Extended essay in Computer Science

Comparison the efficiency of different algorithms of the 2 dimensional convex hull

Word count: 3687

Abstract

During my preparation for Olympiad in informatics, I have come across an algorithmic problem where I was supposed to find the convex hull. This brought me to search for other algorithms with better time efficiency that might be implemented to figure out this problem. Hence, I decided to make extended essay in Computer Science and set it up on **comparison of the efficiency of different algorithms of the 2 dimensional convex hull.**

The application used for testing was coded in C++. The script generated points with random coordinates in a given range for a given input.

The time efficiency of each of the algorithm was measured by the computation time and number of operation calls.

The tested algorithms are brute force, Jarvis and Graham scan algorithm. Each algorithm found the convex hull of the points properly, although the time efficiency was dissimilar.

For the higher number of points given in the input the Brute force algorithm worked rather inefficiently and took significantly more time than the other two algorithms. When the number of vertices of the hull was relatively low, the Jarvis algorithm worked more efficiently than the Graham Scan algorithm. Nevertheless, as the number of the vertices increased and reached approximately 1% of the overall number of points, the Graham Scan algorithm worked more efficiently.

Beyond that, these algorithms are widely used in practice and can be implemented on points in two dimensional space. Whether one wants to build a fence for sheep or find

out how a robot can avoid an obstacle, this work might be useful. There is a number of improvements that can be made – comparing other algorithms that were not mentioned, increase size of the input or change the structure of the points.

Word count: 288

Table of contents

1 Introduction

During the preparation for this years' Olympiad in informatics, I have come across some of the problems from the previous years. It was this time that I approached the problem of the convex hull. My first solution was rather time exhausting and did not receive enough points. This made me not only further think about this problem, but also conduct some research on it.

I found a number of real life applications of the convex hull. The most interesting seemed to me that robots use the convex hull algorithms to find the shortest path when they try to avoid an obstacle. Such calculations are also used by the famous rover Curiosity (Fudan, 2007).

Furthermore, in computer games and computer visualization it is often necessary to find out if some objects intersect. Since this is a rather complicated process, the objects are firstly replaced by bounding boxes. Convex hull is then used to determine whether there are bounding boxes that do not intersect. If the bounding boxes do not intersect, neither will the objects. Then all these objects will be excluded and the process of finding which objects intersect will be shortened.

Another important use of the convex hull could be in evacuation. In a city where censors are evenly distributed, in a case of a disaster, all people who are in the area of the convex hull could be evacuated (Rufai, 2015).

The applicability of the convex hull in real life in many different areas combined with my will to make the problem more time efficient carried me to do my extended essay based on this problem. Hence, my topic is: Comparison the efficiency of different

algorithms of the 2 dimensional convex hull

This extended essay will be focused on 3 different algorithms. Firstly, the algorithms will be theoretically explained and their time complexity will be estimated. From the data gathered I will afterwards compare the estimated time complexity with the real time complexity.

The algorithms will be tested on randomly generated points in a given range for different sized inputs. To decrease uncertainty of the measurement, execution time and number of operation calls for every algorithm for a given size of the input will be calculated as the average of 100 different measurements.

The algorithms will be coded in C++ since I am already familiar with this language from previous algorithmic challenges.

2 Convex Hull

The convex hull is the smallest convex set of points in which a given set of points is contained (Chan, 1996). The word convex specifies that none of the interior angles in the hull can be equal to or greater than 180. When a rubber is stretched around the points on the convex hull so the whole set of points is included in the convex hull, then there will be tension in rubber on all points of the hull (Muhammad, 2010). In the figure below, an example of a convex hull of a set of points is illustrated.

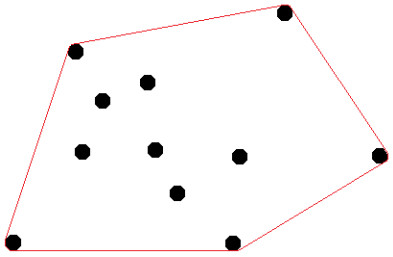

Figure 1. An example of a convex hull of a set of points

2.1 Time complexity

Time complexity of an algorithm signifies the total time required by the program to run to completion (studytonight.com, 2016). It is a function of the input size that indicates how much time the program will need. It is often estimated by counting the number of elementary functions performed during the algorithm.

Asymptotic time complexity of an algorithm represents how quickly the time complexity of an algorithm will rise or decrease depending on the input. It is usually notated by big O (Canelake, 2011).

Imagine an example when the time complexity of an algorithm is as in the equation below.

$$T(n) = n^3 + 2n^2 + 5n + 10$$

Equation 1. Time complexity of an algorithm

Since the *n* with the highest exponent will rise much faster than other elements, the other elements are negligible. Therefore, the asymptotic time complexity of the same algorithm will be as followed:

$$\boldsymbol{O(n) = n^3}$$

Equation 2. Asymptotic time complexity of the same algorithm

2.2 Mathematical background

2.2.1 The shortest distance between two points

The shortest distance between two points in Euclidean space can be calculated through the Pythagoras's theorem. The distance between two points in 2 dimensional Euclidian plane can be calculated as follows:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Equation 3. Calculation of the shortest distance between two points in Euclidean space

2.2.2 Orientation of a triplet of points

Different mathematical approaches can be used to find the orientation of a triplet of points. In our case, orientation of a triplet will be calculated by comparing the gradients. In order to find the orientation of a triplet consisting of points (P_1, P_2, P_3) the gradients of lines connecting points (P_1, P_2) and points (P_2, P_3) are calculated. If the gradient of the second line is higher than of the first line, then the triplet is counter-clockwise. Such a case is illustrated in figure 2.

