Convex Hulls from Line Intersections: An In-Depth Comparative Study

Patrik-Grațian Pusztai
Computer Science Department,
West University of Timișoara,
Email: patrik.pusztai05@e-uvt.ro

January 2025

Abstract

This paper follows two essential concepts in the field of computational geometry: generating the intersection of a finite number of lines and computing the convex hull of a finite number of points. We find the intersection of lines using Cramer's rule and the Bentley-Ottmann Line Sweep algorithm. We generate convex hulls using the points resulting from the process mentioned above and the Graham's Scan, Andrew's variant of the Graham's Scan (also known as Andrew's monotone chain) and Jarvis March algorithms. In the latter half of the paper we present the results achieved through comparing algorithms based on time and space efficiency to find the most efficient combination of algorithms.

Contents

1 Introduction

The construction of convex hulls is one of the most studied problems in computational geometry [4]. In this study, the points forming the convex hull possess a special property—they are intersections of lines. Computing line intersections also represents a fundamental problem in this field and together, they offer many real world applications, the most notable being in geographical information systems(e.g. computing accessibility maps and boundaries), robotics and motion planning, and computer visualizations(e.g ray shooting/ray tracing) [2, 6].

Motivation of the problem

This paper aims to review existing literature and experimentally evaluate the time and space efficiency of the algorithms used to compute line intersections and convex hulls. With both problems being applied across many important fields, we believe that identifying optimal algorithmic approaches are absolutely necessary.

2 Description of the problem and solution

2.1 Line intersection algorithms

Input: Let set $S = \{s_1, \ldots, s_n\}$ of (closed) line segments.
Output: All intersection points between lines in S.

For solving this problem we used a trivial algorithm which applies Cramer's rule to find the intersections of lines and an algorithm which uses a sweep line for the same result. We will describe how each algorithm manages to reach to expected output.

Cramer's Rule

We create the equation of a line using the points which create the segments.

$$\text{Line 1: } a_1x + b_1y = c_1$$

$$\text{Line 2: } a_2x + b_2y = c_2$$

We compute the determinant of the equations to check if the two lines are parallel or if they intersect at a unique point. If the determinant is zero we conclude that they do not intersect. Else, we calculate their intersection point by applying the formulas below.

$$x = \frac{c_1b_2 - c_2b_1}{\text{determinant}}$$

$$y = \frac{a_1c_2 - a_2c_1}{\text{determinant}}$$

Bentley-Ottmann Line Sweep Algorithm

We add the starting point and the ending point of each segment to a list called events. The list events will be sorted lexicographically based on the values of the x coordinates, with a focus on 'start' points coming before 'end' points in case of equalities. We simulate a vertical sweep line which we move from left to right and iterate through the list events. When the sweep line reached the start point of a segment, the segment becomes **active**. An active segment is a segment which intersects the sweep line. The intersection is done based on the orientation of the points that define the extent of the segments.[8]

2.2 Convex hull algorithms

Input: Let set $S = \{S_1, \ldots, S_n\}$ of line intersection points
Output: Ordered subset of h points which construct the smallest enveloping polygon of the points in set S.

For solving this problem we used the algorithms Jarvis march, Grahams's scan and Andrew's variant of the Graham's scan also known as Andrew's monotone chain. We will describe how each algorithm manages to reach the output.

Jarvis March

We find the leftmost point or the point with the minimum x coordinate which we call p_0. We add p_0 to a list which contains all the points in the convex hull. We will call this list hull. The next point in the convex hull will be the point furthest to the right. To find this point we iterate through the list of points and find the orientation of p_0, pcurrent and i where pcurrent tracks the current rightmost vertex and i is the next point in the iteration. If i is further right we replace pcurrent with i and so on. After the iteration is complete we add pcurrent to the list hull and perform the steps described above for all remaining points.[5] The time complexity of this algorithm is O(nh) where h represents the number of points on the convex hull.[3]

Graham's Scan

We find the point with the minimum y coordinate which we call p_0(in case of equalities we choose the one with the smallest x-coordinate). We sort the remaining points based on the polar angle they form with this point. We iterate through the list of points and check at each step whether the last three points make a non-left turn , if they do we pop the stack and check again until we find three points that make a left turn. The time complexity of this algorithm is of O(nlogn).[3] [5]

Andrew's variant of the Graham's Scan

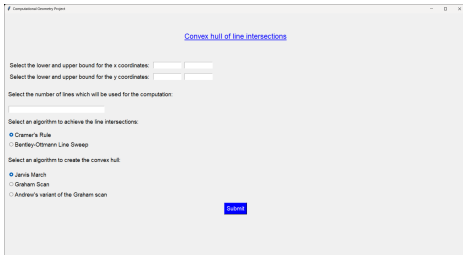
Andrew's monotone chain , also known as Andrew's variant of the Graham's scan proposes the construction of two hulls, a lower hull and an upper hull. Un-

like Graham's scan, which sorts points by polar angle, Andrew's algorithm first sorts the points lexicographically. We build the lower hull by iterating through the sorted points while maintaining a stack and removing points that cause a non-left turn. The upper hull is constructed in the same way, but iterating through the points in reverse order. After creating the hulls we concatenate them and check to make sure that there aren't any duplicate elements in the hull.The time complexity of this algorithm is of O(nlogn)

3 Implementation of the problem and solution

The algorithms used to conduct this study can be accessed through the attached Github repository . For the implementation of the algorithms [3, 5, 8, 6] were used.

