A practical data structure for the dynamic lower envelope of pseudo-lines

van Loon, B.

Award date:
2020

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

A practical data structure for the dynamic lower envelope of pseudo-lines

Master Thesis

Bas van Loon

Supervisors:
Kevin Buchin
Wolfgang Mulzer

Additional committee member:
Irina Kostitsyna

Final version

Eindhoven, September 2020

Abstract

Many algorithms to compute convex hulls or envelopes are static and cannot be extended to work dynamically, that is, to support changes like insertions or deletions in the data set. Overmars and van Leeuwen [11] proposed a data structure that allows this for both convex hulls and envelopes. The data structure is a binary search tree where elements are stored in the leafs and the internal nodes are augmented with another data structure to store the convex hull or envelope. Recently a new algorithm was developed to maintain the lower envelope of pseudo-lines by Agarwal et al. [1] which is based on the Overmars and van Leeuwen data structures.

While dynamic convex hulls have been studied extensively from a theoretical perspective, there is only very little experimental work. In this work we evaluate the practicability of the Overmars and van Leeuwen data structure and its extension by Agarwal et al.

We implemented and compared several versions of the data structure by Overmars and van Leeuwen and compared it to a simple approach based on maintaining the Delaunay triangulation. In our experiments, Delaunay triangulations outperformed the dedicated data structures except for very large sizes of convex hulls.

We also implemented the data structure for maintaining the lower envelope of pseudo-lines by Agarwal et al. The algorithm performs as expected, but our implementation suffers from robustness issues.

Acknowledgements

I want to start with thanking Kevin Buchin and Wolfgang Mulzer for their guidance, feedback and trust in me during this project. All the discussions we had were very interesting and I learned a lot from them. I would like to thank Lars and Michael for the time we spent together studying, computer science related discussions and feedback on my thesis progress.

I would also like to thank my parents and my sister for their support during the past years because it was sometimes a rough time, especially the last year. You will be missed mom.

And of course all my friends in and around Breda for their support and friendships. You made my student live a lot more fun and gave me a lot of great stories to remember.

Finally, I'm grateful to all the interesting people I met in Berlin which are too many to name for a wonderful time over there that ended way too soon unfortunately. But I specifically want to thank Andreas for being a good friend and always being available.

Bas van Loon

Teteringen
September 2020

Contents

Chapter 1

Introduction

Convex hulls of a set of points, or similarly envelopes of a set of lines, are geometric structures which are studied extensively. A lot of work has been done for developing algorithms to compute these structures, but almost all of them are non-dynamic. This means that changes in the set (by insertions or deletions) require a recomputation of the complete structure which is terribly inefficient. Overmars and van Leeuwen [11] were the first to propose a data structure and different algorithms to maintain convex hulls or envelopes dynamically, so allowing insertions or deletions. Section 3.3 contains an overview of further theoretical work on dynamic convex hulls. While later work improved the running time of operations, this also comes at the cost of considerably more complicated data structures.

The data structures proposed by Overmars and van Leeuwen are the most important ones to this thesis project. These data structures store the convex hull (right of Figure 1.1) in two parts: the lower hull (left side of Figure 1.1) and the upper hull which is the same but mirrored. The points are stored in the leafs of a binary search tree and each internal node is augmented with the two hulls of the subtree rooted at this node, so the convex hull of the entire set is stored at the root (as two separate parts). The hulls on their turn are stored as an ordinary binary search tree. An insertion or deletion requires the convex hull to be updated and this can be done in $O(\log^2 n)$ time. The convex hull is defined in Section 2.2 and more details about the dynamic maintenance of a convex hull by Overmars and van Leeuwen are explained and described in Chapter 3.

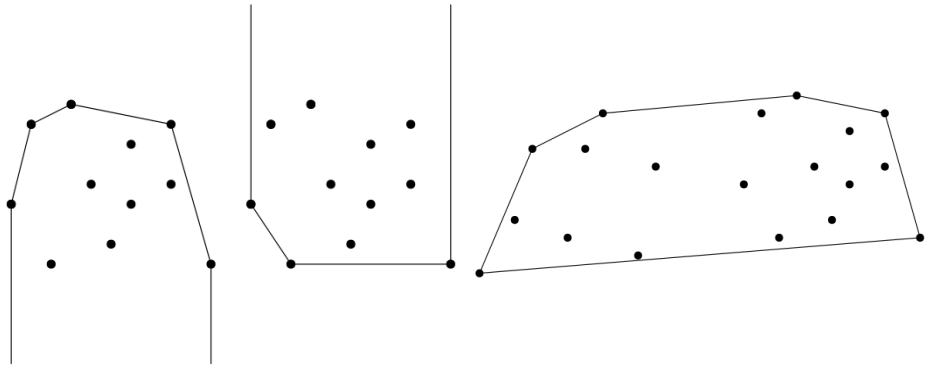


Figure 1.1: Left: the upper hull and lower hull of a set of points. Right: an example of a convex hull.

A new algorithm to maintain the lower envelope of pseudo-lines was recently proposed by a team of computer scientists led by Agarwal [1] based on the data structures by Overmars and van Leeuwen. The original algorithm by Overmars and van Leeuwen for envelopes could not be used because it was unclear how this could be abstracted to pseudo-lines. This is the first time

a solution was found for this specific problem and it is proved to work in theory, but it has not been evaluated yet in practice. Insertions and deletions can be done in $\log^2 n$ time. Examples of envelopes are given in Figure 1.2 which also shows various shapes of pseudo-lines. A detailed description about envelopes is found in Section 4.1 and details about pseudo-lines are found in Section 5.1.

The first step of this thesis project was to make an implementation for the convex hull because it was difficult to find existing implementations online and in existing literature (Section 3.3). The dynamic maintenance of envelopes of lines based on Overmars and van Leeuwen (Chapter 4) was studied and implemented afterwards and the envelope for pseudo-lines (Chapter 5) in the end using the new algorithm by Agarwal.

For pseudo-lines, the first step was to decide, which types of pseudo-lines to support in the implementation. Pseudo-lines may have various shapes and a few are shown in Figure 1.2: Overmars and van Leeuwen only covered the top-left case (ordinary lines) while the new algorithm should be able to cover all three cases shown in the figure, and more in the future. The implementation is evaluated in Chapter 6 along with the dynamic convex hull structures.

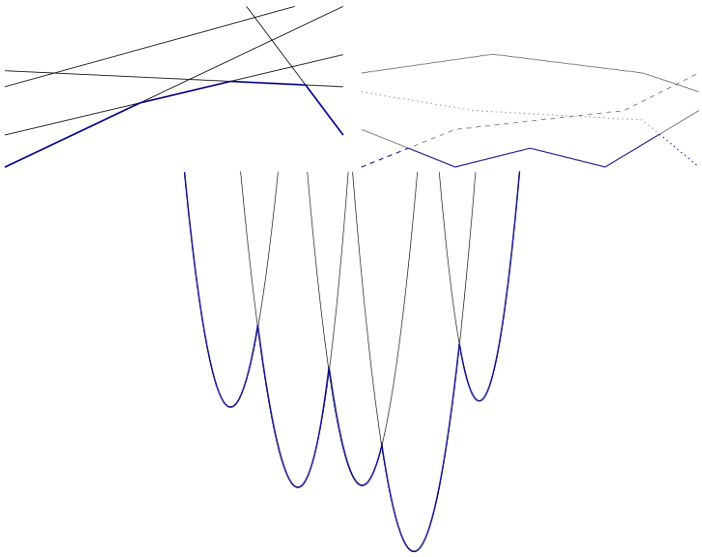


Figure 1.2: In clockwise order: lines, x-monotone collections of line segments, parabolas. The envelopes are presented by the fat blue parts. The x-monotone lines vary between dashed, dotted and solid lines so that distinguishing them is easier.

Eventually we want to know how well maintaining the envelope of pseudo-lines works in practice regarding the theoretical running time, its correctness and if it works on different types of pseudo-lines, hence the evaluation of the various structures in Chapter 6. These results are used to draw conclusions and recommendations for future work in Chapter 7.

1.1 Contributions

The goal of this thesis project is to verify if the data structure and algorithm proposed by Agarwal [1] are not only of theoretical interest, but can also lead to a practically efficient implementation. In order to achieve this, I present an implementation to maintain convex hulls since it is based on this, designed by Overmars and van Leeuwen [11]. This has been implemented before, but those implementations turned out to be outdated.

The dynamic maintenance of convex hulls has three variations: two variants of the Overmars and van Leeuwen structures where the data structure for the subhull differs and all variations are

experimentally evaluated for efficiency and correctness.

Overmars and van Leeuwen also described an algorithm to maintain envelopes based on the same data structure. This algorithm is theoretically correct but it turned out that implementing it is not trivial and requires additional work, which is explained in Chapter 4.

Because pseudo-lines are barely researched before, there was no good definition available until now. This definition and a practical guideline that makes it possible to use for the lower envelope of pseudo-lines which was necessary before implementing the pseudo-code [1]. The algorithm is implemented for three types of pseudo-lines: straight lines (that give ordinary envelopes) and two types of parabolas: $y = (x - a)^2 + b$ and $y = ax^2 + bx + c$. The running times and correctness for insertions and deletions of lines are evaluated for the first two types of lines. The implementation of the new algorithm is not bug free yet but it works almost correct.

This report finishes with recommendations for possible future improvements of maintaining dynamic convex hulls and envelopes of pseudo-lines.

Chapter 2

Preliminaries

This section covers some more in depth theoretical background which is needed in order to understand this thesis project. It is mainly about introducing the Overmars and van Leeuwen data structure and algorithm to maintain convex hulls dynamically along with the definition of envelopes, the concept of duality and the introduction of Delaunay triangulations. The algorithm to dynamically maintain envelopes will be covered in Chapter 4 and (envelopes of) pseudo lines will be covered in Chapter 5.

2.1 Operations

There are a few operations on data structures essential for this thesis project that are used often in the project and often can be performed on multiple types of data structures, but will work different for each type of data structure.

2.1.1 Random access

This means that for a data structure X of size n it is possible to retrieve any element in X at any arbitrary index in constant time. Arrays, sets, maps and lists like the C++ *vector* support this, but linked lists in C++ do not and trees cannot by their design.

2.1.2 Linear scan

A linear scan is a way to search for an element in a structure. You simply start in the beginning of the list and iterate through it until you find the element or the end is reached. It is not required that the list is sorted in order to search. It runs in linear running time in the worst case and less if the element is found.

2.1.3 Binary search

A more efficient of searching is done with a binary search. This works on lists as well as trees and runs in $O(\log n)$ time. Note that every log has a base of 2 in this report. A binary search halves the number of elements left to search in every iteration. The only requirement is for lists in order to be used is that they should be ordered and it works on binary search trees as well.

2.1.4 Split

For a collection S and element u, partition S into S_1 and S_2 such that $S_1 \cap S_2 = \emptyset$, $S = S_1 \cup S_2$, $S_1 = \{v : v \in S_1 \wedge v \leq u\}$ and $S_2 = \{v : v \in S_2 \wedge v > u\} = S \setminus S_1$. This operation works on lists and trees. The splitting of X into Y and X' is denoted as $Y = X[a, b]$ where the items in the

range $[a, b]$ are stored in Y and X', the items outside of this range, will be stored as X. X' and Y will be sorted, assumed that X is sorted before the split.

In practice, we will see that either a or b is unbounded (denoted as $\dots$). This means the splitting function will only have one parameter $c = X_i$ and splits X into $[\dots, X_i], [X_{i+1}, \dots]$. The function has some additional parameters to determine in which subset to include c and which subset to store as Y.

Lists

The implementation of this differs depending on how the list is designed. Splitting a linked list is really efficient if you know where to split but it is more intensive for lists or vectors because the elements have to be copied to a new container. So the running time is $O(n)$ time in the worst case. But linked lists do not support random access which means you first have to search for the place to split and each data structure has its own pros and cons.

Trees

Splitting for trees can be done in $O(\log n)$ time. Splitting a binary search tree on a key e works recursively. We go down to the node with value $v \leq e$ and for every internal node n we separate the left and right child from n and recursively split one of its children, depending on e. The leaf is the base case, where we simply do nothing. Traverse the tree back up to the root after e is found and concatenate all the subtrees to one of the two resulting trees.

SplitTree$(T, includeLeft, e)$
  if $T.v < e$ **then**
    $(T_1, T_2) \leftarrow SplitTree(T.left, includeLeft, e)$
    create a new tree T_3
    $T_3.left \leftarrow T_2, T_3.right \leftarrow T.right$
    $T_3.height \leftarrow 1 + \max(T_3.left.height, T_3.right.height)$
    return $(rebalance(T_1), rebalance(T_3))$
  else if $T.v > e$ **then**
    $(T_1, T_2) \leftarrow SplitTree(T.right, includeLeft, e)$
    create a new tree T_3
    $T_3.left \leftarrow T.left, T_3.right \leftarrow T_1$
    $T_3.height \leftarrow 1 + \max(T_3.left.height, T_3.right.height)$
    return $(rebalance(T_3), rebalance(T_2))$
  else ▷ The node or leaf containing e is found
    if $includeLeft$ **then**
      $T_r \leftarrow T.right$
      empty $T.right$
      $T.height \leftarrow 1 + T.left.height$
      return $(rebalance(T), rebalance(T_r))$
    else
      $T_l \leftarrow T.left$
      empty $T.left$
      $T.height \leftarrow 1 + T.right.height$
      return $(rebalance(T_l), rebalance(T))$
    end if
  end if

SplitTree splits a binary search tree T into $T_1 = \{a \in T : a < e \vee includeLeft \Leftrightarrow a = e\}$ and $T_2 = \{a \in T : e < a \vee \neg includeLeft \Leftrightarrow a = e\}$. So using the boolean *includeLeft* we can choose on which side e is stored. One of the trees is returned by the algorithm in the end of the split and the other tree replaces the old tree, this is determined by an additional parameter. This parameter

here is not mentioned in the pseudocode because *SplitTree* is wrapped by a helper function in the code. The rebalance is not necessary, but it is recommended to keep the depth of the tree around $O(\log n)$ nodes. Note that setting the heights is not necessary if rebalancing the trees is implemented because rebalancing should take care of that.

2.1.5 Concatenate

Concatenate should create the union of two collections S_1 and S_2 which can be sets, lists or trees among other things. A requisition is that for each $u \in S_1$ and $v \in S_2$ $u \leq v$ should hold. The result of the concatenation is $S = S_1 \cup S_2$. For this thesis project we only consider the concatenation of ordered lists and trees. In both cases it is denoted as $Z = X \cup Y$ for X and Y.

Ordered lists

Concatenating is similar to splitting with regard to the running time. It is easy for vectors or lists, we just have to insert all the elements from S_2 at the end of S_1. Linked lists are easy and efficient: we only have to point end of S_1 to the first element in S_2 and this takes constant time.