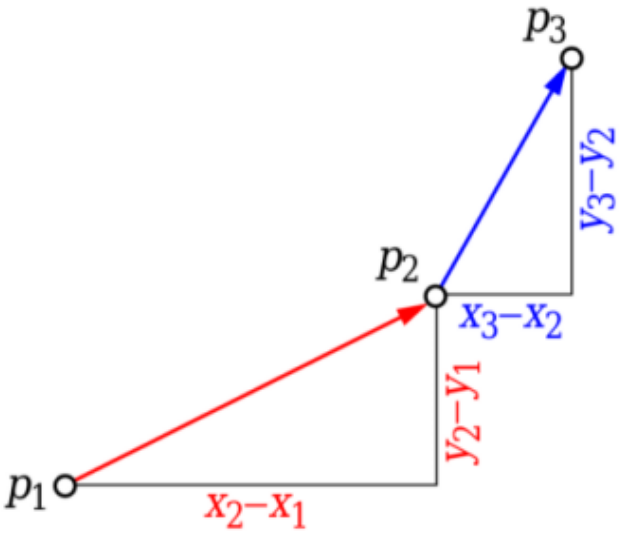

Figure 2[1]. The orientation of these three points is counter-clockwise as the gradient between p_1 and p_2 is lower than in the following two points

When the two gradients are the same, the points are collinear. The last option is that the first gradient is higher than the second gradient. In such a situation, the triplet is rotated clockwise.

2.2.3 Sorting points by the polar angle

Sorting points by the polar angle means sorting points by their angle with respect to the point with the lowest y-coordinate. In case there are more points with the lowest y-coordinate, then the left-most is chosen. The points will be sorted in an ascending way and a function of the coding language will be used for this purpose. In a sorted set of points, a point A will be placed before point B if the point A has lower angle with respect to the bottom-most point. This is shown in the figure 3 below.

[1] http://www.geeksforgeeks.org/orientation-3-ordered-points/

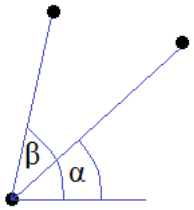

Figure 3. Two points sorted by their polar angle with respect to the bottom most point

Despite this technique of sorting is not going to be used in the algorithms, it represents an important aspect in realizing how the graham scan algorithm works.

2.2.4 Sorting by the polar angle using the trigonometric functions

Trigonometric functions are monotonic which means they are always decreasing or increasing. Since all of the points are located in the first two quadrants with respect to the point with the lowest y-coordinate, a trigonometric function that is always increasing in these two quadrants can be used to sort the points (Koochooloo & maybeWeCouldStalAVan, 2013).

The minus value of cotangent function of the points will provide us with the same results as sorting the points by their polar angles. The minus sign is used in order for the function to increase as the polar angle increases. This method is quicker and therefore will be more appropriate as a calculation for sorting the points by the polar angle.

The only flaw of this method is that it is necessary to separate such points where the y-coordinate is the same, since it is impossible to divide by 0.

2.3 Convex hull algorithms

There is a number of algorithms to determine the convex hull. The first that came to my mind was brute force. After I have done some research on the topic, I found out many

algorithms that are much more time efficient.

2.3.1 Brute force algorithm

Brute force algorithm was the first that came to my mind when I approached this problem. The algorithm begins at the start point. The main idea of the algorithm is to flip through all of the points of the set and connect each two points. The two points are connected by an imaginary line. For the given two points it is necessary to find out if all remaining points of the set are located at one side of the line.

When all points are placed at one side of the line that connects the two given points, then these points are part of the hull. However, only the first one is added to the hull and the algorithm continues with next iteration, since the second point of the hull will be selected again.

This algorithm seems to be rather time-inefficient since it is necessary to connect all two points in the set and then for each connection find out if other points are on one side. This requires three loops in one another and all of the loops are dependent on the size of the set of points. Hence, when the given number of set of points is n, then there are three loops that will be iterated through n times. The forecasted time-complexity of the algorithm is, therefore, n^3.

2.3.2 Jarvis algorithm

This algorithm was discovered independently by Chand & Kapur in 1970 and three years later, in 1973 by R. A. Jarvis (Gift wrapping algorithm, no date). It starts with the left-most point and continues at the circumference of the set of points until the left-most point is reached again. The left-most point is certainly a part of the hull since it is located

at the edge of the set.

Once the left-most point is reached, the algorithm takes another, independent point and determines whether the remaining points are located on one side of an imaginary line that connects these two points. Since the independent point is simply the next point taken from the input, it is unlikely to be one of the points of the hull.

A triplet consisting of the left-most point (a), currently iterated point (b) and the independent point (c) is selected. If such triplet is counter-clockwise, then there is already at least one point that is on the other side of the imaginary line than should be. This is illustrated in Fig 4.

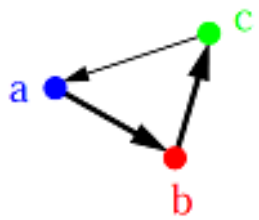

Figure 3[2]. Triplet of counter-clockwise points

Hence, the algorithm replaces the independent (c) point by the current point (b) and further continues iterating. When the algorithm stops iterating, the independent point is the most counter-clockwise point with respect to the left-most point. This point is therefore added to the hull and continues with replacing the left-most point by the point currently added to the hull. Once the left-most point is reached again, the algorithm stops (Cormen, Leiserson, Rivest, Stein, 2008).

As the algorithm goes at the circumference of the points, the estimated time-complexity of this algorithm is O(h*n) where *n* is the number of points of the set and *h* is the number

[2] http://www.geeksforgeeks.org/orientation-3-ordered-points/

of points of the hull (Bradley, 2016).

It seems to be very time-efficient when the number of points of the hull is low. In such case, the algorithm should converge to the time-complexity O(n).

Nevertheless, when all of the points of the set are on the hull, then the algorithm should have time-complexity $O(n^2)$ that ought to be the worst case scenario of the algorithm. This will be tested and further evaluated later in this extended essay.

2.3.3 Graham scan algorithm

This algorithm is named after the Ronald Graham who published it in 1972 (Graham scan, no date). The algorithm begins by searching for point with the lowest y-coordinate. If there are more points with the same lowest y-coordinate, then the point with the lowest x-coordinate is selected. If such point is not the first point in the input, then the point is swapped with the first point. This process requires time complexity O(n) since every point in the data structure needs to be accessed once.

The remaining points are then sorted in a counter-clockwise direction by their polar angle using the trigonometric functions as described in 2.2.4 with respect to the bottom-most point. Time complexity of this process depends on the type of sorting. In our case, the points will be sorted using the quick sort, whose average time complexity is O(n*log n) (Cormen & Balkcom, 2014). The worst case scenario of this sorting algorithm is $O(n^2)$. Nevertheless, it is improbable that the algorithm would have such a time complexity since the points are randomly generated. There might be more points with the same polar angle which means that these points are collinear. In such a situation, only the point that is the farthest from the bottom-most point is considered.

