex hull.
   d. Step 5 but this time only convex hull will be sent through with a red line.

Total Time and Space Complexity:

Divide and Conquer = O (n log n)

Jarvis Algorithm = O (nh) * 3 (Each set we test and run through the Jarvis algorithm)

Merging = O (n)

Space complexity = O (n)

Divide and Conquer time complexity break down:

1. Sorting point by x value O (n log n)
2. Dividing into two sets O (1)
3. Calculate the complex hull of left O(n/2)
4. Calculate the complex hull of right O(n/2)
5. Merge into one convex hull O(n) ← Which I thought is what I did since have all the points.
6. ** possibly what I might of done additionally calculate merged hull O(n/2)**

I have questions about this, is what I implemented faster then trying to map all the points to the upper and lower tangents of each hull? I would think so because it can take what it already calculated and calculate it even faster now.

Articles:

https://stackoverflow.com/questions/7131372/count-number-of-triples-in-an-array-that-are-collinear

https://personal.utdallas.edu/~daescu/convexhull.pdf

Convex Hull Source Code:

```
from idlelib import window

import PyQt5
import numpy as np
import points as points
import math

import start as start
from Tools.scripts.pindent import start

from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF, QObject
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time

# Some global color constants that might be useful
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Global variable that controls the speed of the recursion automation, in seconds
#
PAUSE = 0.25



#
# This is the class you have to complete.
#
class ConvexHullSolver(QObject):

    # Class constructor
    def __init__(self):
        super().__init__()
        self.pause = False

    # Some helper methods that make calls to the GUI, allowing us to send updates
    # to be displayed.

    def showTangent(self, line, color):
        self.view.addLines(line, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseTangent(self, line):
        self.view.clearLines(line)

    def blinkTangent(self, line, color):
        self.showTangent(line, color)
        self.eraseTangent(line)

    def showHull(self, polygon, color):
        self.view.addLines(polygon, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseHull(self, polygon):
        self.view.clearLines(polygon)

    def showText(self, text):
        self.view.displayStatusText(text)

    def set_points(self, unsorted_points, demo):
        self.points = unsorted_points
        self.demo = demo

    def get_direction(self, origin, point1, point2):
        # What does this function does:

        # For example - origin -> point1 -> point2
        # origin -> point1  =  a = (y2 - y1) / (x2-x1) = slope
        # point1 -> point2  =  b = (y3 - y2) / (x3-x2) = slope
        # https://stackoverflow.com/questions/7131372/count-number-of-triples-in-an-array-that-are-
collinear

        upper = (point1.y() - origin.y()) * (point2.x() - point1.x())
        lower = (point1.x() - origin.x()) * (point2.y() - point1.y())

        # Now this will show if positive or negative or collinear

        orientation = upper - lower
        # This will say clockwise direction
        if orientation > 0:
            return 1

        # This will determine it points are collinear
        elif orientation == 0:
            return 0

        # This will say counter clockwise position
        else:
            return -1

    def get_points(self, sorted_points, point_size):
        hull_points = []  # this will hold all of our outside points
        left_point = 0  # We know our left most point already(sort_points[0])
        q = 0
        while True:

            hull_points.append(sorted_points[left_point])  # We need to add the left most point
            far_point = (left_point + 1) % point_size  # Helps determine if all the points have been checked,
will end with remainder of 1.
            # We will do a for loop for all points inside that array
            for i in range(point_size):

                # This article gives a good example on what this does.
                # https://personal.utdallas.edu/~daescu/convexhull.pdf Thought this article was good.
                # We just want the number because we already know our points are sorted by x value.
                # Since we are in a while loop we need to append into the list anyways line 107
                if self.get_direction(sorted_points[left_point], sorted_points[i], sorted_points[far_point]) == -1:
                    far_point = i

            left_point = far_point  # Next point and repeat the process

            if left_point == 0:
                break
        return hull_points  # Follows onto lines 147, 152, and 162

    # Will calculate and show the lines in the graph of points that have been found
    def left_hull(self, left_points):
        left_hull = self.get_points(left_points, len(left_points))
        for i in range(len(left_hull)-1):
            polygon = [QLineF(left_hull[i], left_hull[i + 1])]
            self.showHull(polygon, BLUE)
        polygon = [QLineF(left_hull[0], left_hull[len(left_hull) - 1])]
        self.showHull(polygon, BLUE)
        return left_hull
```

```
# Will calculate and show the lines in the graph of points that have been found
def right_hull(self, right_points):
    right_hull = self.get_points(right_points, len(right_points))
    for i in range(len(right_hull) - 1):
        polygon = [QLineF(right_hull[i], right_hull[i + 1])]
        self.showHull(polygon, GREEN)
    polygon = [QLineF(right_hull[0], right_hull[len(right_hull) - 1])]
    self.showHull(polygon, GREEN)
    return right_hull

# Will calculate and show the lines in the graph of points that have been found
def full_hull(self, merged_points):
    calc_hull = self.get_points(merged_points, len(merged_points))
    for i in range(len(calc_hull) - 1):
        polygon = [QLineF(calc_hull[i], calc_hull[i + 1])]
        self.showHull(polygon, RED)
    polygon = [QLineF(calc_hull[0], calc_hull[len(calc_hull) - 1])]
    self.showHull(polygon, RED)
    return calc_hull

# Just a random function that I had a idea for to get rid of the lines byt calculating the pints in each hull
# then those that arent don't draw a line to. This was going to be used to delete lines later on
'''def compare_points(self, right_hull,left_hull, full_hull):
    fix_left = []
    fix_right = []
    for i in range(len(left_hull)):
        for j in range(len(full_hull)):
            if left_hull[i] == full_hull[j]:
                fix_left.append(left_hull[i])

    for i in range(len(right_hull)):
        for j in range(len(full_hull)):
            if right_hull[i] == full_hull[j]:
                fix_right.append(right_hull[i])'''

# This is the method that gets called by the GUI and actually executes
# the finding of the hull
def compute_hull(self, points, pause, view):
    self.pause = pause
    self.view = view
    self.points = points

    assert (type(points) == list and type(points[0]) == QPointF)
    # TODO SORT POINTS BY X VALUES
    sorted_points = sorted(points, key=lambda pt: pt.x())  # We need to SORT our points by x values

    if len(points) <= 5:
            calc_hull = self.get_points(sorted_points, len(sorted_points))
            self.full_hull(calc_hull)
            return

        points_size = len(sorted_points)  # Not needed but convenient to call

        # The code below will divide the array by half leading to a divide and conquer
        # We will call the two left_points and right_points
        mid = len(sorted_points) // 2
        left_points = sorted_points[0:mid]
        right_points = sorted_points[mid:]

        # This below will get the left side and right side of the convex hull that we divided
        # Time is used to see how fast it measured calculating points
        t1 = time.time()
        left_hull = self.left_hull(left_points)
        right_hull = self.right_hull(right_points)
        t2 = time.time()
        self.showText('Time Elapsed (Convex Hull): {:3.4f} sec'.format(t2 - t1))

        # This below will get the left side and right side  points we calculated to be the outside and merge
them.
        # Time is used to see how fast it measured calculating points
        merged_points = left_hull + right_hull
        t3 = time.time()
        calc_hull = self.full_hull(merged_points)
        t4 = time.time()
        self.compare_points(right_hull, left_hull, calc_hull)


        # when passing lines to the display, pass a list of QLineF objects.  Each QLineF
        # object can be created with two QPointF objects corresponding to the endpoints
        self.showText('Time Elapsed (Convex Hull): {:3.4f} sec'.format(t4 - t3))
```