Trees

Concatenating two trees can be done in multiple ways. Let T_1 and T_2 be two binary search trees where every element in T_1 is smaller than the elements in T_2 and these will be concatenated into a new tree T_3. The easiest solution is to add T_1 to the leftmost leaf of T_2, but this can make T_3 extremely unbalanced. Another way is to create a new node n, assign T_1 to $n.left$ and T_2 to $n.right$. These solutions are fast but the result can be extremely unbalanced.

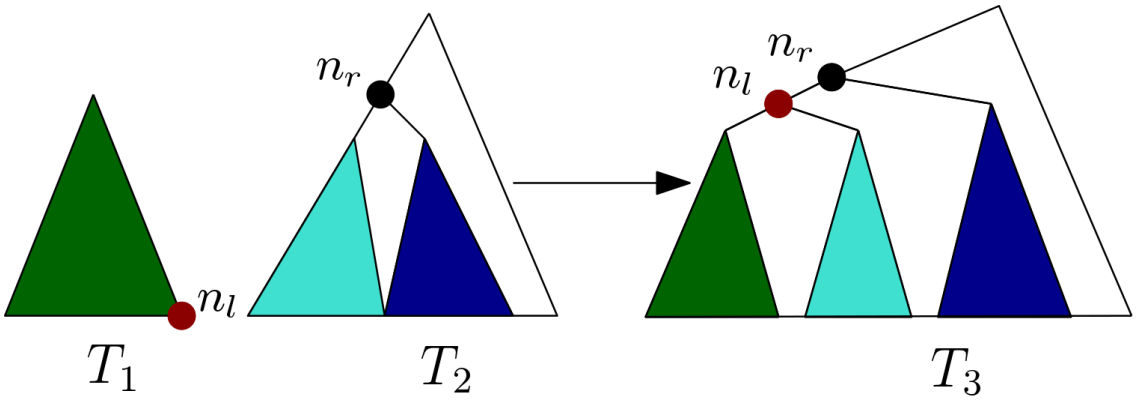


Figure 2.1: A sketch of how two trees are concatenated when the right tree is higher than the left one. Finding n_l and n_r take both at most $O(\log n)$ time and rearranging the children takes constant time. Therefore the total running time is $O(\log n)$ time.

A more efficient but more complicated way is the following: let h be the height of T_1 and assume that T_2 has a bigger height for simplicity. Denote n' as the rightmost leaf of T_1. Now we traverse T_2 h nodes down until node n. Then we move $n.left$ to $n'.right$ and T_1 to $n'.left$. The other around when T_1 is higher will also work. The pseudo code to concatenate them is given below and it covers both cases. A visual example is provided in Figure 2.1. Rebalancing is not mandatory but it is recommended to do.

ConcatenateTrees(T_1, T_2)
  $h_l \leftarrow$ height of T_1, $h_r \leftarrow$ height of T_2
  if $h_l < h_r$ **then**
    $n_l \leftarrow$ rightmost leaf of T_1, split n_l from T_1

$n_r \leftarrow$ the node reached when traversed $h_r - h_l$ to the left from $T_2.root$
$n_l.left \leftarrow T_1$
$n_l.right \leftarrow n_r.left$
$n_r.left \leftarrow n_l$
return $rebalance(T_2)$
else if $h_l > h_r$ **then**
$n_r \leftarrow$ leftmost leaf of T_2, split n_r from T_2
$n_l \leftarrow$ the node reached when traversed $h_l - h_r$ to the right from $T_1.root$
$n_r.left \leftarrow n_l.right$
$n_r.right \leftarrow T_2$
$n_l.right \leftarrow n_r$
return $rebalance(T_1)$
else
$n \leftarrow$ rightmost leaf of T_1 and split n from T_1
Let T_3 be a new tree with n as root.
$n.left \leftarrow T_1, n.right \leftarrow T_2$
return $rebalance(T_3)$
end if

2.1.6 Rebalancing trees

A lot of running times involving trees assume that the tree is height balanced. Height balanced means that every path from the root to the leaf contains approximately $\log n$ nodes. The height of a balanced tree can be out of balance after an insertion, deletion, splitting or a concatenation so then it will be necessary to restore the height balance. The way to do this is by 'rotating' trees using local operations called tree rotations. A tree rotation takes a node and increases one side of its subtree while decreasing the other side. The contents of the subtree of this node remains the same. Every node or leaf n has a height factor h which is defined as $h = n.right.height - n.left.height$ or 0 for leafs and it has to be either -1, 0 or 1. So n has to be restored if $n.h$ is neither of those and it depends on the value what we have to do.

There are two possible rotations: left and right rotation, both are displayed in Figure 2.2. The left rotation decreases the height of P by 1 while increasing the height of Q by 1. It is the other way around for a right rotation. The subtrees α, β and γ remain unchanged. Pseudocode for the rotations can be found in the literature [6].

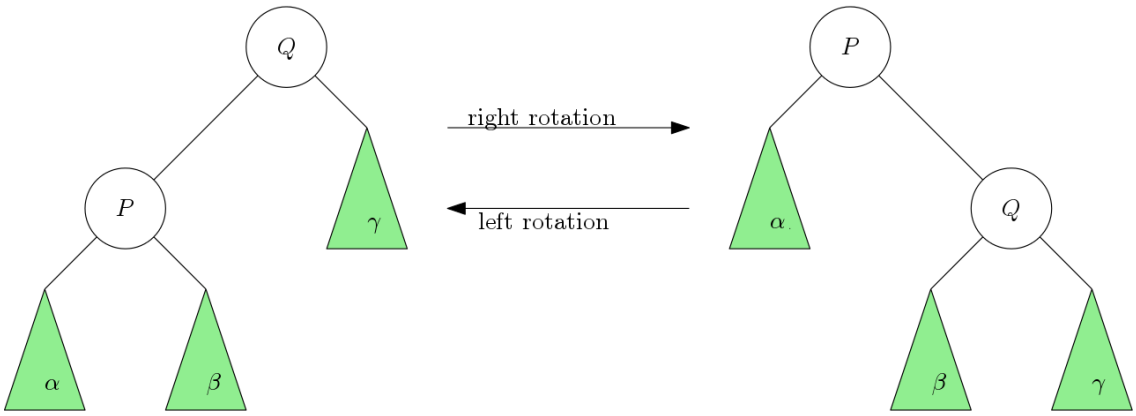


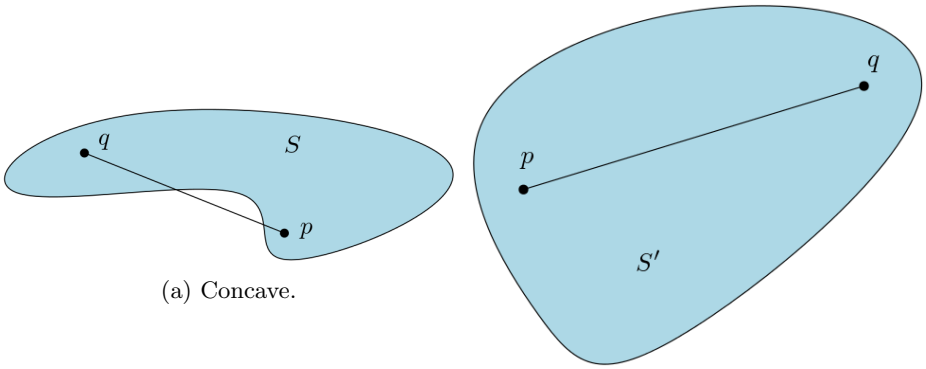
Figure 2.2: Subtrees α, β and γ stay intact while rotating.

2.1.7 Other relevant functions

The function $|X|$ is defined as the size of collection X and $|\emptyset| = 0$. $\lceil x \rceil$ is used as the rounding function. It rounds x to the nearest higher integer of returns x if it already is an integer. The intersection of a list is defined as $X[a, b]$ from the start point a to the end point b. These points could also be replaced by $X[\ldots, b]$ or $X[a, \ldots]$ which should be interpreted as 'from the beginning to b' 'or from a to the end'.

2.2 Computing a convex hull

A subset $S \subset \mathbb{R}^d$ is convex if for any pair of point p, q the line segment $[p, q]$ is completely contained in S. This is visualised in Figure 2.3 for $d = 2$: $[p, q]$ on the left goes through the exterior of S. This makes it *concave* set while S' on the right is a *convex* set: it only contains pairs of points whose line segments go through the interior of S'. In the scope of this project we are limited to planar sets of points (or lines later on) which are always finite. This means two things: that a set $S \in \mathbb{R}^2$ only contains points and that $|S| \neq \infty$. But convex hulls for $d > 2$ also exist.



(a) Concave.

(b) Convex.

Figure 2.3: The difference between concave and convex.

A *convex hull* is defined in the following way [7]: for a finite set of points $S \subset \mathbb{R}^2$, the convex hull of S is the smallest convex set that contains S. Also, the convex hull of S is the intersection of all convex sets containing S. Because the convex hull of S is the smallest possible set, it follows that the convex hull is a subset or equal to S and since S it finite it follows that the convex hull of S is also finite. The points of S contributing to the convex hull are called vertices and a convex hull is represented by its vertices, usually ordered in some sort of order (like anti-clockwise) that enables us to retrieve the line segments of the convex hull by traversing the vertices. An example of a convex hull and how it changes after the set of points increases is showed in Figure 2.4.

So the goal of computing a convex hull is to retrieve a subset $C \subseteq S$ that is as small as possible but still contains S: for every pair of points $a, b \in S$ we have that $[a, b]$ is in the interior of C. There exist a few different algorithms to compute convex hulls with running times varying from $O(n^3)$ or even $O(n^4)$ [13] as the slowest to $O(n \log h)$ as the fastest with h as the size of the output. Each of these algorithms takes S as its input and outputs the convex hull of S.

One of these algorithms [7] is an incremental algorithm called *Graham's scan* that iterates over the input set and extends the convex hull if the new point is on the convex hull. It runs in $O(n)$ time, but the input should be sorted which means that the running time increases to $O(n \log n)$ time in practice and might go up to $O(n^2)$ time depending on the size of the convex hull [13]. The points are sorted *lexicographic*: they are sorted on x, but if the x coordinates is the same for some points then they are sorted on their y coordinates. It works by walking around the border of the

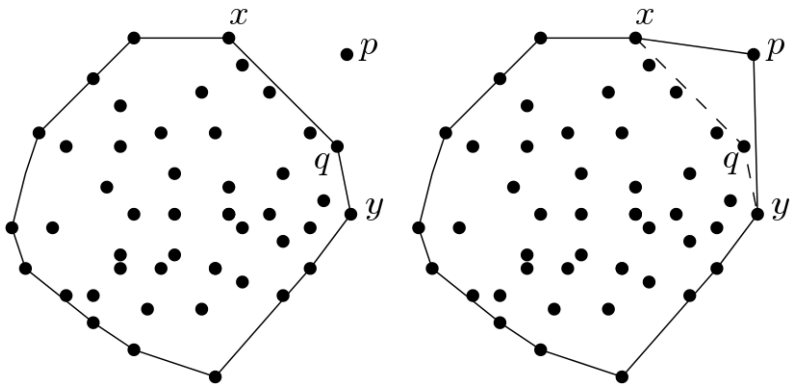


Figure 2.4: An example of a convex hull for a set of points P before and after insertion p into the set. p is in the exterior of P before the insertion, so the convex hull changes after the insertion. The convex hull is updated by removing line segments $[x, q]$ and $[q, y]$ and adding line segments $[x, p]$ and $[y, p]$. The same works for deletions.

polygon and add the vertex to the convex hull if it is a part of it. This is done for the lower and upper hull which are concatenated at the end.

Graham's scan focuses on finding the points of the convex hull. Another algorithm called *Jarvis's march* or *gift wrapping* [13] [7] focuses on the line segments of the convex hull instead. This results in an algorithm that runs in $O(nh)$ time and depends on the size of the convex hull h. It starts at a point that is certainly on the convex hull, so the lowest point or the right most point for example. Then it 'walks around' the convex hull (hence the name) and tracks all the points of the convex hull. Because it depends on the output size it might not be the most efficient algorithm to choose because it can get up to $O(n^2)$ time in the worst case.

Another interesting algorithm is *Chan's algorithm* [4] which is a combination of Graham's scan and Jarvis's march. Suppose the size of the convex hull h is known. First we divide the point set in n/h subset, run Graham's scan over these subsets, and then we apply gift wrapping to these results. The first step takes $O\sum_{i=1}^{n/h} O(h \log h) = h * O\sum_{i=1}^{n/h} O(\log h) = h * \frac{n}{h} O(\log h) = (n \log h)$ time and the second part takes $O(n \log h)$ time as well. If we know h, which is often not the case. When h is unknown we have to change the first step: suppose we iterate $t \in \{0, 1, 2, \dots\}$ and run the Chan's hull using $\min(n, 2^{2^t})$ as h. We continue until a correct convex hull is computed and the running time now evaluates to $O(n \log h)$ because the last iteration of t would be $t = \log(\log h)$.

2.3 Duality of the plane

A point and a line have two parameters: a point has an x and an y coordinate and a line has a slope and the y-coordinate where it intersects the y-axis at $x = 0$. Using these parameters it is possible to map a point into a line and a line into a point. This is called 'duality'. The dual of a point $p := (x, y)$ is defined as the line $p^* := (y = ax - y)$ and the dual of a line $l : y = ax + b$ is defined as the point $l^* := (a, -b)$. The dual of a vertical line does not exist.