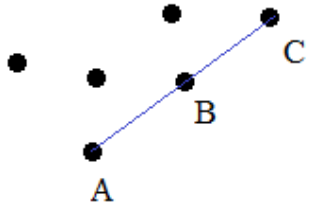

Figure 4. When B and C lay on the same line, only C will be considered as its distance from the point A is greater than in case of point B

The first three points are automatically added to the hull.

Once the array has been sorted by the polar angle, the algorithm starts with the fourth point and continues through the rest of the sorted array. Every time a left turn is made between two consecutive points, the next point is added to the convex hull. If a right turn is needed to be made to get to the next point, then the current point is removed from the convex hull. The algorithm then traces back until the left turn can be made again (C. & Simonson, 2012). Since every point can be visited at most twice, the complexity of this process is O(n).

The overall time complexity of this algorithm is estimated to be O(n*log n) as sorting of the points by polar angle requires most of the time and other processes made in the algorithm are negligible.

3 Testing

3.1 Research plan

The time-efficiency of the algorithms have already been estimated. In order to determine which of the algorithms is the fastest, they have to be firstly tested. Different sets of inputs with a various size will be created to gather enough data to determine the

quickest algorithm. The testing will be consisting of 3 main parts:

1) Create an implementation of the 3 different algorithms of the convex hull in C++.
2) Measure execution time and operation calls for all algorithms
3) Evaluate data gathered in the second step and show which of the algorithms seems to be the fastest

3.2 Building the application

To carry out the research plan and gather the information about the execution time of the algorithms, I will create a C++ script.

This script will generate a random set of points in a given range for a given input and change the input values itself. For the given set of points one of the algorithms will be used to create the convex hull and the execution time and operation calls of that process will be calculated. Furthermore, the desirable number of vertices of the convex hull will be calculated and created.

In order to get more precise results, for a given size of input 100 different sets of points will be created. The overall execution time and number of operation calls will be the average of those measurements.

Gathered results will be written into an excel file to easily compute the data and create appropriate tables and diagrams.

3.2.1 Generating random points

By generating random points it is meant that for a given input the necessary number of

points will be created and a random x- and y-coordinate will be added to each of the points. In order to get a random number of a given range, library <*cstdlib.h*> will be included[3]. Every set of points should have different, randomly generated values. Hence, the following command will be used to make sure that new random numbers will be generated.

```
srand ( time(NULL) );
```

3.2.2 Computation time

Computation time is a core aspect that allows us to compare time complexity of the algorithms. For measuring this time, library <*ctime*> will be included that also involves *clock()* function **(**H. & Pate, 2010**)**. Two variables will be created which will store time before and after the algorithm is executed. By calculating the difference of these two variables we get the computation time of the algorithm in seconds.

3.2.3 Operation calls

For the small number of points given in the input it is rather unprecise to calculate the computation time as it might reach zero value. For this reason when the input size is lower, the number of operation calls of the algorithms will be compared. The number of operation calls will be calculated as the sum of the most important and characterising operations for a given algorithm.

[3] http://www.cplusplus.com/reference/cstdlib/rand/

4 Data collections and results

4.1 System specifications

During all measurements, a computer with the following properties was used:

Processor: AMD A8-5550M APU with Radeon™ HD Graphics 2.10 GHz

RAM: 8.00 GB

4.2 Operation calls

Every data point mentioned in this extended essay is an average of 100 measurements in order to decrease uncertainty and improve precision of the testing. Since the random generation of points might create different number of vertices of the hull for the same points of the set, I firstly set a fixed number of vertices by calculating their coordinates.

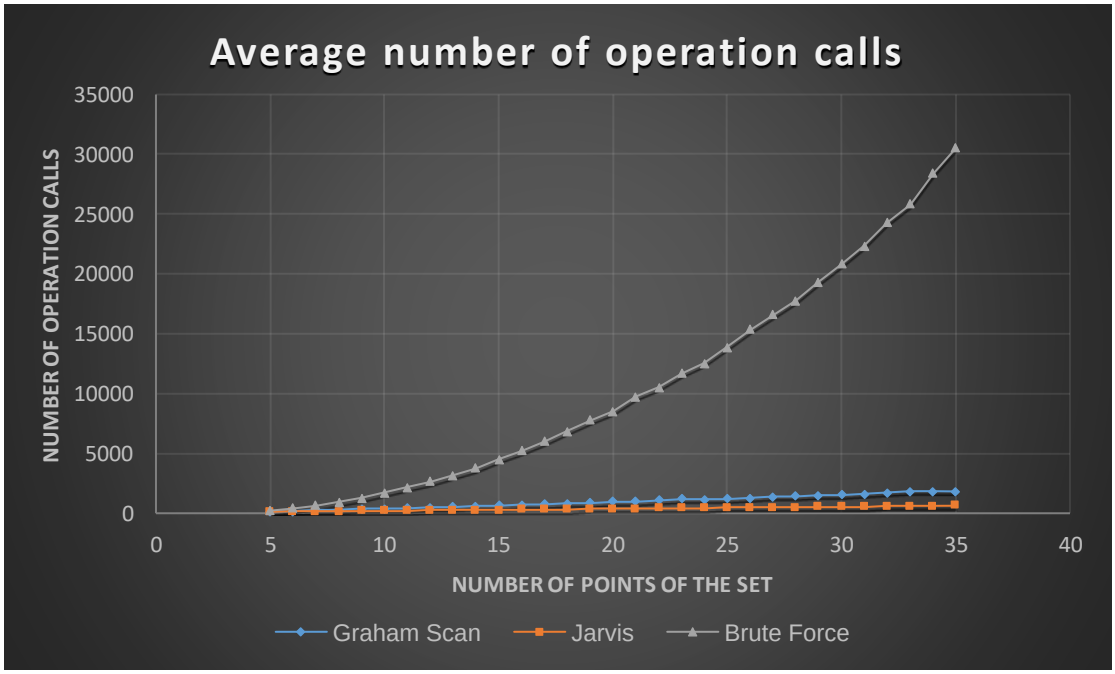

Figure 5. Dependence of the number of operation calls on the number of points given in the input. The number of vertices of the convex hull remains the same – 5.

Even for the low numbers of points the number of operation calls significantly differs. The brute force algorithm takes much more operations than the other two algorithms.