When the user runs the code, they are met with a graphical user interface scene created using the module TKinter.



Users are asked to enter a minimum and maximum value for the x and y coordinates of the lines and the number of lines which they want to generate. The bounds can also be negative.

Until we haven't generated the exact number of lines provided by the user, we create two points Point1(x1,y1) and Point2(x2,y2) where x1,x2,y1,y2 are random but distinct integers in the range provided by the user. We achieved this by using the function **random.randint** from the module random. These points represent the basis for creating the lines in our application.

The user can choose the algorithm which will be used for computing the line intersections and the convex hull from the presented range of algorithms All values entered are checked and validated before running the visualization.

The output will be a visualization which contains the number of lines the user entered(colored with black), the intersections of the lines(colored with blue) and the convex hull(colored with red) (See Figure 2). To measure the execution time of each algorithm, we used the 'Horology' library, which allows us to track the runtime of specific functions. More on Horology can be found here: Horology documentation. For monitoring memory consumption of algorithms we used

the module memory-profiler in Python. More on memory-profiler can be found here:Memory-profiler

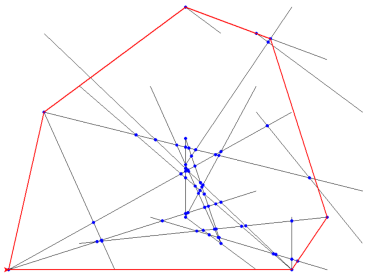


Figure 1: Example using 20 lines with intersections computed using Cramer's rule and the convex hull created using Jarvis March

4 Case study

4.1 Experiment description

The computer used to execute the study is a Lenovo ThinkPad E15 with AMD Ryzen 7 5700U with Radeon processor and 16.0 GB of RAM, a 64 bit operating system, Windows 11, and x64 based processor. For accurate results in the case study we took the following measures:

1. We tasked each algorithm with the same range of points and number of lines
   x ϵ (100,200) , y ϵ (200,300) for a number of lines lesser than **100**
   x ϵ (1000,2000) , y ϵ (2000,3000) for a number of lines equal to **500** or **1000**
   x ϵ (10000,20000) , y ϵ (20000,30000) for a number of lines equal to **5000** or **10000**

2. We ran 4 tests with the same value for each algorithm . The final results visible in the table represent the average value of the tests.

We compared the convex hull algorithms using their execution time. We used the units of measurement us(microsecond), ms(milisecond) and s(second).

Regarding the line intersection algorithms, we performed comparisons using their memory usage.The library memory-profiler in Python provides results in Mebibyte (shortened as MiB). It's important to note that the terms Mebiyte and Megabyte are closely related however 1 MIB =1024*1024 bytes and 1 MB=1000*1000 bytes.We will provide results using both measurements.

4.2 Results

4.2.1 Convex hull algorithms

The table below contains the average values of the execution times for each algorithm when tasked with creating the convex hull of the intersections of n number of lines where n is a positive integer.

Number of lines	Jarvis march	Graham's scan	Andrew's variant
10	114 us	247 us	75 us
50	1.37 ms	2.05 ms	0.994 ms
100	5.08 ms	6.03 ms	2.665 ms
500	160.5 ms	103.5 ms	51.75 ms
1000	1.029 s	501.75 ms	269 ms
5000	41.9 s	18.15 s	11.25 s
10000	2 min	85.3 s	56.175 s

Table 1: Values for execution time

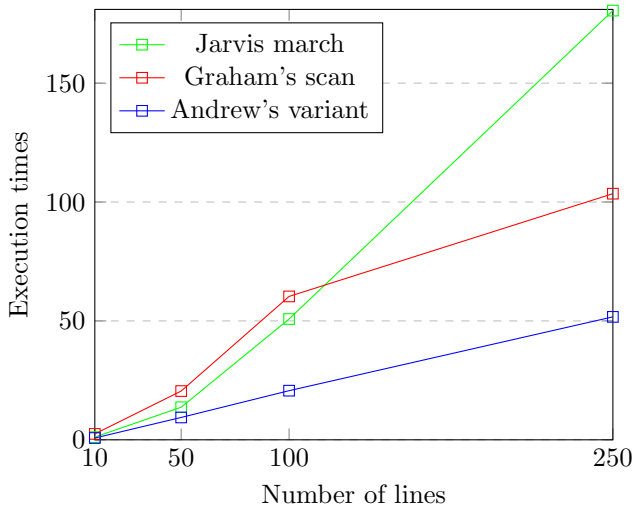


Figure 2: The behaviour of the convex hull algorithms when tasked with computing the hull of number of lines less than 250

4.2.2 Line intersection algorithms

Computing line intersections using Cramer's has a time and space complexity of $O(n^2)$. This happens because we check all pairs of lines for intersections leading to $\binom{n}{2}$ comparisons. For sweep line algorithms we have O(n) space complexity and time complexity of O(nlogn+ Ilogn) where I represents the number of

intersection points[9]. The table below contains the average memory required by each algorithm when tasked with computing the intersections of n number of lines.