These duals have a few properties [7] and are illustrated with the help of examples in Figure 2.5:

- The incidence is preserved: p is on l if and only if l^* is on p^*;
- The order is reversed. If point p lies above line l, then p^* passes below point l^* in the dual plane. Examples are the red line l goes through the green dot in the primal plane, but l^* is on the dual line of the top green dot and the cyan dot and purple line;
- Vertical distances between p and l are preserved for p^* and l^*. The best example of this are the green dots and their duals;

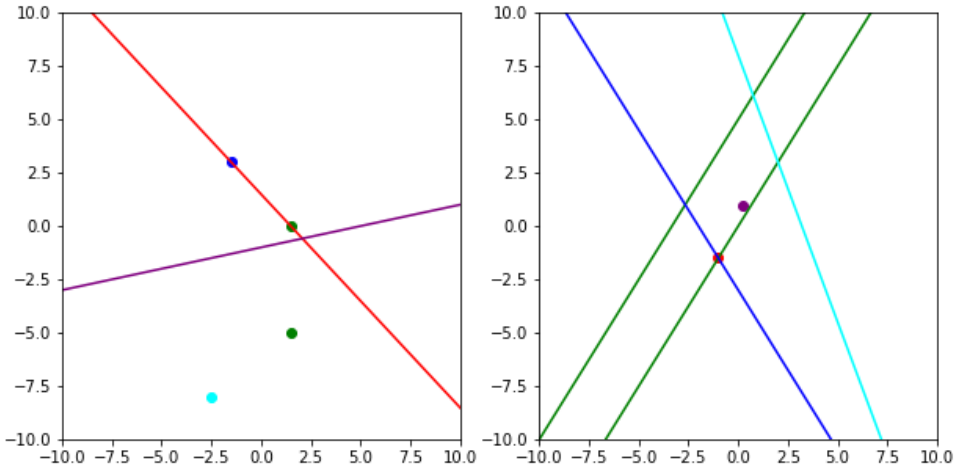
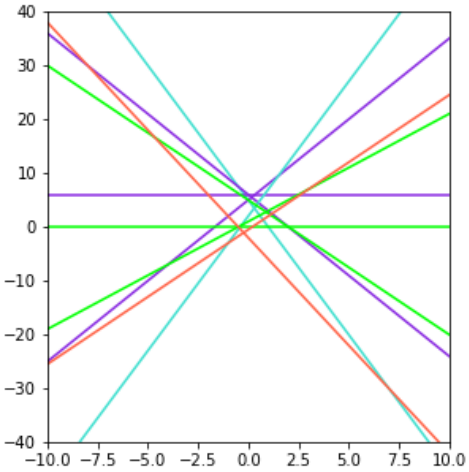


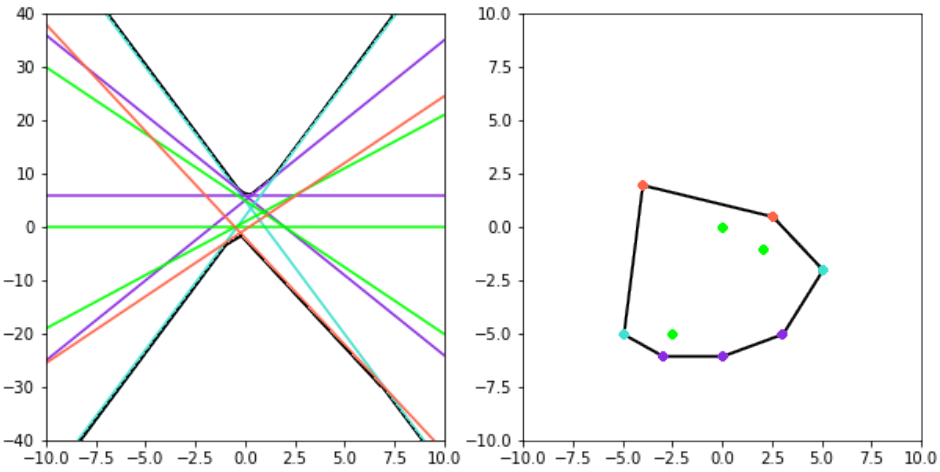
Figure 2.5: On the left we have an example of points in the primary plane and we see their dual lines on the right. The red points $p3, p6, p7$ share the same x-coordinates which results in parallel dual lines. The intersection of lines represents the line through their dual points.

- Duals are self-inverse: $(p^*)^* = p$ and $(l^*)^* = l$;
- The intersection point l^* of points p_1 and p_2 is the dual of the line l that goes through p_1 and p_2.

Duality can be used to maintain the envelope of a set of lines L. First we take the duals of every $l \in L$, denoted as L^*, and maintain the convex hull of L^*. The lower hull of L^* corresponds to the upper envelope of L and the upper hull corresponds to the lower envelope because of the reversing of the order. This is visualised in Figure 2.6. The top plot contains a set of lines L in the primal plane. The light blue lines are the first and last lines on both the lower and upper envelope, which corresponds to the leftmost and rightmost points on the convex hull. The lines on the upper envelope are painted in purple and correspond to the lower hull of L^*. The red lines correspond to the upper hull of L^*. All duals of lines that do not contribute to an envelope (coloured in green) are contained in the interior of the envelope of L^*. This way of maintaining envelopes is a lot easier as the Overmars and van Leeuwen algorithm but this is covered in Chapter 4 to present a complete picture of ways to maintain envelopes dynamically.



(a) The original set of lines L



(b) The envelopes of L in the primal plane and the convev hull of L^* in the dual plane.

Figure 2.6: An example of duality and envelopes. The envelopes and convex hull are drawn in black lines

2.4 CGAL libary

The Computational Geometry Algorithms Library or *CGAL* in short is a library containing a wide variation of packages to solve mathematical problems. Most of them are aimed at geometric problems like: convex hulls, envelopes, operations on polygons, 3D surfaces, triangulations, processing or generating meshes (most of these problems have 2D and 3D versions). But CGAL also provides various complex data structures and algebraic operations.

CGAL is used in this thesis for a few reasons among others: the representations of points and lines, operations on lines (intersections, finding the y-value at a x-coordinate, derivatives), computing intersections between circles and Delaunay triangulations. Polynomials is also an interesting package that might be useful to compute intersections between pseudo-lines. This is some future work however (Chapter 7).

2.4.1 Computing a convex hull

There are a few algorithms to calculate the convex hull of a set of points and CGAL [1] has implementations for some of them. None of these algorithms can be used for dynamically maintaining a convex hull though since every change of the point set requires a recomputation of the convex hull. This is why these algorithms are not evaluated in practice (Chapter 6).

2.4.2 Delaunay triangulation

Triangulations of sets of points are important geometric constructions and there is one type that is especially interesting: *Delaunay triangulations* because it can be used to dynamically maintain a convex hull. Delaunay triangulations are closely related to Voronoi diagrams and Figure 2.7 shows an example of these. For more details about Voronoi diagrams and Delaunay Triangulations, see the book by de Berget al. [7]

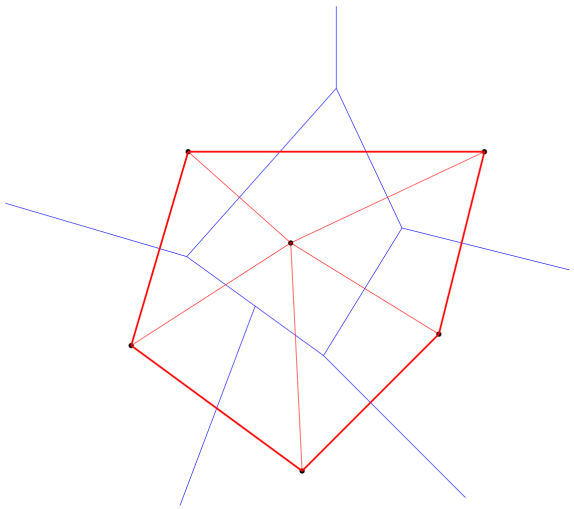


Figure 2.7: A Voronoi diagram in blue of a set of points and its coresponding Delaunay triangulation in red. The dark red edges for the edges of the convex hull.

A Delaunay triangulation can be used to maintain a convex hull using the fact that the vertices on the boundary of the triangulation form the convex hull of the set of points contained in the triangulation. So the only thing to do is traverse the outer edges of the Delaunay triangulation after an insertion or deletion to update the convex hull. In fact, CGAL even provides this as

[1] documentation is found here.

an example of maintaining a convex hull using the Delaunay triangulation [2]. Insertions have a running time of roughly $O(\sqrt{n})$ when the points are uniformly distributed but the worst case running time is $O(n)$ time. Removing a vertex from the Delaunay triangulation takes $O(d^2)$ time (unless $d \leq 7$) where d is the degree of the vertex but first we have to find this vertex of course (it is not clear in the literature how much time this takes). The practical running times of the Delaunay triangulation will be compared with the Overmars and van Leeuwen structures to see how practical both are. See Chapter 6 for more details.

[2] Documentation is found here.

Chapter 3

A dynamic data structure for convex hulls

From this point it is assumed that the data structures and algorithms described in Chapter 2 are clear to the reader. In this chapter we will discuss the implementation of the algorithms and data structures and other practical sides.

3.1 Dynamically maintaining convex hulls by Overmars and van Leeuwen

The problem with the algorithms to compute convex hull we saw is that they cannot be adapted to allow insertion or deletions from the point set. Insertions could be possible, but this is not the case for deletions since no information about the interior of the convex hull is maintained. This was the motivation for Mark Overmars and Jan van Leeuwen to develop a data structure that allows maintenance of the convex hull after insertions or deletions in $O(\log^2 n)$ time [11]. It is designed using a binary search tree that store all the points in the point set in its leafs and each internal node is augmented with the convex hull of the points rooted in the subtree of this node. Each convex hull is composed of two separate *subhulls*: the *upper hull* and the lower part of the convex hull, the *lower hull* which are essentially the same thing but mirrored, see Figure 3.1. It can be retrieved by concatenating the lower and upper hull of the root.

3.1.1 Mathematical representation

The points in the tree are ordered on there x-coordinates and these coordinates are assumed to be unique for simplicity. The most optimal way to store subhulls is using trees because the algorithm depends on an elaborate binary search, but ordered lists are also possible to use although this would increase the running time.

The lower and upper hull are computed in the highly similar way since the only differences are the conditions to find the point at the convex hull to split or merge. So for simplicity only the computation of the lower hull is described.

A convex hull can be split into two parts: the upper and lower *subhull* (Figure 3.1). Let C be a convex hull, $a \in C$ the leftmost point and $b \in C$ the rightmost point. The upper subhull $U \subseteq C$ are the points encountered if we start traversing the points of C in counter-clockwise direction starting at b and ending at a. The lower subhull D are the points encountered if we start traversing C in clockwise direction from b to a. Note that a and b are both on the upper and lower hull.

Let A and B be subhulls (upper and lower hulls work both) such that all the points in A are strictly left of B and we want to merge them into one subhull. What we need to do is to find a so called 'bridge' between the hulls. This bridge is a line segment starting in $u \in A$ and ending in

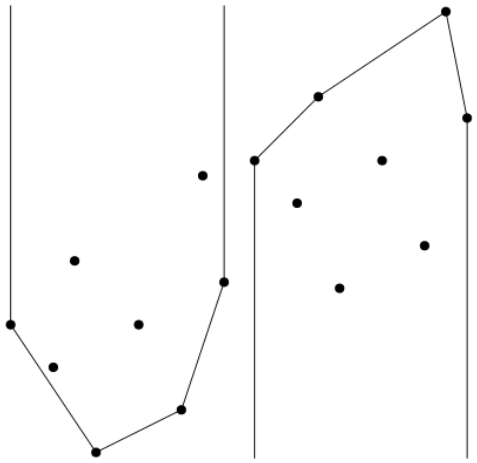


Figure 3.1: The lower hull (left) are the points on the convex hull encountered going clockwise from the rightmost point to the leftmost point on the convex hull in clockwise direction. The upper hull (right) are the points on the convex hull encountered from the rightmost point to rhe leftmost point in counter-clockwise direction.

$v \in B$ such that all the points left of u in A, u, v and all the points right of v in B form a subhull. Mathematically this is defined as $C = A[\ldots, u] \cup B[v, \ldots]$. The algorithm designed by Overmars and van Leeuwen is made to find u and v.

3.1.2 The data structures

The data structure consists of a special binary search tree where points in the plane are stored in the leafs, sorted on their x-coordinates. It is special because usually points are also stored in internal nodes. And on top of that every internal node is augmented with two data structures to store the upper and lower hull for the left and right subtree rooted at this node. This augmented data structure should be a tree for the most optimal running times, but it could also be another data structure like a list.

So, there are two data structures to be considered: a binary search tree where the points are stored in the leafs which will be referenced as the **main tree** and a data structure that will be augmented in the internal nodes of the main tree. This is originally called a concatanable queue but this is confusing since it does not work like a queue so it is renamed the **augmented structure** from now on.

The main tree should have the properties and attributes. Originally, the algorithm was given as a for loop traversing from the root down to a leaf and then traversing back to the root. This traversal was possible because every node in the main tree had a pointer to its parent. However, this is not necessary anymore once the for loop is refactored to a recursive traversal.

1. α contains pointers to its left and right child $\alpha.left$ and $\alpha.right$ and a pointer to the leaf in the left subtree with the highest x-coordinate $\alpha.lmax = \max(\alpha.left)$.

2. Each node has two augmented structures to store the upper and lower hull $\alpha.Q_d$ and $\alpha.Q_u$ respectively. The leafs are also augmented with this in practice and initially contain the point they represent. Because of how the tree is used in the algorithm, only the parts of $\alpha.Q_d$ and $\alpha.Q_u$ that do not contribute to the parent of $\alpha.Q_d$ and $\alpha.Q_u$ respectively are stored.

3. The points are sorted x-coordinates. The points were originally sorted on the y-coordinates. This does not result into a convex hull, but something similar and rotated 90 degrees.

4. The points $a \in Q_l$ and $b \in Q_r$ for $Q_l = \alpha.left.Q_u$ and $Q_l = \alpha.right.Q_u$ are the points that form the new subhull between Q_l and Q_r. This pair of points is stored for both the lower hull in $\alpha.B_d$ and for the upper hull $\alpha.B_u$. These pairs are the bridges.

5. A leaf α has a field v to store a point.

An insertion or deletion of a point p are highly similar to each other. It works as follows: we start with a binary search in the main tree using the x-coordinates of the points. On the way back we split $n.Q_u$ and $n.Q_d$ according to their bridges and merge the results to the hulls of $n.left$ and $n.right$ for every internal node n we encounter on the way down. After we found the leaf l containing p (in case of a deletion) or where p will be inserted, we insert or delete p. Now we start traversing back up to the root while and restore the subhulls at the internal nodes we encounter including the root (these are the same as on the way down). See Figure 3.2 for a step by step example of an insertion.

The structure for the augmented structure has to support the following actions. It should be possible to split one into two, or merge two into one. There are several ways to implement the augmented structure. The simplest one is using a list or *vector* which is already provided by C++. However, merging and splitting a *vector* is not efficient so this will slow the algorithms down a lot. Searching in a vector can be done in $O(\log n)$ time by implementing a binary search and the easiest but slowest way is just using a linear scan in $O(n)$ time. The best but more complex way to implement the augmented structure is using a binary search tree (with points stored in the internal nodes), especially if it is height balanced. This will bring the running times of the required actions to $O(\log n)$ time. This is the most complex though hence a *vector* was used initially in this thesis project. Using multiple data structures also allows us to see the impact on the running times in practice.

The augmented structure does not require the points to be stored only in the leafs, when using a tree. So every node or leaf stores a point in the field p.