The Jarvis algorithm seems to be more efficient as for the number of operation calls than the Graham scan algorithm. This is caused by the low number of vertices of the hull. If the number of vertices were higher, the Jarvis algorithm should require more operation calls than the graham scan algorithm as explained in section 2.3.2. Hence, I will examine behaviour of the algorithms within the same range of the set of points when the number of vertices of the convex hull is set to the size of the input.

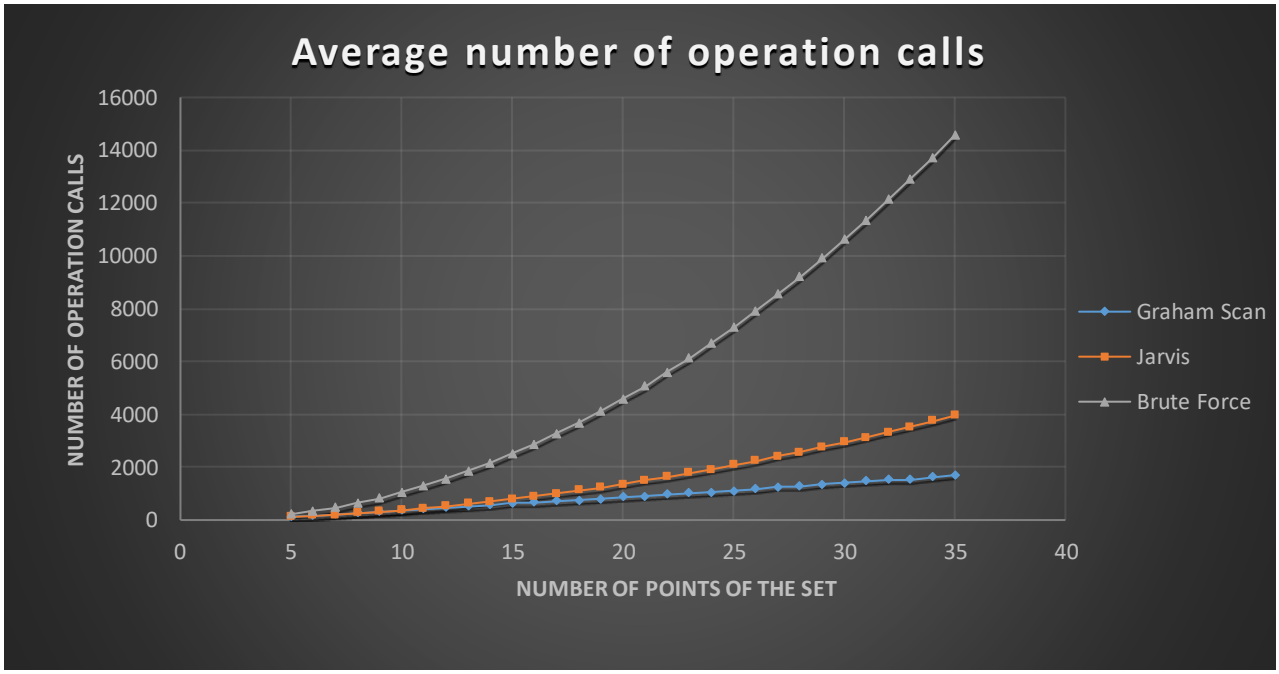

Figure 6. Dependence of the number of operation calls on the number of points given in the set. The number of vertices of the convex hull is the same as the number of points in the input.

The Graham scan algorithm works more efficiently than the Jarvis algorithm because the number of vertices of the convex hull has been increased. Since the number of vertices is the same as size of the input point set, the dependence of the time complexity on the number of point set should be quadratic. This will be further tested.

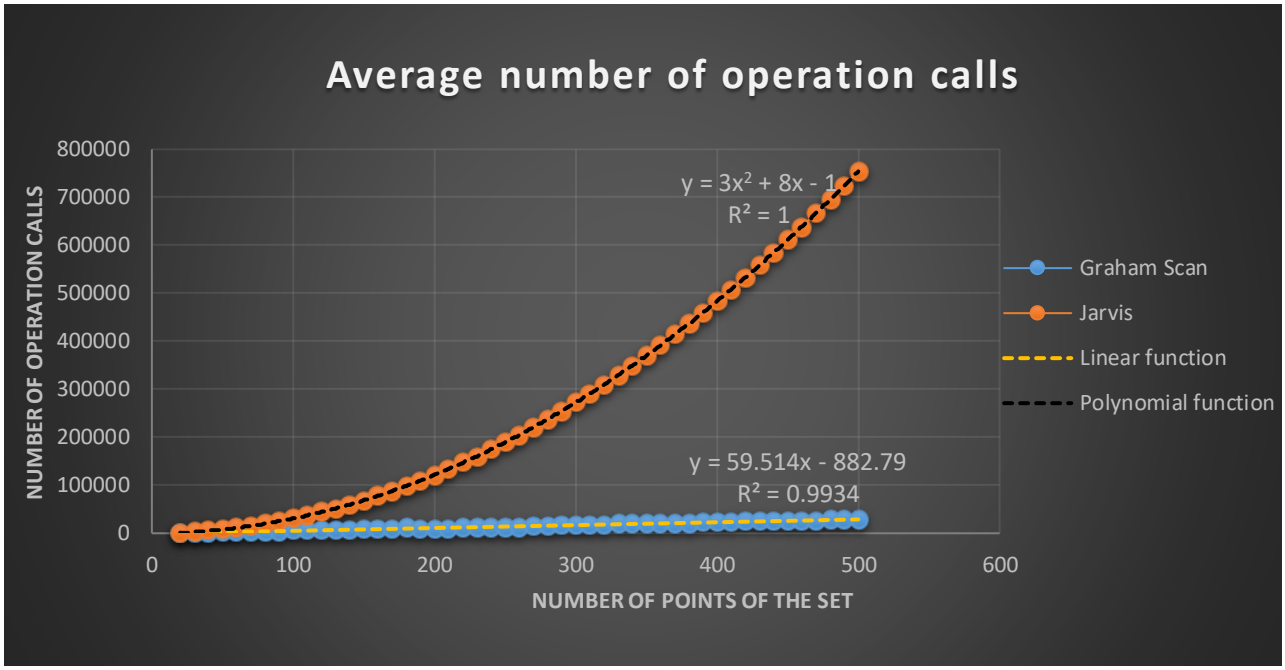

Figure 7. Dependence of the number of operation calls on the number of points. The number of vertices of the convex hull is the same as the number of points in the input.