Number of lines	Cramer's	Bentley Ottman
10	48.5 MiB/50.8 MB	48.6 MiB/50.9 MB
50	48.8 MiB/51.1 MB	48.7 MiB/51.0.6 MB
100	49.2 MiB/51.5 MB	48.9 MiB/51.3 MB
500	57.2 MiB/59.97 MB	55.4 MiB/58.1 MB
1000	72.8 MiB/76.3 MB	68.8 MiB/72 MB
5000	501.7MiB/526.07 MB	385.5 MiB/404.25 MB
10000	2031.96 MiB/2130.66 MB	1496 MiB/1568.67 MB

Table 2: Values for memory usage

5 Related work

There have been many articles published regarding comparative studies on convex hull algorithms or line intersections algorithm. One of the most notable ones is a 1997 article titled A Survey of Convex Hull Algorithms [7] which analyzes algorithms such as Quickhull, Graham's scan, and Jarvis March theoretically, using the Big O, Big Θ and Big Ω asymptotic notations and recurrence relations for the time and space complexity analysis. This paper reinforces the theoretical conclusions from this article through a more experimental approach, evaluating each algorithms performance on specific input. Although few papers assess both topics , we found an article titled "Computing the Convex Hull of Line Intersections" which manages to solve the problem of generating the convex hull of line intersections by creating an edge hull. An edge hull is a convex hull without the non-corner points [1]. The way the algorithm proposed by Atallah works is that it first sorts the lines by decreasing slope, it finds the set $Q = \{q_1, \ldots, q_n\}$ where q_i denotes the intersections of two lines, constructs the edge hull using a convex hull algorithm and deletes non corner points if necessary. However, it's important to note that his approach works only by assuming that all lines have distinct slopes, none of them are vertical, and that no more than two lines intersect at one point[1]. Our algorithm design works without these assumptions and manages to generate a convex hull as long as multiple intersection points exist.

6 Conclusions and Future work

6.1 Conclusions

- The visualization of the elements took much more than expected especially for a number of lines greater than 1000. We managed to adjust some of

the initial visualization processes to get faster results.

- Jarvis March is faster than Graham's scan when dealing with smaller sets of points. This is clearly showcased in the provided graph titled Figure 3, where we can observe that for up to 100 lines, Jarvis March outperforms Graham's scan in terms of execution time. However, as the number of lines increases, Jarvis March slows down. We expected this behaviour of Jarvis March considering that it's time complexity depends on the number of points on the convex hull which naturally increases as the number of lines increase.

- Andrew's version of the Graham scan is much faster than any of the algorithms used in this study. We believe this could be caused by multiple factors

    1. The initial sorting of the points is done lexicographically not based on the polar angle like in Graham's scan, thus avoiding unnecessary computations.
    2. Andrew's variant divides the convex hull problem into 2 subproblems. This division significantly reduces the complexity of the computations.

- Computing line intersections using Cramer's rule requires significantly more memory compared to the Bentley-Ottmann algorithm. This increase in memory usage is confirmed by the experimental results presented in Table 2. We believe this can be caused by how each method processes and stores data. Bentley-Ottmann only stores active segments and efficiently processes intersections without excessive duplication.

- Our experiments show that one of the best combination of algorithms for determining the convex hull of line intersections is using a line sweep algorithm(in our study we used the Bentley-Ottman algorithm) for generating the intersections and Andrew's version of the Graham scan for constructing the convex hull.

6.2 Future Work

In the future we hope to:

- Further explore the concept of implementing an optimal line sweep algorithm which is more efficient than the one used in the study.

- Extend the number of algorithms used for generating convex hulls with an implementation of Chan's algorithm and the Kirkpatrick–Seidel algorithm which run in O(nlogh) time complexity where n represents the input points and h the number of points on the convex hull and test their speed relative to Andrew's monotone chain algorithm.

References

[1] Mikhail J Atallah. Computing the convex hull of line intersections. *Journal of Algorithms*, 7(2):285–288, 1986.

[2] Cyril Briquet. Introduction to convex hull applications. Cours d'Algorithmique Avancée (INFO036), February 2007.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[4] Jeff Erickson. New lower bounds for convex hull problems in odd dimensions. *SIAM Journal on Computing*, 28(4):1198–1214, 1999.

[5] Jeff Erickson. Computational geometry course: Lecture schedule. `https://jeffe.cs.illinois.edu/teaching/compgeom/schedule.html`, 2022.

[6] Dave Mount. Cmsc 754: Lecture 4 – line segment intersection. Lecture Notes, 2023. Fall 2023, University of Maryland.

[7] Adli Abu Shaban. A survey of convex hull algorithms. 1997.

[8] Michiel Smid. Computing intersections in a set of line segments: the bentley-ottmann algorithm. 2003.

[9] Mihai-Sorin Stupariu. Geometrie computațională — suport de curs, 2016. Semestrul I, 2016-2017.