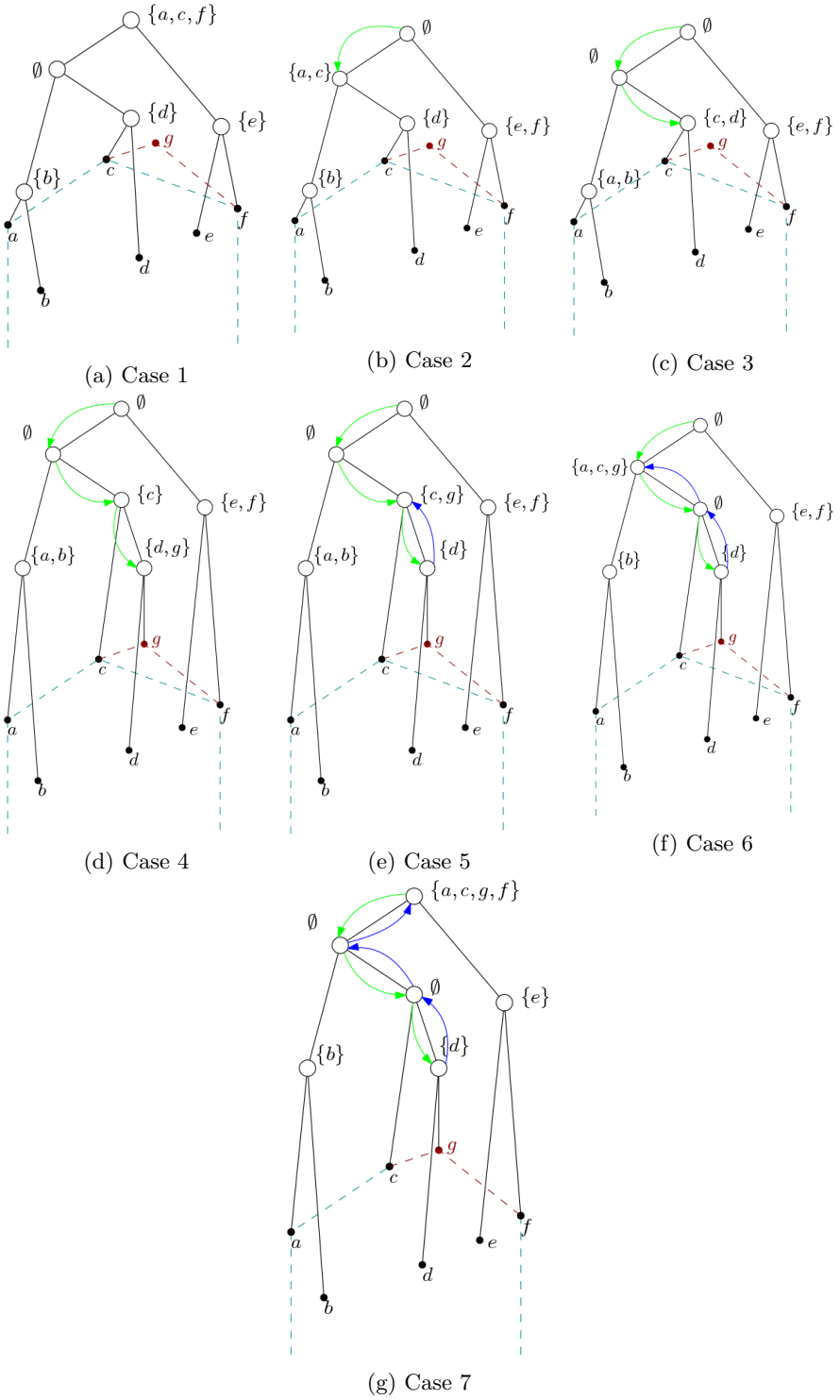


Figure 3.2: A step by step example for inserting g into $\{a, b, c, d, e, f\}$ starting from the topleft. The leaf containing d in panel 4 is replaced by a node n such that $n.left$ contains d and $n.right$ contains g. The green arrows indicate the traversal down while the the blue arrows indicate the traversal up. Removing g is also displayed here if you start at the bottom panel and traverse the arrows in reversed direction. Although this example only shows the upper hull, computing the lower hull is equivalent.

3.1.3 Combining two hulls into one

The algorithm to compute the convex hull is based on this recursive structure. Computing subhull $\alpha.Q_d$ is based on the assumption that the subhulls $\alpha.left.Q_d$ and $\alpha.right.Q_d$ are known. Finding a hull is defined recursively. Every node in the main tree represents the convex hull using the upper and lower hull of the points that are stored in the subtree rooted in this node. For leafs this means that the convex hull is just the point itself.

Let α be an internal node and assume that the upper and lower hull for $\alpha.left$ and $\alpha.right$ are known. So if you want to know the complete convex hull, you build it up starting at the leafs and finish at the root. Then you take the union of the lower and upper hull of the root and you have the convex hull. How this building exactly works is described in this section.

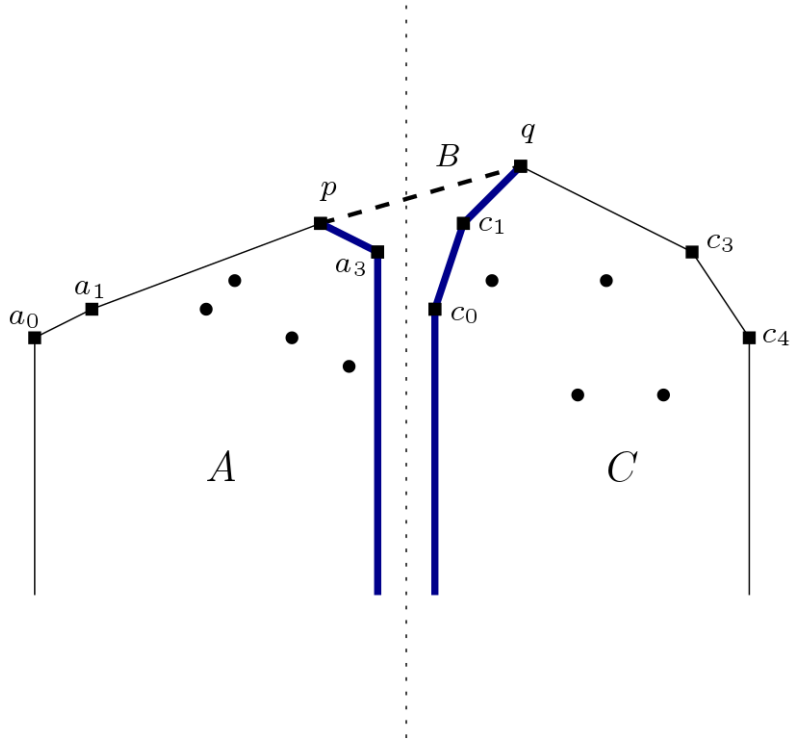
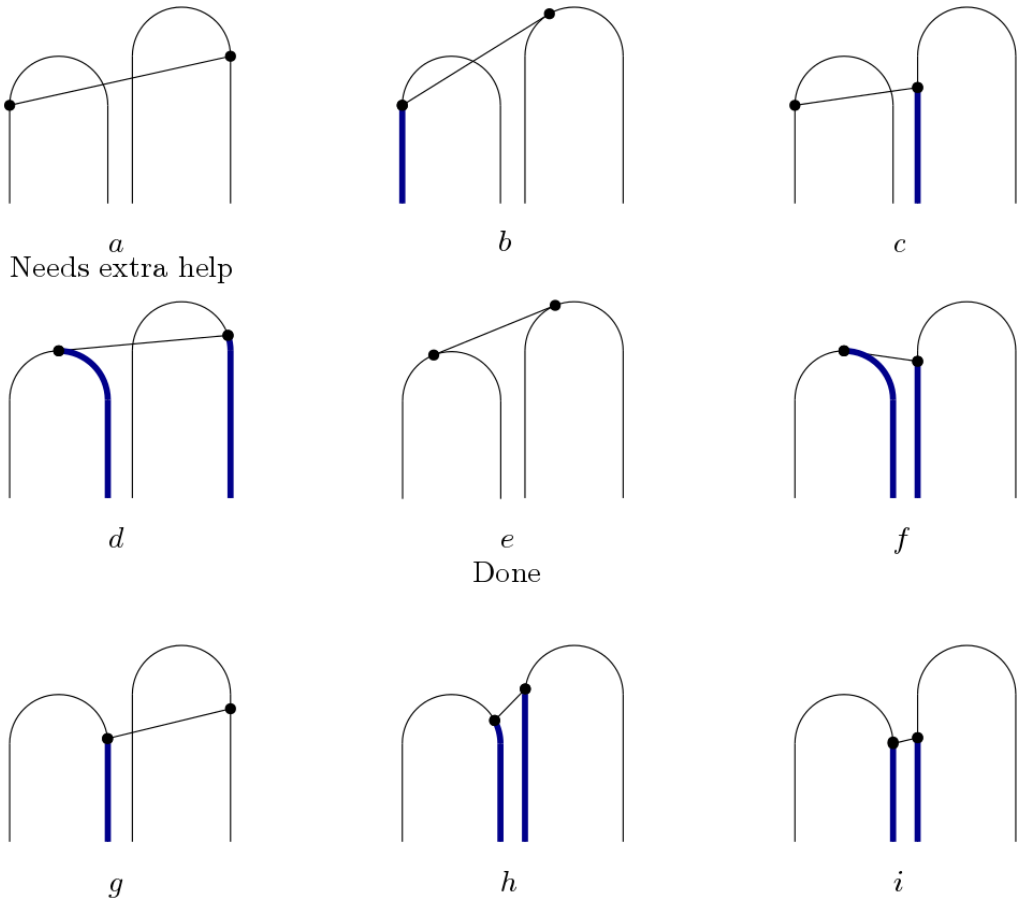


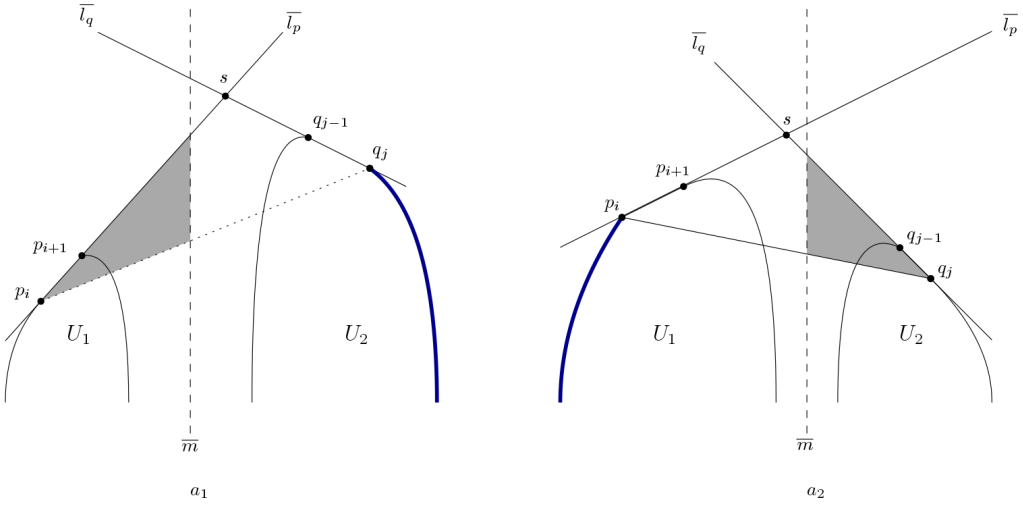
Figure 3.3: The result of looking for the bridge of two hulls in a point set $P = A \cup C$. The points contained in the convex hull are represented as squares, the points that are in the interior as circles. The subhull $A' = \{a_0, a_1, p, a_3\}$ of A and subhull $C' = \{c_0, c_1, 1, , c_3, c_4\}$ of C are represented with a line. The bold blue parts of the line will be discarded by the bridge B and those points will become a part of the interior. The new upper hull will be $(A' \setminus \{a_3\}) \cup (C' \setminus \{c_0, c_1\}) = \{a_0, a_1, p, q, c_3, c_4\}$.

We see an example of how the bridging works in Figure 3.3. Subsets A and C can be separated by a vertical line, as well as subhulls A' and C' since $A' \subseteq A$ and $C' \subseteq C$.

However, u and d are not known, so we need to find them. We do this by taking arbitrary points $p \in A'$ and $q \in C'$ and creating bridge $\overline{pq}$. It is likely that $\overline{pq}$ will not give us the correct bridge, but we do know where to continue the search depending on where $\overline{pq}$ intersects A' and C'. Figures 3.4a and 3.4b show us which situations can occur when testing the bridge. We reduce the search space by half every time we investigate a new $\overline{pq}$ because if the augmented structure is implemented as a binary tree than it is basically a binary search. The figures only show the cases for the upper hull, but the cases for the lower hull are exactly the same, but then horizontally mirrored.



(a) These are the possible cases we may encounter while making a bridge of hulls of set U_1 and U_2. U_1 is always the one the left, U_2 on the right. Case e is the one we want, then we are done searching. A bold blue line represents the part of the hull that can be discarded which means we will search in the other direction. Case a is special, it needs some help to determine what needs to happen.



(b) The two possible subcases of case a.

Figure 3.4: An overview of all the case distinctions.

Figure 3.4a shows the possible cases we can encounter when searching for the bridge and all are straightforward. However, case a has two subcases a_1 and a_2 shown in Figure 3.4b, and these exist because it is not clear on first sight which direction we need to go. The case distinction is based on the line $\overline{l_p}$ through the current left point p_i and the point p_{i+1} next to it on the right, the line $\overline{l_q}$ through the current right point q_j and the point q_{j-1} next to it on the left and a line m in the middle of the hulls. Let s be the intersection of $\overline{l_p}$ and $\overline{l_q}$. If s is right of $\overline{m}$, then we are in case a_1 and if it is left of $\overline{m}$ then we are in case a_2. The blue part of the hull can be ignored and the points for the bridge are in the grey area. Instead of checking on which side s is, in practice the y-values of $\overline{l_p}$ and $\overline{l_q}$ are compared. In case a_1 go left on U_2 and in case a_2 we go left on U_1.

This pseudo-code describes how a bridge can be found for an internal node α. It is assumed that $\alpha.left.Q_d$ and $\alpha.right.Q_d$ are already computed. The same holds for the upper hulls. Only the lower hull is covered here for simplicity.

findBridge(α) ▷ α is the pointer to a node from the main tree.