A wider range of input size was used to determine the behaviour previously mentioned. The data in the figure 7 justify that the Jarvis algorithm portrays a polynomial function with second order when the number of vertices is the same as the size of point set. The Graham scan algorithm is much more time efficient when the number of vertices approaches number of points of the entire set. Nevertheless, the Graham Scan algorithm seems to trace a linear function which was not supposed as the algorithm includes a sorting method.

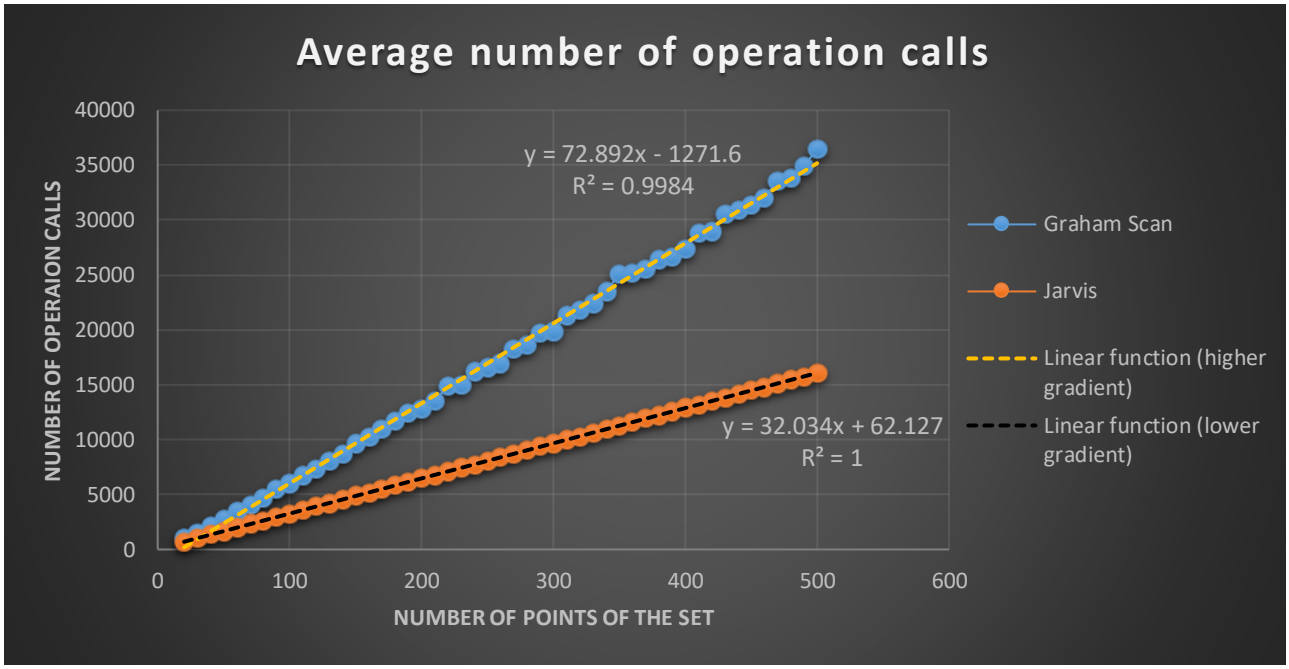

Figure 8. Dependence of the number of operation calls on the number of points. The number of vertices of the convex hull remains the same – 10.

The number of vertices being held at a constant value, the Jarvis algorithm is only dependent on the number of points of the set. This relation is linear which was theoretically estimated and practically proved. The Figure 8 demonstrates that when the number of vertices is relatively low, then this algorithm is more time efficient than the Graham scan algorithm.

The relation between the time complexity of the Graham scan algorithm and the number of point set seems linear.

4.3 Computation time

As in the part 4.2, every result mentioned is an average of 100 measurements. The number of vertices of the convex hull will always be calculated so that it does not randomly change during the measurements. The computation time of the algorithms is going to be measured on higher level of inputs since in the lower sizes of input it often

reached 0 value in mini seconds.

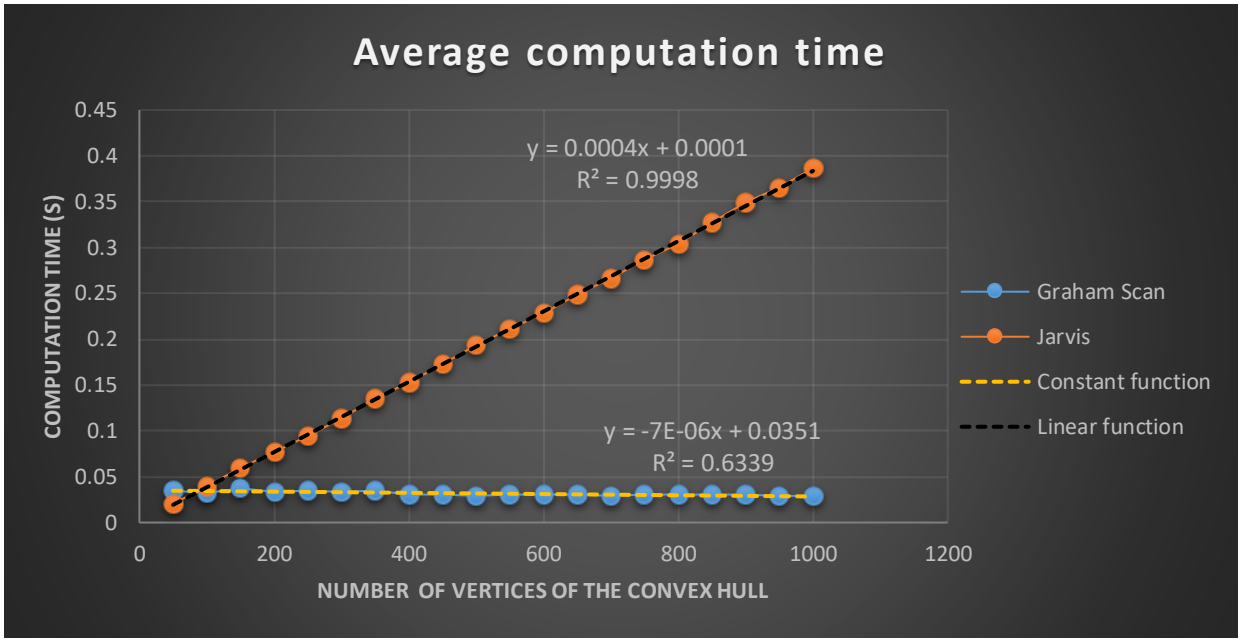

Figure 9. Dependence of the number of computation time on the number of vertices of the convex hull. The number of the entire point set remains the constant – 1000.