1: $lPointer \leftarrow \alpha.left.Q_d.root, rPointer \leftarrow \alpha.right.Q_d.root$
2: **while** *true* **do**
3: $slope \leftarrow \frac{b.y-a.y}{b.x-a.x}$
4: $iL \leftarrow DetermineStateDown(lPointer.p, slope)$
5: $iR \leftarrow DetermineStateDown(rPointer.p, slope)$
6: **if** Bridge is found (e.g. iL, iR = case e) **then**
7: *break loop*
8: **end if**
9: $lPointer \leftarrow lPointer.left$ or $lPointer \leftarrow lPointer.right$; $rPointer \leftarrow rPointer.left$ or $rPointer \leftarrow rPointer.right$ ▷ Depending on case from 3.4a or 3.4b
10: **end while**
11: $a \leftarrow lPointer.p$
12: $b \leftarrow rPointer.p$
13: $\alpha.B_d \leftarrow (a, b)$
14: $Q_1 \leftarrow \alpha.left.Q_d[\dots, a]$ ▷ Including a to Q_1
15: $Q_2 \leftarrow \alpha.right.Q_d[b, \dots]$ ▷ Excluding b to Q_2
16: $\alpha.Q_d \leftarrow Q_1 \cup Q_2$

3.1.4 Insertions and deletions

Restoring the convex hull is done using the procedures *DOWN* on the way down and *UP* on the way up. *DOWN* split the convex hulls at a node and puts the results back in its children and *UP* restores the convex hulls on the way up. These procedures are using in the recursive functions *insert* and *delete*. The pseudo-code is given for *insert*, but deleting a point is almost the same. This pseudo-code assumes that $p.x$ does not occur yet in the data set.

insert(α, p) ▷ Start with $\alpha = root$ for inserting a point p

1: **if** α is a leaf **then**
2: Insert p into α
3: **else**
4: *DOWN*(α)
5: **if** $p.x \leq \alpha.lmax.p$ **then**
6: $\alpha.left \leftarrow insert(\alpha.left, p)$
7: **else**
8: $\alpha.right \leftarrow insert(\alpha.right, p)$
9: **end if**
10: **end if**
11: *UP*(α)
12: **return** α

The procedure *DOWN* splits $\alpha.Q_u$ into P and Q using $\alpha.B_u = (a, b)$ such that $\{\beta \in P : \beta.x \leq a.x\}$ and $\{\beta \in Q : b.xleq\beta.x\}$. Then it concatenates this to its children: $\alpha.left.Q_u \leftarrow P \cup \alpha.left.Q_u$ and $\alpha.right.Q_u \leftarrow \alpha.right.Q_u \cup Q$. The bridges remain unchanged but will be overwritten using *UP*.

UP is responsible for updating the convex hull and the bridges for node α using calls to *findBridge*.

Inserting p in a leaf n means that we create a new node n' for p and set n' as the left or right leaf of n, depending on the value of p. When deleting p we halt at n where p is stored in one of its children, remove the child and start traversing back starting at n.

Now the convex hull can be retrieved by $root.Q_u \cup root.Q_d$. Note that the left-most and right-most point of the convex hull is stored twice, so this is something to keep in mind. This union is simply traversing both subhulls. The order of the traversal influences the order of the points, so starting with the upper hull gives the points in clockwise direction and starting with the lower hull gives the points in anti-clockwise direction.

3.1.5 Running time

Updating a subhull while inserting or deleting a point consists of various parts and both contribute differently to the running time. The first part *DOWN* runs in $O(\log n)$ time. Let k be the size of the convex hull and the tree to traverse be height balanced. While traversing down, *DOWN* splits the hull $\log n$ times and reduces the size of the hull k by half every step. In the worst case we have that all the points are on the convex hull, so $k = n$ at the root. This evaluates to $k_i \in K = \{n, \frac{n}{2}, \frac{n}{4}, \ldots, 1 = 2^{\log n}, 2^{\log(n/2)}, 2^{\log(n/4)}, \ldots, 2^0\}$ where K is the set of the worst case sizes for the hulls in the traversed path. It sums up to $O(\log^2 n)$:

$$\begin{aligned} &\sum_{i=0}^{\log n} O(\log k_i) \leq \\ &\sum_{i=0}^{\log n} O(\log(2^i)) = \\ &\sum_{i=0}^{\log n} O(i) = O(\log^2 n) \end{aligned} \tag{3.1}$$

UP is bit more complex: while it traverse upwards it searches for the bridge, splits at the bridge and merges the results. All of these things take $O(\log n)$ time individually, and it happens every step on the way up for the same set of K as for *DOWN*. It evaluates to the same running time as *DOWN*:

$$\begin{aligned} &\sum_{i=0}^{\log n} O(3 \log k_i) = \\ &\sum_{i=0}^{\log n} O(\log k_i) \leq \\ &\sum_{i=0}^{\log n} O(\log(2^i)) = \\ &\sum_{i=0}^{\log n} O(i) = O(\log^2 n) \end{aligned} \tag{3.2}$$

The total running time for an insertion or deletion is $O(2 \log^2 n) = O(\log^2 n)$ time while using trees as the data structure for envelopes. It is also possible to use sorted lists for the envelopes

but this will increase the running time as splitting, merging and finding the bridge all take $O(n)$ time now. For the same $k \in K$ while traversing down and up it becomes:

$$\sum_{i=0}^{\log n} O(k_i) \leq \sum_{i=0}^{\log n} O(2^i) = O(n) \tag{3.3}$$

The running time for trees is $O(\log^2 n)$ time and for ordered lists $O(n)$ time which is obviously worse. So why bothering with lists? Trees are the most efficient to work with, but also more complex to start with. There is no default implementation provided for C++ and the trees used are adapted so the trees and their algorithms to split and merge had to be implemented first before starting on the convex hull. So if there is a bug in the code, it could be in the trees or in finding the bridge. Starting first with lists eliminates the possibility that there are bugs in the data structure because these implementations are given. This made it possible to correct the bridging algorithm a lot faster and the lists were replaced by trees after this step.

3.2 Implementation details

These data structures and algorithms have been implemented twice before. Once in C++ [1] for another Master Thesis project in 2007 and one in Java [2]. The source code for both implementations were online available but unfortunately it turned out after some tests that the C++ version did not work anymore. Insertions worked partly and deletions not at all and it was slow if it worked so this was not a good base to start building on. The Java version worked really well, it was fast and correct. Only the upper hull was implemented initially, but adding the lower hull was no issue. This was translated into C++ and the current class structure is based on this. So the plan was now to translate this Java code to C++ which had its rough edges and it basically turned out to be reimplementing most things. Especially the details of the trees had to be implemented again but the bridge algorithm was very usable as a base.

3.2.1 Design choices and structure of the code

The class structure consists of five different classes, ignoring the CGAL classes used. The main class is *ConvexHull* which has functions to insert a point, remove a point, retrieve the convex hull or the separate hulls separately, implementations of *DOWN* and *UP*. The main tree is composed of nodes of the type *CNode* and is stored in *ConvexHull*. *CNode* has pointers its left child, right child and maximum leaf in the left child. It also contains both subhulls stored as *SubHull* and their two bridges. It also contains a field to store a point which is only used when it is a leaf.

SubHull contains functions to find the bridge of two subhulls and specialises *Base* which contains the binary search tree *AVLTree* for the subhull and a list (*std::vector*) to store points. Depending on the version of the convex hull, either the tree or the list is used. Despite the name, *AVLTree* is not an AVL-tree (a height balanced tree). It is named this way, but height balancing is never implemented.

3.2.2 Completing the convex hull implementation

The data structure is rather complicated, so it is divided into multiple parts to implement it correct and as fast as possible.

[1] Implementation can be found on Google Code.
[2] Implementation can be found on GitHub.

Using a list: *vector*

C++ has several implemented data structures to offer and all of them have different properties, but the one we need has to support the following operations: insertions, deletions, keeping it sorted, merging and splitting. Using a list would make the most sense. There are several list-like structures: *vector*, *forward_list* and *list*. The last two are linked lists, while *vector* works like an array with a variable size. *vector* turned out to be the best option at first sight, because it is the only one supporting random access which is the easiest to use in this setting. A *vector* is not a sorted list by default, but it can be by carefully choosing how to merge and split it such that it is always ordered.

Using this in the algorithm works roughly the same as the pseudocode for trees given in Chapter 2. But instead of a binary search in a tree, it now uses a linear scan over an ordered list starting at the middle or beginning of the list. I would also be possible to implement a binary search on the list to reduce the running time a bit, but it does not lower the asymptotic running time as splitting and merging still takes linear time. Procedures *DOWN* is also a bit different simply because now it has to split lists instead of trees.

Using a linked list

Another data structure that can be interesting to use is the linked list, implemented as *list*. Merging two linked lists takes up to $O(n)$ time, like *vector* does, but splitting a linked list takes only constant time (but it still has to copy the elements to split) [3] which is a big advantage over the $O(n)$ time for splitting a *vector*. The only downside is that it lacks random access, so it is only possible to perform a linear scan while it should be possible to implement a binary search for the *vector* based structure. The data structure would be a bit more complex in order to achieve this constant splitting time. Since there is no random access, we have to store pointers to the places in the list where we need to split and this turned out to be too tricky in practice to be useful.

Using a tree

This is exactly like the pseudocode describes, using merge and split functions from Chapter 1. Implementing trees as the structure for convex hulls was done the last.

3.2.3 Height balancing the trees

Unfortunately it was not possible to implement tree rotations to height balance trees due to time limits. It will work anyway regardless if the tree is height balanced or not but it might slow some inputs down a lot. This is discussed more in detail in the results (Chapter 6).

3.2.4 Correctness

A visual debugging tool was made to inspect the hulls and spot where things go wrong. A more formal way of testing was made after everything seemed to work and this is discussed in Chapter 6. A proof of correctness is given in the original paper [11].

The approach using *vector* is slow and the one using linked lists is even worse, but they work. It works the same as the approach using trees, aside from the running time and splitting and concatenating the data structures. We have to assume that *Lhull* and *Rhull* are correct subhulls and correctly ordered, but this does not really differ from the original assumptions. Since the subhulls are correct, the points $a \in Lhull$ and $b \in Rhull$ exist such that a new subhull can be formed using the bridge from a to b. These points exist, independent of the underlying data structure. The only thing to mind is that the splitting and concatenating functions for trees should work correct and be tested, since those functions for the *vector* are available by default.

[3] Documentation about C++ containers can be found here.

3.3 Related work

Overmars and van Leeuwen were the first to propose a way of dynamically maintaining a convex hull, but it is not the only one as this topic has been studied a lot before. We have seen in the preliminaries (Section 2.4.2) that Delaunay triangulations can be used for this [4], but there is more. A new algorithm for planar point sets with $O(\log n)$ amortized time was introduced by Gerth Stølting Brodal and Riko Jacob [2]. They also present some operations on convex hulls in $O(\log n)$ time.

At least one other algorithm exists as well, made by Franco Preparata in 1979 [12], but it is restricted to only allow insertions because it was not clear how information about the interior of the convex hull should be maintained. Insertions are performed in $O(\log n)$ time in the worst case, so processing an entire set takes $O(n \log n)$ time. Expanding the convex hull is no problem, but removing a vertex and thus shrinking the convex hull is because then a point from the interior could be on the new convex hull, but it is impossible to know which point because the interior is not maintained. So Overmars and van Leeuwen were the first ones to propose a data structure that allows deletions as well.

Computing convex hulls or envelopes is also possible in Python. The package *scipy*[5] offers some functionality to computes convex hulls incrementally, the intersection of half spaces and Delaunay triangulations. [14]

Overmars and van Leeuwen present applications of dynamic convex hulls and this is extended by for example Timothy Chan [5] who presents three more applications related to dynamic convex hulls or envelopes with some extensions to 3D versions.

In the last years there has been some work on dynamically maintaining *geodesic* convex hulls in the plane. This means that the plane contains 'barriers' where a segment of the convex hull has to go around, but in such a way that this segment has the shortest possible length. Ishaque and Tóth [8] introduced a semi-dynamic solution that only supports insertions of barriers and point deletions in $O((m+n)\text{polylog}(mn))$ time for m barriers and n points. Eunjin Oh and Hee-Kap Ahn [10] present data structures and algorithms that is fully dynamic for geodesic convex hulls which means it also allows deletions of barriers and insertions of points.

[4] Delaunay triangulations are also implemented in Python.
[5] Documentation about convex hulls can be found here. Documentation about envelopes here.

Chapter 4

A dynamic data structure for envelopes

In the previous chapter we made the first big step: implementing the Overmars and van Leeuwen method for maintaining convex hulls dynamically. The next step is to implement their method for envelopes. This is step is not entirely necessary, but we will look into it now to have a complete picture about maintaining envelopes dynamically. It turned out unfortunately that the implementation is a lot more complex as expected.

4.1 What is an envelope?

Let L be a set of lines in the plane. All these lines are infinitely long, so at any x-coordinate there is a line of L which is the lowest. Consider the following question: 'for an $x \in \mathbb{R}$, which line $l \in L$ is the lowest at x?'. This line has a range on the x-axis where it is the lowest which means that a segment of this l contributes to the envelope. This question can be extended to the entire x-axis and this results in a sequence of line segments which is called the *lower envelope* of L (blue in Figure 4.1). Each line of the envelope is a part of a line in L and each line in L contributes at most once to the lower envelope. The same question can be mirrored and this results in the *upper envelope* (red in Figure 4.1).

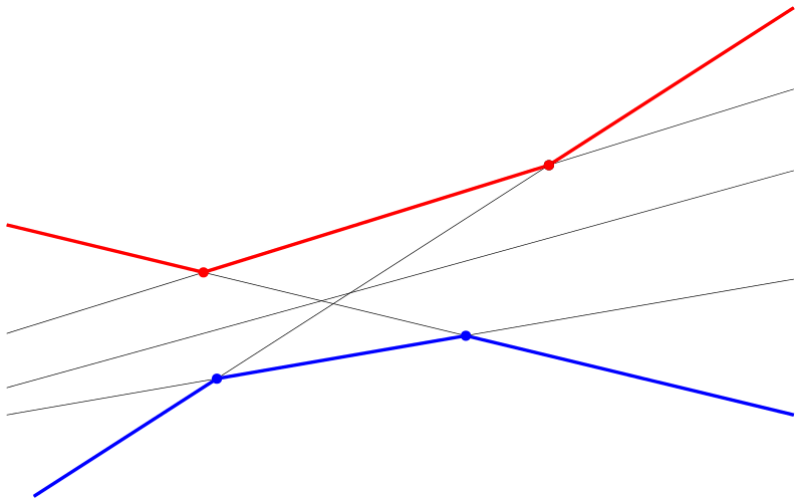


Figure 4.1: An example of an upper envelope (in red) and a lower envelope (in blue).

4.1.1 Computing an envelope

A lower envelope has a few properties. It is a x-monotone collection of line segments, covers the entire x-axis and the slope of the line segments are increasing from left to right. The fastest descending segment is on the left, the fastest ascending segment on the right. Every line can contribute at most once to the envelope.

The Overmars and van Leeuwen algorithm only works for straight lines, but it is also possible to have an envelope of different kind of shapes like circles or curves and line segments. This can be computed using CGAL, but not dynamically. This algorithm provided by CGAL is able to compute the convex hull of a mixed set of polygons like in Figure 4.2. We will see in a bit that an envelope can also be maintained using duality and the convex hull, but this also works only for lines.

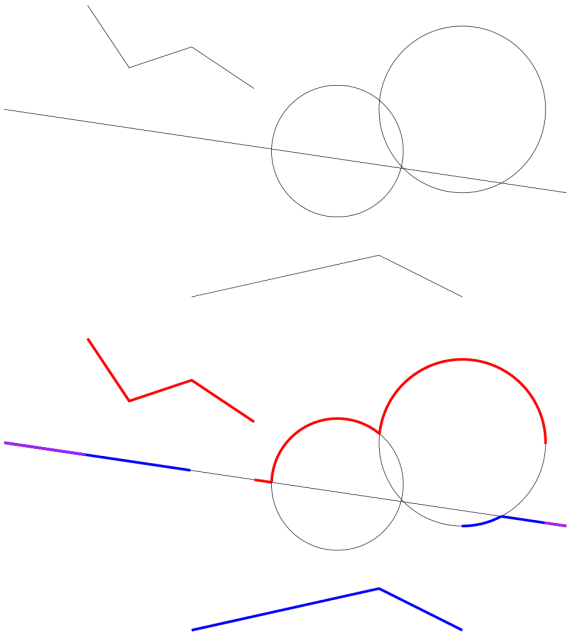


Figure 4.2: This is an example of a lower envelope and an upper envelope of polygons that contains multiple types of polygons instead of just straight lines. The purple part means that this segment is both on the upper as the lower envelope. Note that we only can use normal lines in our data structure. This will be abstracted in Chapter 5.

4.1.2 Duality

A point $p \in P$ is on the convex hull if there is a line l through p such that all the other points in P are below l [7]. This translates to p^* being the lowest line of all the dual lines in P at some point. A point $p \in P$ in the interior has its dual line p^* always above some other line in the dual plane.

The lines on the envelope of the dual lines from right to left correspond to the points of the upper hull from left to right. So it is possible to compute the lower envelope of a set of lines L by taking the dual set L^* and compute the convex hull for L^*. To get the upper or lower envelope we just traverse the lower or upper envelope respectively. The concept of duality was still unknown at the time when Overmars en van Leeuwen designed their data structures and this the reason they did not use duality.

Because of the dual properties, the lower envelope corresponds to the upper hull and the upper envelope corresponds to the lower hull and the order of appearance of the points is the reverse of

the envelopes. Also, the left and right most points of the convex hull appear both on hulls which means that the corresponding lines contribute to both the lower and upper envelope. This is also visible in the envelope: the line with the fastest increasing slope and the line with the fastest decreasing slope both appear on the lower envelope and the upper envelope. When a line does not contribute to an envelope its dual point will lay in the interior of the convex hull.

4.2 Dynamically maintaining envelopes by Overmars and van Leeuwen

Overmars and van Leeuwen designed an algorithm using the same data structure to maintain convex hulls dynamically to solve a different problem than convex hulls: the maintenance of the intersection of *halfspaces* like in Figure 4.3. A halfspace divides the space into two parts and has a convex shape. This can be used for envelopes by simply ignoring the lower half of both spaces and this results as an envelope. The algorithm is designed to find intersection points p and q of two halfspaces. These points are found separately, so this algorithm can be adapted to find the intersection point of two envelopes. This is done be ignoring q so we only look for p and assume the halfspaces are envelopes. It is also possible to ignore the upper half of the halfspaces and find q instead. This means we are looking for the intersection point of two upper envelopes.

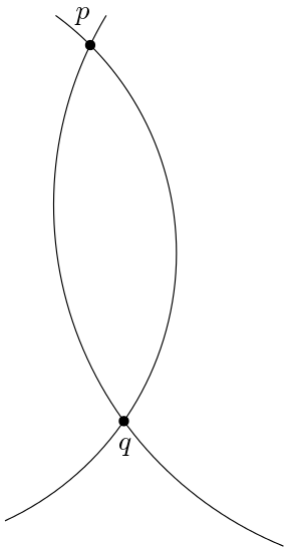


Figure 4.3: The original algorithm will find the intersections p and q between these halfspaces.

Dynamically maintaining the lower envelope uses the same data structures as used for the convex hull, but with lines instead of points. Finding a bridge between two hulls is now replaced by finding the intersection of two envelopes to merge them into a new envelope. More specifically: it finds the two intersecting lines of the envelope. The same data structures are used as for the dynamic maintaining of convex hulls (Chapter 3). The lines in the data structures are ordered in the same way as the lower envelope is ordered. We can see in Figure 4.4 how this works.

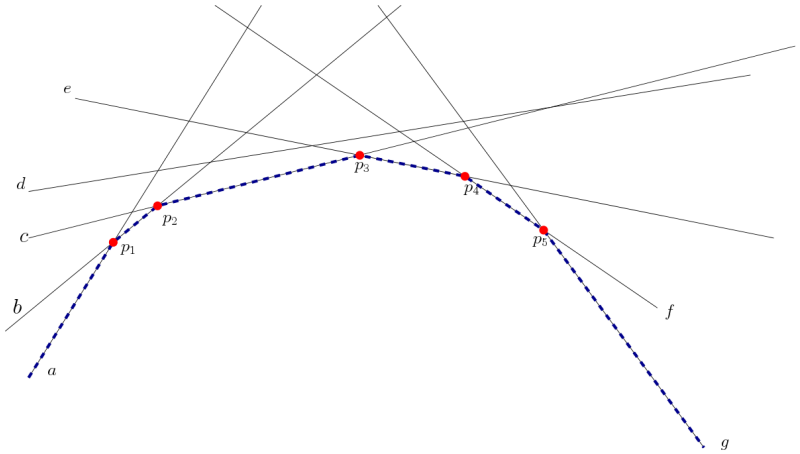


Figure 4.4: This is an example of an lower envelope, showed as a dashed blue line and it is made of line segments from the lines in the set. The order of the lines is g, f, e, d, c, b, a and d is the only line that does not contribute to the lower envelope. The envelope will be stored as the open line segment g ending in p_5, $[p_5, p_4], [p_4, p_3], [p_3, p_2], [p_2, p_1]$ and the open line segment a starting at p_1.

Some assumptions or restrictions are necessary in order to make the algorithm work and to make sure situations like Figure 4.5 do not occur.

1. An envelope covers the entire x-axis so there always exists an intersection between the two envelopes.
2. The slopes of the lines in the envelope of $n.left$ are strictly lower than the slopes of the lines in $n.right$ for a node n in the main tree.
3. The left and right envelope should be valid envelopes of the lines stored in the subtree of this node.
4. Each line in the set of lines has a distinct slope.
5. The lines in the set are sorted on their slopes from the fastest decreasing on the left to the fastest increasing on the right.

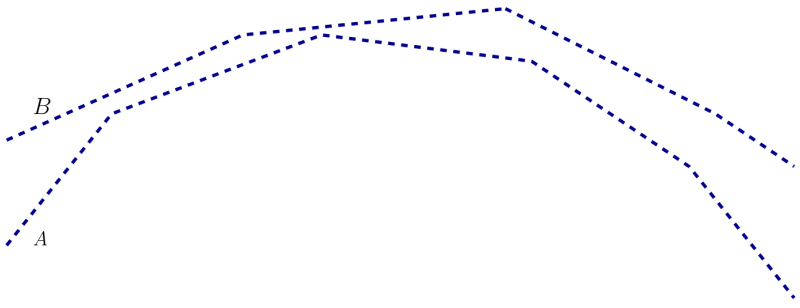


Figure 4.5: This situation cannot occur in practice.

4.2.1 Finding the intersection

There are four cases that can be encountered while finding the intersection and the algorithm is designed to handle accordingly. This is similar to the case distinction for convex hulls. Before we go into details, we need to introduce some variables. $S_l = [p', p]$ is the current segment in the left envelope and $S_r = [q_1, q_2]$ is the current segment of the right envelope. For each step points

$p \in S_l$ and $q \in S_r$ are sampled and used in the case distinction. See the implementation details (Section 4.3) for more details about sampling p and q. Let m be a horizontal line through p and l the line through S_l. The case distinction is based on the fact that l and m split the plane into four parts and the intersection point between the envelopes is somewhere in one of these parts, but we do not know where. The cases are based on the fact that q is also in one of these four parts hence there are four possible cases for the pointers to be updated. One of the cases is illustrated in Figure 4.6 but all the cases are also illustrated in the original paper [11]. The possible cases and actions we need to perform are:

1. *q is in region I.* means that the part below q can be ignored.
2. *q is in region II.* means that the part above p can be ignored.
3. *q is in region III.* means that the part above q can be ignored.
4. *q is in region IV.* means that the part below p can be ignored.

These cases replace the bridge algorithm for convex hulls and is highly similar, so this algorithm is also based on a binary search. The order of the splitting and merging at the end of finding the intersection is exactly the same and the running time remains $O(\log^2(n))$ time for insertions and deletions.

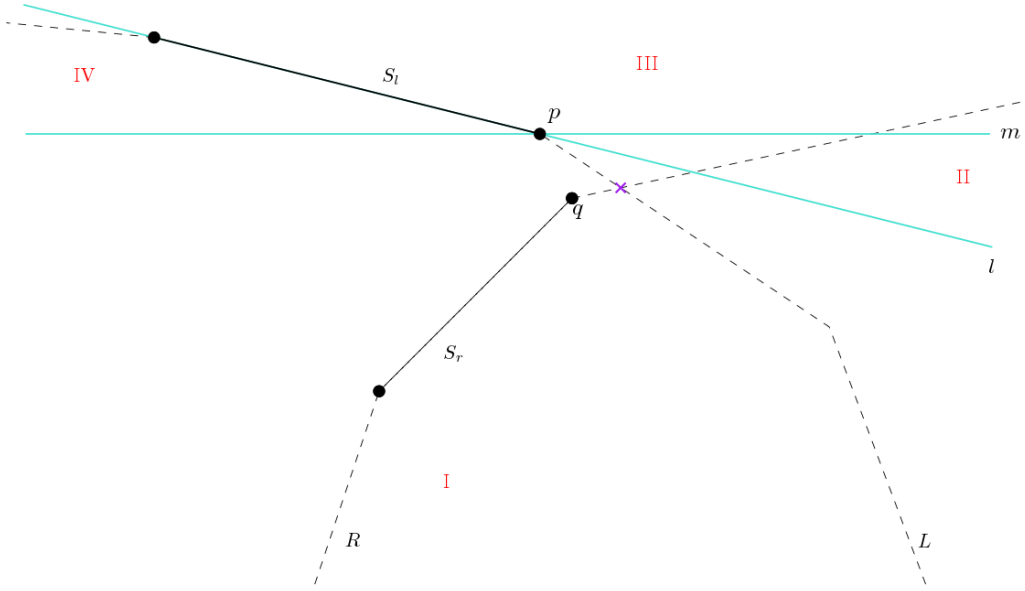


Figure 4.6: An illustration of the algorithm finding the segments that intersect. Lines m and l divide the plane into 4 parts and the intersection point must be in one of them, hence there are 4 possible cases.

This intersection algorithm is used in the place where the bridge algorithm is used for convex hulls. So the same procedures *UP* and *DOWN* are used to split the envelope at a node n and merge the split parts in $n.left$ and $n.right$ on the traverse down while inserting or removing a line. We traverse the tree back up all the way to the root and restore the envelope at every internal node encountered. This is again similar to the convex hulls and the only difference is this new algorithm to find the intersection. The complete envelope is stored in the root.

4.3 Implementation details

Maintaining dynamic envelopes without duality turned out to be a lot more complex to implement as expected. It was not finished because of this and because it is not really necessary to implement.

Maintaining envelopes dynamically can also be done using convex hulls and duality and the data structure for pseudo-lines is a bit different, so we actually do not really need it for that. We will see in Chapter 5 that lines can also be used as pseudo-lines.

Convex hulls are straightforward: a point is either on the convex hull or not, but lines are not that binary: a line can be partly on the envelope which means that the contributing part should be tracked somehow and this was not mentioned by Overmars and van Leeuwen. A possible solution is to augment lines with endpoints which is also done for pseudo-lines (Section 5.4) and that seemed to work reasonable, but there are more severe issues.

Overmars and van Leeuwen were a bit unclear about how the lines should be ordered. They only mentioned that the lines should be ordered on slope, but not in which direction: fastest increasing to fastest decreasing or the other way around. It took some time to figure out the correct way and this is important because the case distinctions do not work otherwise.

The biggest issue is the sampling of p and q. There are a lot of options and the most reasonable thing would be to pick one of the endpoints but the big question is: which one to pick? The big problem is that one part of a line segment lies in a different section then the other, so you would get different outcomes on how to traverse depending on how you sample. Because of this is it not really possible to just pick the middle point of the segment. An example of this is given in Figure 4.7. The cases are technically correct, the problem is how it translates into practice. The cases are correct about ignoring parts of the envelope but with the current way representing the envelopes it is possible to discard more than necessary which makes it possible to move away from the right segment and this is not trivial to solve. It could probably be solved by adding side cases but this problem can be solved much easier by just using duality for example.

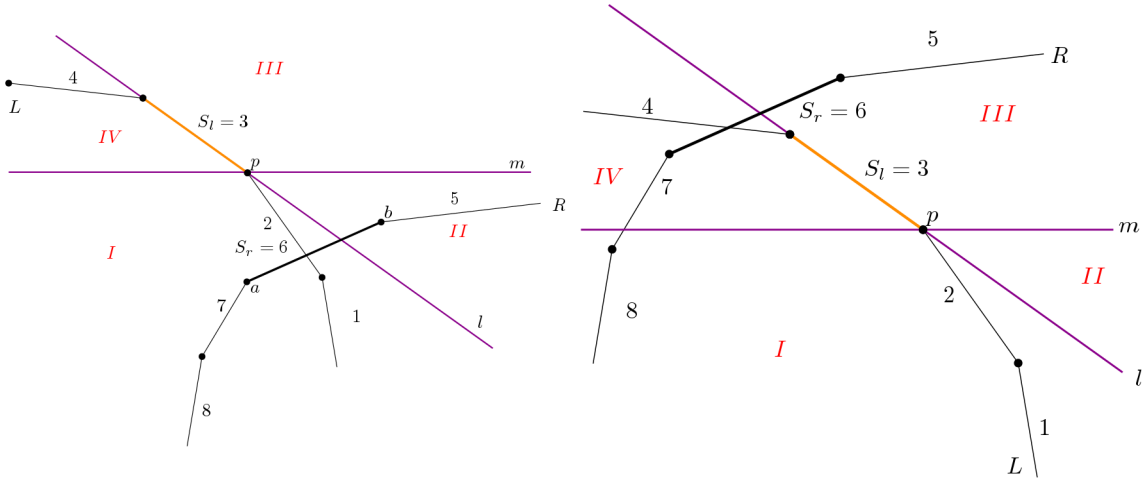


Figure 4.7: These examples show the same envelopes, but the right envelope is shifted a bit upwards in the right example. Segment 3 is in both case the current segment on the left envelope and the right endpoint is chosen as p. Segment 6 is the current segment on the right envelope. There are 2 options possible to choose q from: a or b. b works fine in the left panel. It gives us case 2: the part above p can be ignored so we move down to segment 2 or 1 which is great. In the right panel however we get the wrong case if we use $q = b$ (case 3 instead of 4). Technically it is correct because the intersection point lies on segment 6 and below b but in practice we traverse the tree down to segment 7 or 8 instead of moving from segment 3 to 4 and this is incorrect. So for the right panel we should have $q = a$. This causes problems in the left panel for almost the same reason: we have case 1 so everything below a can be ignored. In practice this means that we move to segment 5 instead of moving to segment 2 and move away from the correct segment.