The data of computation time gathered from the figure 9 prove that the time complexity of the Jarvis algorithm is proportional to the number of vertices of the convex hull. The computation time of the Graham scan slightly decreases as the number of vertices increases. Nonetheless, the function that the Graham scan algorithm seems to trace has a very low coefficient by which the number of the vertices of the convex hull is multiplied. This makes the relation almost independent on the number of vertices. In can be deduced from the figure 9 that the higher the number of vertices of the hull is, the more efficient is the Graham scan algorithm.

Average number of computation time. Number of point set is 10000 and the number of vertices is 100.	
Algorithm	Computation time (s)

Brute Force	9.28417
Jarvis	0.04288
Graham Scan	0.03684

Table 1. Average computation time of the algorithms when the number of the points of the set is 10 000 and the number of vertices of the convex hull is 100.

The table 1 shows the enormous difference in computation time between the brute force algorithm and the other two algorithm. The Graham scan algorithm is working more time efficiently even though the amount of vertices of the hull is only 1% of the points given in the input. This infers that the Jarvis algorithm is more time efficient for a very small ratio between the number of vertices and the number of points of the set.

Average computation time when the number of vertices of the convex hull is randomly created.		
Number of points	Graham Scan algorithm Computation time (s)	Jarvis algorithm Computation time (s)
100	0.00031	0.00016
1000	0.00313	0.00100
10000	0.03436	0.00700
50000	0.15951	0.05612

Table 2. Average computation time of the algorithms when the number of vertices of the convex hull is randomly created. The size of the overall point set varies as the table illustrates.

The Jarvis algorithm works more efficiently than the Graham Scan algorithm when the number of vertices of the convex hull is randomly generated. The reason for this is that the number of vertices in such a case is relatively low. It is improbable for range of this size to include enough points for the Jarvis algorithm to work less time efficiently than

the Graham scan algorithm.

5 Evaluation and conclusion

The aim of this extended essay was to compare the time efficiency of different algorithms for finding the convex hull.

The algorithms were tested on randomly generated points. The number of vertices of the convex hull was in most cases calculated. Nevertheless, the table 2 shows behaviour of the algorithms when the vertices are randomly generated. The number of points in the overall set and/or the number of vertices of the convex hull were changed in order to achieve appropriate results.

Understanding the algorithms and constructing the data sets was a lengthy process that required a lot of energy and effort. Nevertheless, it was an enriching experience that was necessary for writing this extended essay.

Comparison the efficiency of different algorithms of the 2 dimensional convex hull.

The three following algorithms were compared:

- Brute force algorithm
- Jarvis algorithm
- Graham Scan algorithm

The brute force algorithm confirmed to be algorithm with the worst time efficiency with computation time 9.28417s when the number of point set was 10 000 and the number

of vertices 100. It cannot be evidently stated which of the algorithms was the fastest. For absolutely randomly generated points the Jarvis algorithm worked more efficiently with computation time 0.00700 seconds at the size of the input being 10 000. For the same values, the Graham scan algorithm had computation time 0.03436 seconds. Nonetheless, for the number of vertices being relatively low to the number of all points of the set, the Graham scan algorithm worked significantly more efficiently. When the number of point set was 500 and the number of vertices the same as the size of the point set, the Graham scan required 28 694 operation calls. For the same instructions, the Jarvis algorithm needed 753 999 operation calls.

The convex hull has a wide scale of real life applications from avoiding an obstacle to a possible evacuation of a city. Hence, optimising the algorithms to achieve a lower time complexity of finding the convex hull is fruitful not only for competitive programming, but is also used in different fields in the real life.

5.1 Future study

In order to lower the time complexity of finding the convex hull, other algorithms such as The ultimate planar convex hull algorithm or the Chan's algorithm could be implemented.

Furthermore, other than randomly generated sets of points could be used. For instance, a plan of a city or other objects from the real life.

Finally, the algorithms could be tested on a wider range of data. This could include the number of the points given in the input, the number of vertices of the hull as well as the

range in which the points were generated.

Bibliography

Bradley, S. (2016). ECG Response: January 12, 2016. *Circulation, 133*(2), 232-233. doi:10.1161/circulationaha.115.020740

C., & Simonson, S. (2012, October 30). Geometric Algorithms: Graham & Jarvis - Lecture 10. Retrieved from https://www.youtube.com/watch?v=J5aJEcOr6Eo&t=3534s

Calanake, S. (n.d.). Appendix B. Big Oh Notation. *Gamma*. doi:10.1515/9781400832538.219

Chan, T. M. (n.d.). Geometry. doi:10.1155/7958

Cormen, T., & Balkcom, D. (2014). Khan Academy. Retrieved from https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2008). Introduction to Genetic Algorithms. doi:10.1007/978-3-540-73190-0

Fudan. (2007). Applications. Retrieved from http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm

Gift wrapping algorithm. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

H., & Pate, R. (2010, May 11). Easily measure elapsed time. Retrieved from http://stackoverflow.com/questions/2808398/easily-measure-elapsed-time

Koochooloo, N., & M. (2013, May 12). Sorting points by their polar angle in Java. Retrieved from http://stackoverflow.com/questions/16509100/sorting-points-by-their-polar-angle-in-java

Muhammad, P. R. (2010, May 5). Convex Hull. Retrieved from http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/ConvexHull/convexHull.htm

Rufai, R. (2015, April 5). What are the real life applications of convex hulls? Retrieved from https://www.quora.com/What-are-the-real-life-applications-of-convex-hulls

Time Complexity of Algorithms and Data Structures | Data Structure Tutorial | Studytonight. (2016). Retrieved from http://www.studytonight.com/data-structures/time-complexity-of-algorithms