So because of all these difficulties, time pressure and the fact that this implementation is not really necessary it was decided to let this rest and move on to the envelopes for pseudo-lines. The

main tree and the augmented tree used for the hulls were designed from the beginning in such a way that they could store both points and lines. This is done by making use of an abstract type T. This is still there although it is not really necessary anymore.

4.4 Related implementations

CGAL has implemented four algorithms to compute envelopes[1]: two to compute the lower or upper envelope of x-monotone lines or curves and two to compute the lower or upper envelope of input that does not have to be x-monotone. These are not dynamic algorithms and the documentation is unfortunately unclear about how the algorithms work in theory. The positive side is that it is possible to use a wide variation of input: lines, line segments, circles and geometrical shapes like circles among others[2]. There is a dynamic way to maintain the envelopes of a set of lines, but this uses duality so it is a different approach to tackle the same problem.

[1] Documentation of the functions.
[2] A full list is found here or here.

Chapter 5

A dynamic data structure for pseudo lines

Now we have seen how envelopes of straight lines can be maintained dynamically (Chapter 4). The final implementation step of this project is to abstract the case for straight lines to pseudo-lines, which is what we are going to do in this chapter. We start by defining what a pseudo-line is before we move on to the Agarwal [1] algorithms to maintain the envelope of pseudo-lines. It is based on the data structures designed by Overmars and van Leeuwen (Chapter 3) but the algorithm to merge envelopes is new. This algorithm is used in the place where *Bridge* is used for the dynamic convex hull.

Dynamically maintaining the envelope using duality is less complex as doing it without duality so it would make sense to apply duality here as well. But until now there is no known notion of duality for pseudo-lines which means the envelopes of pseudo-lines cannot be 'simplified' into an easier problem using duality.

5.1 What is a pseudo-line?

Pseudo-lines are an abstraction of straight lines. Agarwal et al. uses a lot of x-monotone finite sequences of line segments [1] in their examples but there are many possible options. The ones that are relevant in this thesis are straight lines, parabolas, and approximations of unit disks.

In order for the algorithm to work, there are some conditions that a pseudo-line must comply with in order to be used. Each type of line also has specific restrictions on its own.

1. A pseudo-line might follow a function f. This f must have an y-value for every x-value;
2. Every pair of pseudo-lines in a set has exactly one intersection point. The algorithm might behave unexpected otherwise. And it should be known how to compute this intersection;
3. A pseudo-line must be x-monotone so it follows that the function it follows also must be x-monotone in the case it follows a function.

The relevant types of pseudo-lines are listed below. Other possible pseudo-lines can be made by using functions like exponential functions, higher polynomial functions, sequences using this function or maybe even some goniometric functions as long as all the conditions for the algorithm are met.

1. *x-monotone curves.* This is an infinite sequence of line segments. Each pair of consecutive segments (a, b) is connected at point p: the right endpoint of a and the left endpoint of b, which is also their only intersection point. A curve is x-monotone which means that every point on a should be strictly left of every point in b, except for p. The first and last

segments of the curve are open segments: the first is unbounded on the left side and the last is unbounded on the right side. Curves with only one segment are a special case: this segment is unbounded on both sides since it is both the first and last segment and this means that it is equivalent to a line. The segments are stored in a list ordered on their appearance from left to right on the x-axis. A problem is that CGAL does not support open segments so it is a bit difficult to implement but this problem is not specific for these curves but a general issue. A more pressing issue is that there is no easy way to generate x-monotone curves randomly and assure the intersection restriction will always hold. This is the reason why x-monotone curves are not implemented, but they are mentioned because they appear frequently in examples from the paper.

2. Ordinary lines. A pseudo-line is an abstraction of lines so this means that ordinary lines are just an application of pseudo-lines or as even as an instance of x-monotone curves having only one segment. It is defined as $f : ax + by + c = 0$ (provided by CGAL), where b is constrained to a constant and a and c are generated randomly per line. A pair intersects once because this function is a polynomial of the first degree but only if at least a, c are unique for every line used. These lines can be easily generated by sampling points using CGAL and taking the duals of the points.

3. Approximations to unit disks. A set of unit disks are disks where all the radii are equal to a constant r. An unit disk is defined as $r^2 = (x-a)^2 + (y-b)^2$ centred around a point $p = (a, b)$ and has a problem: this is not a pseudo-line so this definition needs to be changed in order to use it as a pseudo-line. It is not x-monotone and not infinite and therefore we only consider the lower half of the disk and append open line segments l_1, l_2 respectively at the beginning and the end of the disk. It is not clear yet how these lines of the boundary should look like since we have to make sure that two unit disks always can intersect.

4. Polynomial curves in the form of $y = ax^2 + bx + c$ where $a > 0$ is constant and b and c are random. Consider 2 curves $f(x) = ax^2 + bx + c$ and $g(x) = ax^2 + dx + e$. The intersection $f(x) = g(x)$ evaluates like:

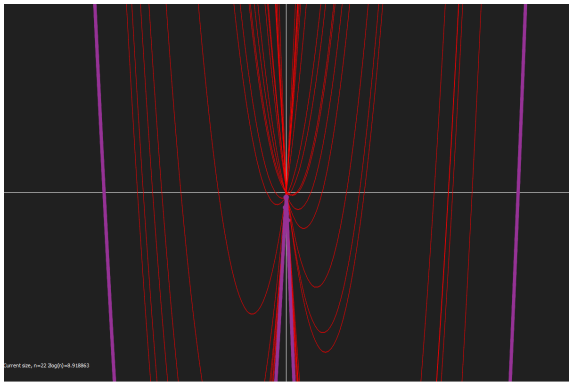
$$\begin{aligned} & f(x) = g(x) \\ & ax^2 + bx + c = ax^2 + dx + e \\ & bx + c = dx + e \\ & x = \frac{e-c}{b-d}, b - d \neq 0 \end{aligned} \tag{5.1}$$

   The problem this polynomial equation has is that every line l runs through $(0, l.c)$ because the terms ax^2 and bx are close to 0 when x approaches 0. So all the lines tend to go through 'hang' together around the origin because it is so difficult to spread them. This is visible in Figure 5.1 where the curves were generated using $a = 0.05, -100 < b, c < 100$ and yet they all tend to go through the same small area near the origin. This is a problem because now it is difficult to influence the size of the envelope if it is not possible to spread these lines, but they work.

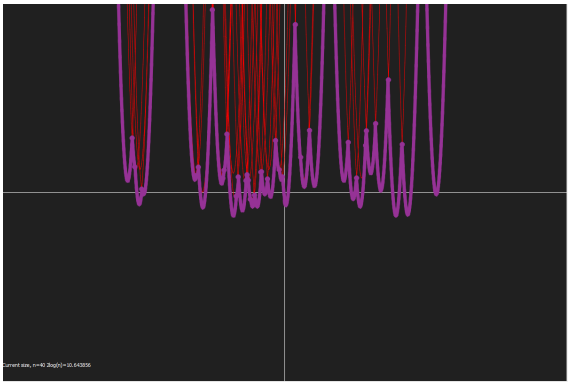
5. The possible forms of polynomials is endless and polynomials in the form of $y = (x-a)^2 + b$ work a lot better. This is similar to the one above but these are able to shift to the left and right along the x-axis using a which is picked randomly for every line. This is a huge advantage over the previous curves because now it is possible to increase the size of the envelope by spreading the curves more around the plane. The intersection is evaluated as

follows:

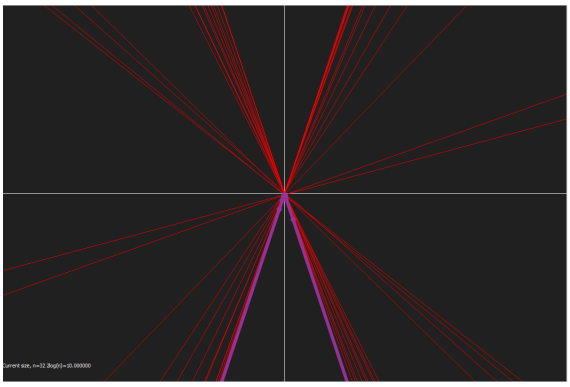
$$
\begin{aligned}
& f(x) = g(x) \\
& (x-a)^2 + b = (x-c)^2 + d \\
& x^2 - 2ax + a^2 + b = x^2 - 2cx + c^2 + d \\
& -2ax + a^2 + b = -2cx + c^2 + d \\
& -2ax + 2cx = -a^2 - b + c^2 + d \\
& 2ax - 2cx = a^2 + b - c^2 - d \\
& 2x(a-c) = a^2 + b - c^2 - d \\
& x = \frac{a^2 + b - c^2 - d}{2(a-c)}, a \neq c
\end{aligned}
\tag{5.2}
$$



(a) Curves with equation $y = ax^2 + bx + c$.



(b) Curves with equation $y = (x-a)^2 + b$.



(c) Lines.

Figure 5.1: Examples of different types of pseudo-lines. Examples of x-monotone sets of line segments can be found in the paper describing the algorithms [1].

5.2 Ordering pseudo-lines

Let K be a set of pseudo-lines of size n. Each line intersects every other line once, so this gives $n-1$ intersections per line which sums up to $n(n-1)$ intersection points for K. These intersection points are called the vertices or the arrangement of K and is denoted as $A(K)$.

In order to store the pseudo-lines we need to be able to sort the elements based on the $\leq$ relation between lines and this is done using a vertical line $l : x = x_0$. x_0 should be picked left of

the leftmost vertex of $A(K)$, so $-\infty$ or some approximation to $-\infty$ will work. For two pseudo-lines e_1 and e_2 the relation $e_1 \leq e_2$ is based on the intersection between l and e_1 and the intersection between l and e_2: $e_1 \leq e_2$ if and only if e_1 intersects l below e_2 intersects l.

Instead of ordering based on this definition, $e_1 \leq e_2$ is defined differently (which will be explained more detailed in Section 5.4). Recall that the Overmars and van Leeuwen algorithm for envelopes uses the slopes of the lines, from the fastest increasing on the left to the fastest decreasing on the right. This actually gives us the same result as comparing the lines with $-\infty$. Slopes of lines are the same as derivatives of lines and it turned out that ordering pseudo-lines based on the derivatives at their intersection point is a more practical solution because these numbers are not as extreme as σ. This is stated in Lemma 5.2.2.

Lemma 5.2.1. *Instead of comparing $f(\sigma)$ and $g(\sigma)$ for pseudo-lines f and g and $\sigma \approx -\infty$ when evaluating $f \leq g$ it is equivalent to compare f and g at $p.x - 1$ for intersection point p.*

Proof. Let f and g be pseudo-lines of the same type with intersection point p and assume $f < g$. Because of this we know that $f(\sigma) < g(\sigma)$ and since p is the only intersection point it follows that $\{\sigma \leq x < p.x : f(x) < g(x)\}$. This means that we can pick any $\sigma \leq x < p.x$ to evaluate $f < g$ so we can definitely pick $x = p.x - 1$. □

However, it turned out that there is an easier way of comparing, but this was found after the version using derivatives. $e_1 \leq e_2$ can also be evaluated by comparing the y-values of the points at e_1 and e_2 just left of the intersection point between e_1 and e_2. This is stated in Lemma 5.2.1.

Lemma 5.2.2. *Instead of comparing $f(\sigma)$ and $g(\sigma)$ for pseudo-lines f and g and $\sigma \approx -\infty$ when evaluating $f \leq g$ it is equivalent to evaluate $f'(p.x) > g'(p.x)$ at intersection point p.*

Proof. Let f and g be pseudo-lines of the same type with intersection point p and assume $f < g$ and that f and g have a derivative f' and g'. Like Lemma 5.2.1 we have $\{\sigma \leq x < p.x : f(x) < g(x)\}$, $f(p.x) = g(p.x)$ and $\{p.x < x < \infty : g(x) < f(x)\}$. In other words: f increases faster and faster until it catches up with g which means f' is higher than g' so $f < g \Leftrightarrow f'(p.x) > g'(p.x)$. □

In some cases it could be useful to use the derivatives instead if that is easier or more efficient. Lines for example can be compared using their slopes which are constant. This saves a bit of time.

5.3 Maintaining the envelope by Agarwal et al.

The proposed algorithm to dynamically maintain the lower envelope denoted as $L(K)$ of a set of pseudo-lines K is new, but it is based on the existing Overmars and van Leeuwen data structure. The main tree T is almost identical except for the fact that it uses pseudo-lines instead of points and the augmented structure for the envelope differs a bit more. It is described in detail what it should look like which is different compared to Overmars and van Leeuwen who were more abstract about it. Let us start with introducing variable names and names of operations. $L(K)$ is represented as a tree T and a node $n \in T$ has the following attributes:

- Pointers to its left and right child $n.left$ and $n.right$. And a pointer to the maximum pseudo-line in the left subtree $n.lmax = \max(n.left)$;
- The envelope of the pseudo-lines stored in the (sub)tree of n is $n.E$;
- If n is a leaf of T then it stores a pseudo-line in $n.l$ and $n.E = \{n.l\}$ after initialising n;

A node $\alpha \in n.E$ has the following attributes:

- α has pointers to its left and right child $\alpha.left$ and $\alpha.right$ and pointers to the rightmost leaf (containing the maximum pseudo-line) in the left subtree $\alpha.lmax = \max(\alpha.left)$ and the leftmost leaf (containing the minimum pseudo-line) in the right subtree $\alpha.rmin = \min(\alpha.right)$;

- Retrieving the envelope of α is also denoted as $L(E)$. This is an in order tree traversal that outputs the leafs;
- The lower envelope of $\alpha.lmax$ and $\alpha.rmin$ is stored in $\alpha.L$ if α is an internal node. Computing $\alpha.L$ takes constant time so it is also adequate to only store $\alpha.lmax$ and $\alpha.rmin$;
- α stores a pseudo-line in $n.l$ if n is a leaf;
- The intersection point between $\alpha.lmax$ and $\alpha.rmin$ is stored in $\alpha.p$. $\alpha.p$ on its turn has a x-coordinate and y-coordinate and are denoted as $\alpha.p.x$ and $\alpha.p.y$.

The envelope is maintained in exactly the same way the convex hull is maintained where intersections work the same as deletions: when inserting a new pseudo-line l into T, we split the envelopes of every internal node we encounter and merge the results back with the envelopes of its children, and we do this all the way down until the leaf to insert l is found. Then we traverse back up to the root and restore the lower envelope on every node using the new algorithm. The complete envelope is stored at the root. Even the procedures *DOWN* and *UP* (which split and merge envelopes the same as subhulls) are the same and used in the same places so the only difference besides the fact that points are replaced by pseudo-lines is this algorithm to find the intersection.

The splitting and merging of envelopes is a bit different compared to the convex hulls because the tree differs a bit, but it is roughly the same. A difference is that splitting the envelope removes an internal node and merging adds one. More details are found in the implementation details (Section 5.4).