Appendix A – source code

A.1 – Brute force algorithm

```
#include <iostream>
#include <stack>
#include <vector>
#include <ctime>
#include <fstream>
#include <math.h>
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define PI 3.14159265

using namespace std;

struct Point {
    double x,y;
};

long long operationCalls;
long long operationCallsAverage;

int getOrientation(Point a, Point b, Point c) {

    int value;

    // Determine orientation of the triplet by comparing the
gradients
    value = (b.y-a.y)*(c.x-b.x) - (c.y-b.y)*(b.x-a.x);
    operationCalls++;

    // If value > 0, then the direction is clockwise
    // If value== 0, the points are on the line
    // If value < 0, the direction is counterclockwise
    if (value > 0) return 1;
    if (value < 0) return 2;

    // If neither of the conditions above have been fulfilled, then
the points are on the same line
```

```
    return value;
}


void convexHull(Point points[], int n) {

    vector<Point> pointsOfHull;
    bool cond;
    int val;

    for(int i=0; i<n; ++i) {

        int j=0;
        cond = false;
        operationCalls+=2;

        while (j<n && cond==false) {

            cond = true;
            operationCalls++;

            if (i != j) {

                // If all remaining points are on the left side
                // That means that the cond remains true, then add
the point i to the convex hull
                for (int k=0; k<n; ++k) {

                    operationCalls+=2;

                    if (k != i && k != j) {

                        val =
getOrientation(points[i],points[j],points[k]);
                        operationCalls++;

                        // if for all k val == 2 => point is part of
the hull
                        // if for any k val == 0 => point is on the
line
                        // if for any k val == 1 => point is not part
of the hull
                        operationCalls++;
```

```
                        if (val <= 1) {
                            // Break from the loop, there is already
one point for which the condition is not valid

                            cond = false;
                            operationCalls++;

                            break;
                            // Continue with the j-loop

                        }
                    }
                }
                // If condition is still true, all remaining points
were on the left side
                // Hence, add the point i to the hull, continue with
the i-loop

                operationCalls++;
                if (cond) {

                    pointsOfHull.push_back(points[i]);
                    operationCalls++;

                    break;
                }
            }

            cond=false;
            j++;
            operationCalls+=2;

        }
    }
}
```

A.2 – Jarvis algorithm

```
#include <iostream>
#include <stack>
#include <vector>
#include <ctime>
#include <fstream>
#include <cstdlib>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

using namespace std;

#define PI 3.14159265

long long operationCalls, operationCallsAverage;

struct Point {
    double x,y;
};

int getOrientation(Point a, Point b, Point c) {

    int value;

    // Determine of the triplet by comparing the gradients
    value = (b.y-a.y)*(c.x-b.x) - (c.y-b.y)*(b.x-a.x);
    operationCalls++;

    // If value > 0, then the direction is clockwise
    // If value== 0, the points are on the line
    // If value < 0, the direction is counterclockwise
    if (value > 0) return 1;
    if (value < 0) return 2;

    // If neither of the conditions above have been fulfilled, then
the points are on the same line

    return value;
}

void convexHull(Point points[], int n) {

    vector<Point> pointsOfHull;

    //Find the left-most point of the set
    int minX = points[0].x, indOfLeftMostPoint=0;
    for (int i=1; i<n; ++i) {

        operationCalls+=2;
```

```
        if (points[i].x < minX) {

            minX = points[i].x;
            indOfLeftMostPoint = i;
            operationCalls += 2;

        }
    }

    // indOfCurrentHullPoint => index of the point that is added to
the hull
    // indOfNewHullPoint => index of the points that has been found
to be the point of the hull
    int indOfCurrentHullPoint, indOfNewHullPoint;

    // The first point to be added to the hull is the left most point
    indOfCurrentHullPoint = indOfLeftMostPoint;
    operationCalls++;

    // Go through the points and add all points that are at the
circumeference of the set
    do {

        // Adding current point of the hull
        pointsOfHull.push_back(points[indOfCurrentHullPoint]);
        operationCalls++;

        // Let the new hull point be the point after the currently
added point
        // %n so that all the points are reached
        indOfNewHullPoint = (indOfCurrentHullPoint + 1)%n;
        operationCalls++;

        for (int i=0; i<n; ++i) {

            operationCalls+=2;

            // If the given triplet is counterclockwise, it means
that the i-th point is on the right side of the line connecting
            // the currentHullPoint and newHullPoint
            // Hence, the newHullPoint is certainly not a part of the
hull => rewrite it by the i-th point
```

```
            if (getOrientation(points[indOfCurrentHullPoint],
points[i], points[indOfNewHullPoint]) == 2) {

                indOfNewHullPoint = i;
                operationCalls++;

            }
        }
        // The newHullPoint is now the most counter-clockwise point
with respect to the currently added point
        // Hence, store the newHullPoint to the currentHullPoint that
will be added to the hull in the next iteration

        indOfCurrentHullPoint = indOfNewHullPoint;
        operationCalls++;
        operationCalls+=2;

    } while ( indOfLeftMostPoint != indOfCurrentHullPoint );
    // Finish when the left-most point is reached again

}
```

A.3 – Graham Scan algorithm

```
// This code is copied and adjusted from GeeksforGeeks
// URL: http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/
// Website Title: Convex Hull | Set 2 (Graham Scan) - GeeksforGeeks

#include <iostream>
#include <stack>
#include <vector>
#include <ctime>
#include <fstream>
#include <math.h>
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define PI 3.14159265

using namespace std;
```

```
struct Point
{
    double x, y;
};

// A globle point needed for  sorting points with reference
// to  the first point Used in compare function of qsort()
Point p0;

long long operationCalls, operationCallsAverage;

// A utility function to find next to top in a stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    operationCalls+=4;

    return res;
}

// A utility function to swap two points
void swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
    operationCalls+=3;

}

// A utility function to return square of distance
// between p1 and p2
int distSq(Point p1, Point p2)
{
    operationCalls++;
    return (p1.x - p2.x)*(p1.x - p2.x) +
          (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
```

```
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point a, Point b, Point c)
{
    int val = (b.y - a.y) * (c.x - b.x) - (b.x - a.x) * (c.y - b.y) ;
    operationCalls++;

    operationCalls+=2;
    if (val == 0) return 0;  // colinear
    return (val > 0)? 1: 2;  // clock or counterclock wise
}

// A function used by library function qsort() to sort an array of
// points with respect to the first point
int compare(const void *vp1, const void *vp2)
{
   Point *p1 = (Point *)vp1;
   Point *p2 = (Point *)vp2;
   operationCalls+=2;

   // Find orientation
   int o = orientation(p0, *p1, *p2);
   operationCalls++;

   operationCalls+=2;
   if (o == 0) {
        return (distSq(p0, *p2) >= distSq(p0, *p1))? -1 : 1;
   }

   return (o == 2)? -1: 1;

}

void quickSort(Point points[], double arr[], int left, int right) {

    int i=left, j=right;
    double pivot = arr[(left+right)/2];
    operationCalls+=3;

    while (i<=j) {

        operationCalls++;
```

```
        while (arr[i] < pivot) {
            ++i;
            operationCalls+=2;
        }
        while (arr[j] > pivot) {
            --j;
            operationCalls+=2;
        }

        operationCalls++;
        if (i<=j) {
            swap(arr[i],arr[j]);
            swap(points[i],points[j]);
            --j;
            ++i;
            operationCalls+=4;
        }
    }

    operationCalls++;
    if (left < j) {
        quickSort(points, arr, left, j);
        operationCalls++;
    }


    operationCalls++;
    if (right > i) {
        quickSort(points, arr, i, right);
        operationCalls++;
    }

}


// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{

    // Find the bottommost point
   int minY = points[0].y, min = 0;
   operationCalls+=2;
```

```
    for (int i = 1; i < n; i++)
    {
      int y = points[i].y;
      operationCalls+=2;

      // Pick the bottom-most or chose the left
      // most point in case of tie

      operationCalls++;
      if ((y < minY) || (minY == y && points[i].x < points[min].x))
         minY = points[i].y, min = i;
         operationCalls++;
    }

    // Place the bottom-most point at first position
    swap(points[0], points[min]);
    operationCalls++;

    // Sort n-1 points with respect to the first point.
    // A point p1 comes before p2 in sorted ouput if p2
    // has larger polar angle (in counterclockwise
    // direction) than p1
    p0 = points[0];
    operationCalls++;

    double cotArray[n];
    cotArray[0] = 0;
    operationCalls++;

    for (int i=1; i<n; ++i) {

        operationCalls+=2;

        if ( (points[i].y - p0.y) != 0) {
            cotArray[i] = - (points[i].x - p0.x) /
(double)(points[i].y - p0.y);
            operationCalls++;
        }
        else {
            // This point has the same y-coordinate, so the polar
angle is 0
            cotArray[i] = -10000;
            operationCalls++;
        }
```

```
    }

    quickSort(points, cotArray, 1, n-1);
    operationCalls++;

    // If two or more points make same angle with p0,
    // Remove all but the one that is farthest from p0
    // Remember that, in above sorting, our criteria was
    // to keep the farthest point at the end when more than
    // one points have same angle.

    stack<int> index;
    stack<double> distance;

    bool cond;
    double distanceMax;

    int indexMax;
    int m = 1;
    operationCalls++;

    for (int i=1; i<n; i++)
    {

        cond = false;
        operationCalls+=2;

        // Keep removing i while angle of i and i+1 is same
        // with respect to p0
        while (i < n-1 && orientation(p0, points[i],points[i+1]) ==
0){

            cond = true;
            index.push(i);
            distance.push( sqrt(points[i].x * points[i].x + points[i].y
* points[i].y) );
            i++;
            operationCalls+=4;

        }

        if (cond) {
             distanceMax = distance.top();
             indexMax = index.top();
```

```
            index.pop();
            distance.pop();
            operationCalls+=4;
        }


        while (!distance.empty()) {

           operationCalls+=2;

           if (distance.top() > distanceMax) {
               distanceMax = distance.top();
               indexMax = index.top();
               operationCalls+=2;
           }

           distance.pop();
           index.pop();
           operationCalls+=2;

        }



        operationCalls++;
        if (cond) {
            //cout << "Points[" << indexMax << "]    " <<
points[indexMax].x << "  " << points[indexMax].y << endl;
            points[m] = points[indexMax];
        }
        else {
            points[m] = points[i];
        }

        m++;  // Update size of modified array
        operationCalls++;

    }

    // If modified array of points has less than 3 points,
    // convex hull is not possible
    if (m < 3) return;

    // Create an empty stack and push first three points
```

```
    // to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);
    operationCalls+=3;

    // Process remaining n-3 points
    for (int i = 3; i < m; i++)
    {
        operationCalls++;
        // Keep removing top while the angle formed by
        // points next-to-top, top, and points[i] makes
        // a non-left turn
        while (orientation(nextToTop(S), S.top(), points[i]) != 2) {
            S.pop();
            operationCalls++;
        }

        S.push(points[i]);
        operationCalls++;
    }
}
```

A.4 – Main Function

```
int main()
{

    ofstream myFile;
    myFile.open("data.csv");

    // n => number of the points of the set
    // h => number of the vertices of the created hull
    int n,h;
    n = 5;

    int distance = 12000;
    double increment, angle;

    Point points[10000];
```

```
    while (n <= 35) {

        h=n;

        myFile << "N is  " << n ;

        while (h<=n) {

            myFile << "," <<  "H is  " << h << ",";

            operationCallsAverage=0;

            for (int ind=0; ind<100; ++ind) {

                // Make sure that every time new random coordinates
will be generated
                srand ( time(NULL) );

                operationCalls=0;

                // Start the stopwatch
                clock_t begin = clock();

                // The range of the coordinates is -2499 .. 0 .. 2499
                for (int i=0; i<n-h; ++i) {
                    points[i].x = rand()%4999 - 2499;
                    points[i].y = rand()%4999 - 2499;
                }

                increment = 360 / (double)h;
                angle = 0;
                int i = (n-h);

                while (i<n && angle < 360) {
                    // Calculate the points of the hull using
trigonometric functioncs
                    points[i].x = distance*cos(angle*PI/180);
                    points[i].y = distance*sin(angle*PI/180);
                    ++i;
                    angle += increment;
                }

                // Call the procedure convexHull on the given points
to create the hull of the points
```

```
                convexHull(points, n);

                // End the stopwatch
                clock_t end = clock();

                // Calculate elapsed time for the process
                double elapsedTime = double (end-begin)/
CLOCKS_PER_SEC;

                myFile <<  elapsedTime << "," ;

                operationCallsAverage += operationCalls;

            }

            myFile << "," << operationCallsAverage;
            operationCallsAverage /= 100;
            myFile << "," << operationCallsAverage;
            myFile << endl;

            h+=1000;
        }

        n += 1;
    }

    myFile.close();
    return 0;

}
```