This data structure can be used for two things:

- The data structure is able to answer the following query: finding the pseudo-line $e \in L(K)$ that intersects the vertical line at $x_0 \in \mathbb{R}$. This query is a binary search in the envelope which takes $O(\log n)$ time and works by comparing x_0 to the endpoints of internal nodes to find the right pseudo-line. This is a ray shooting query;
- Updating the envelope after an insertion or deletion of x which takes $O(\log^2 n)$ time. We traverse T to find the leaf v' corresponding to x and we split $v.E$ on the way down to v' for every internal node $v \in T$ starting at the root of T using *DOWN*. Then we traverse T back to the root starting at v' and update $v.E$ for every internal node v we encounter using *UP* and the new intersection algorithm. At node v we split the envelopes $v.left.E$ and $v.right.E$, merge and store the parts we need in $v.E$ while the unnecessary parts remain in $v.left.E$ and $v.right.E$. This is exactly the same as we already saw for the convex hulls. To update $v.E$ we need to find the intersection of $v.left.E$ and $v.right.E$.

5.3.1 Determining the intersection

The new algorithm to merge two envelopes of pseudo-lines is based on searching for the intersecting pair of pseudo-lines between the envelopes of $n.left$ and $n.righ$ for an internal node n in the main tree. The envelopes will be represented as $n.left.E = L_l$ and $n.right.E = L_r$ respectively assuming that these are computed correctly.

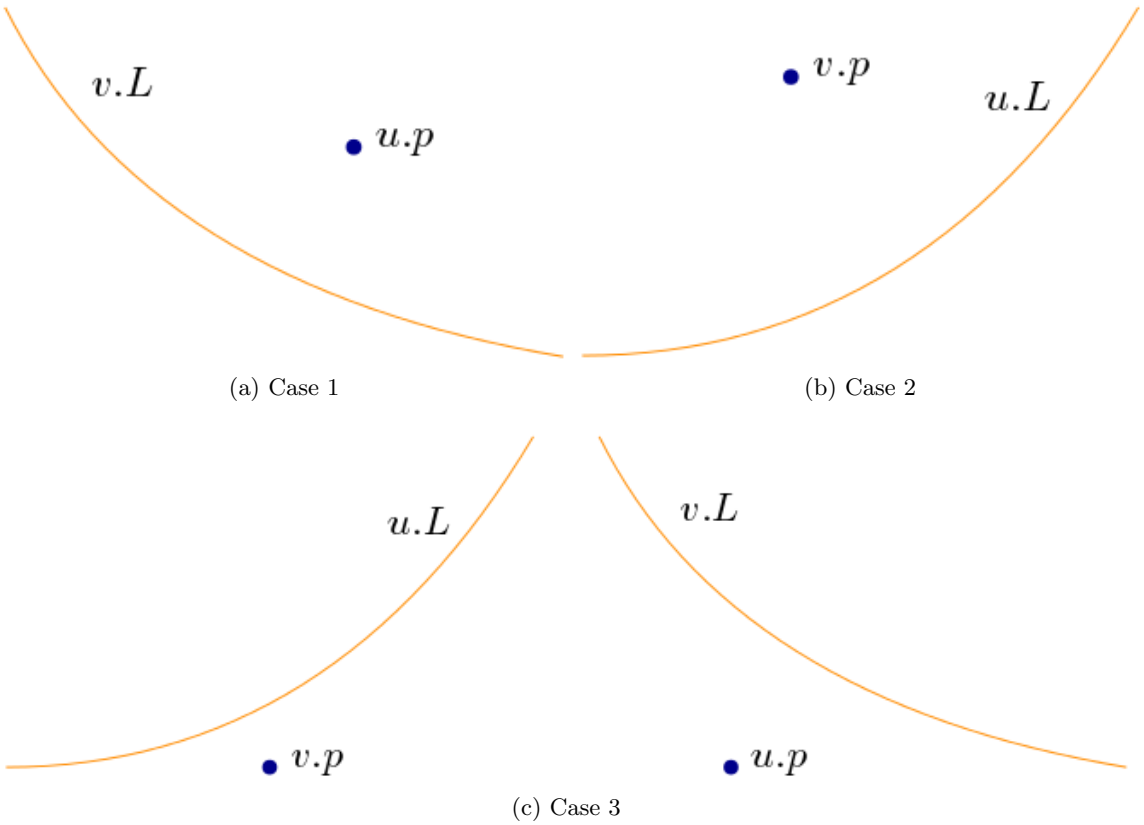
The envelopes intersect exactly once at point q and the goal is to find the intersecting lines $u' \in L_l$ and $v' \in L_r$ that intersect at this point. Let $u \in L_l$ and $v \in L_r$ be the current nodes and we start searching at the roots of L_l and L_r. We are performing a binary search using $u.p, u.L, v.p$ and $v.L$ to find u' and v'. $u.p$ is compared to $v.L$ and $v.p$ to $u.L$. This comparison gives a few cases and each hints us in which direction we need to traverse to find u' and v'. Eventually we stop when $u = u'$ and $v = v'$. Given the current nodes u and v, there are three possible cases with a side note for p: a leaf x does not have a $x.rmin$ or $x.lmax$ to compute a $x.p$, so for leafs we use one of its endpoints. The left endpoint of x is used if x is in the left envelope and the right endpoint is used otherwise. The cases are visualised in Figure 5.2.

Case 1: $u.p$ lies on or above $v.L$. This means that $q = u.p$ or that $q.x < u.p.x$ so u' is definitely not in $u.right$ but in $u.left$ or left of u if u is a leaf.

Case 2: $v.p$ lies on or above $u.L$. This means that $q = v.p$ or that $v.p.x < q.x$ hence v' is not in $v.left$ but in $v.right$ or right of v is v is a leaf.

Case 3: $u.p$ lies below $v.L$ and $v.p$ lies below $u.L$. This cases is more complex because now there are some more possibilities: $u.p$ is strictly left of q (so $u' \in u.right$ or right of u if u is a leaf), $v.p$ is strictly right of q (so $v' \in v.left$ or v' is left of v is v is a leaf) or both.

There are subcases so it is a bit more complex than this. These subcases exist because it is possible to take the wrong path while traversing down and are all covered in the pseudo-code. So the subcases make sure the right direction is always taken. We can change the direction by traversing one step back to the root and simply traverse in the other direction. This is done using two stacks that keep track of the traversed paths from the roots. Taking a step back simply means we take the first node of the stack. The left path is stored in the stack π_l and the right path inπ_r.



(a) Case 1

(b) Case 2

(c) Case 3

Figure 5.2: An overview of the three different main cases.

The intersection points $u.p$ and $v.p$ are compared against the envelope of two lines $v.L$ or $u.L$, but it is also possible and valid to compare them to the entire envelope $L(v)$ or $L(u)$ respectively. This would slow the algorithm down though because it requires a tree traversal. Both are evaluated in practice anyway and it turned out there is not a noticeable difference between the two (Chapter 6).

5.3.2 Running time

The running time should be the same as with maintaining the convex hull, namely $O(\log^2 n)$ time because it uses the same techniques. However, the running time increases in theory if $L(v)$ is used instead of $v.L$. This is because comparing points to $L(v)$ means there is an additional tree traversal while comparing to $v.L$ takes constant time. Finding the intersection using $L(v)$ takes $\sum_{i=0}^{\log n} \log n = O(\log^2 n)$ time in the worst case. Let k be the size of the envelope at step $0 \leq i \leq \log n$:

$$\begin{aligned} &\sum_{i=0}^{\log n} \log^2 k \leq \\ &\sum_{i=0}^{\log n} \log^2(2^i) = \\ &\sum_{i=0}^{\log n} (\log(2^i))^2 = \\ &\sum_{i=0}^{\log n} i^2 = O(\log^3 n) \end{aligned} \tag{5.3}$$

The intersection algorithm was implemented first using $L(v)$ instead of $v.L$. In the test results (Chapter 6) the algorithm using $L(v)$ will be referred to as the old way of finding the intersection (or it is not mentioned) and the intersection using $v.L$ is referred to as the new way.

Using sorted input

Let L be a set of ordered pseudo-lines. When we insert $l \in L$ one by one in order into the envelope T we will see that T will become extremely unbalanced: each l_{i+1} will be inserted at $\max(T)$ (or $\min(T)$ depending on the ordering) which means that T becomes a long trail with depth n. So in order to insert l_{i+1} we have to traverse $O(n)$ nodes and perform splitting and merging operations, and perform the intersection algorithm. The running time for sorted inputs evaluates to $\sum_{i=0}^{n} \log n = O(n \log n)$ time, ignoring the possible influence of the size of the envelope on the running time. Sorted inputs are used in experiments (Chapter 6) to illustrate the issue of unbalanced binary search trees.

5.4 Implementation details

5.4.1 Structure of classes

The class *PseudoEnvelopeWrapper* is responsible for delegating insertions and deletions into the main tree of *PseudoLNode*. It also has the implementations for *UP*, *DOWN*, retrieving the envelope, ray shooting queries and other similar functions can be put here. *PseudoLNode* is used as a node in a tree and can act both as a node and a leaf. it has a field to store a pseudo-line but this is only used if it is a leaf. It also stores the envelope of type *PseudoEnvelope* and has pointers to the left child, right child and maximum leaf in the left subtree.

The implementation of the intersection algorithm can be found in *PseudoEnvelope* along with a binary search tree of type *ElementsInLeafs* which stores objects of type *PseudoLine* and uses some basic tree functions of *AVLTree* which is the same tree used for convex hulls (and still not height balanced). *PseudoEnvelopeWrapper* uses *PseudoEnvelope* from the root of the main tree (has type *PseudoLNode*) to answer ray shooting queries and the retrieval of the envelope and also contains a binary search function that uses the intersection points.

Pseudo-lines are represented in a class called *PseudoLine*. It has multiple constructors and each intialises a different type of pseudo-line. The specific type of the pseudo-line is stored in

a field of type *BaseLine* (which corresponds to $n.l$ for a node n) and every type of pseudo-line has a class derived from this. In this way it is possible to put the responsibilities for things like finding the intersection of lines, retrieving y-coordinates at a certain x and derivatives of the line for each type of pseudo-line. The intersection algorithm does not know what type of pseudo-lines it is working with and fails if two pseudo-lines are used with a special type to represent nodes. In fact only the end user and *PseudoLine* know what type of pseudo-lines are used. The endpoints are stored in *PseudoLine* as attributes and $n.p$ also has a field. Each node in *ElementsInLeafs* representing the envelope stores an object of type *PseudoLine* but only the leafs store actual pseudo-lines. Internal nodes have a special type of pseudo-line to denote that is empty and $n.p$ is only set when the pseudo-line has this empty type and $n.p$ is never set if the pseudo-line contains and actual pseudo-line conversely.

The tree *ElementsInLeafs* storing the envelope is different from the tree storing hulls, so merging and splitting is implemented twice unfortunately. Merging is implemented in the most easy way possible (Section 2.1.5), so this may result in highly unbalanced trees and thus inefficient running times and this still needs to be improved.

5.4.2 Other practical details

Merging two envelopes is implemented differently compared to merging convex hulls. The easiest way of merging is implemented (Section 2.1.5) first, but this never changed. A quick reminder about this method: it simply creates a new root node α, stores the left tree in $\alpha.left$ and the right tree in $\alpha.right$ and updates $\alpha.p$. The first part takes constant time but updating $\alpha.p$ takes two tree traversals for $\alpha.lmax$ and $\alpha.rmin$ which take $O(\log n)$ or even $O(n)$ time each depending on the tree. So this merging algorithm has room for optimisation.

Although unit disks are not implemented to use by the algorithm, some work has been done: the intersection can be found using CGAL and the other functions are implemented as a first version. CGAL's algorithm to compute envelopes (Section 4.4) can be used to validate the correctness.

The original sorting is based on working with $x = -\infty$ and this was initially implemented in the code. $-\infty$ was approximated as $\sigma = -10000$ because it is not really possible to put ∞ into an equation but using an approximation should work. It turned out after a while and a lot of bugs that -10000 was not enough at all. A new problem arose because sorting should use an x-value left of all vertices of $A(K)$, but this approximation was just not enough which failed the sorting often. Decreasing this approximation to $\sigma = -4999000$ helped a lot but essentially pushed the problem away instead of solving it. That is why sorting was approached differently. First using derivatives (Lemma 5.2.2) and later using an x-value just left of the intersection point of the two lines that are compared, which is actually easier than derivatives (Lemma 5.2.1).

The very same σ is still used however to cap open endpoints. Open segments are not supported by CGAL, so they are capped at σ if the open end is on the left and capped at $-\sigma$ if the open end is on the right. It is also possible that a segment is capped at both ends. Capping segments occurs after splitting an envelope: for the new envelopes E_1 and E_2 we need to reset the right endpoint of $\max(E_1)$ and the left endpoint of $\min(E_2)$ because these just became open segments. And equivalent for merging: after the merging of two envelopes we need to update the endpoints to $\alpha.p$ for segments $\alpha.lmax$ and $\alpha.rmin$ where n is the new root and where $\alpha.p$ is the intersection point between $\alpha.lmax$ and $\alpha.rmin$. We also reset both endpoints in the intersection algorithm when we reached a leaf.

Evaluating the correctness for the envelope using lines was not too difficult. CGAL has an algorithm to compute envelopes so this could be used but it is faster to maintain the upper envelope of the duals instead. For parabolas it was initially not clear how this algorithm could be used but it should be possible so this would be something to look into in the future. This algorithm might be useful for unit disks later on but for now this means that correctness of curves is based on visually inspecting the result. More details about correctness can be found in the results (Chapter 6).

Chapter 6

Experiments

The results of the test are represented in two different ways. There are three types of convex hulls to test and three different inputs for each type which means there is a lot of different data. The results are visualised in plots along with the approximated asymptotes. This is a good way to see the details (e.g. how the times vary at a certain n and how well the asymptote matches the data) but it is not the best way to compare different inputs or types of computing convex hulls. Some plots are found in this chapter but more detailed plots are contained in Appendix A.

6.1 Measuring running times

Every algorithm discussed has a running time which is proven in theory, but is not yet evaluated in practice. So one of the goals is to find out if the theoretical bounds are correct and if the running time is influenced by different inputs. The running times of insertions and deletions will be evaluated with respect to different types of input that influence the size of the output (convex hull or envelope). Each operation is timed from the moment the insertion or deletion starts until the moment that the augmented data structure (convex hull or envelope) at the root is updated.

This process of measuring time is repeated k times at different input sizes $n' \in \{\delta, 2\delta, 3\delta, \ldots, N - 2\delta, N - \delta\}$ where $\mod(N, \delta) = 0$ for the final size N and a constant δ. We need to do two things when testing k operations for a certain n': insert some random points until the set of the set of elements (points or pseudo-lines) in the data structure contains n' elements. Then we create a random set K of size k and for each $k' \in K$ we insert k' to measure the insertion time and immediately delete it to measure the deletion time. The insertion and deletion times are stored with the corresponding n'.

Eventually there are k running time measurements for each $n' \in \{\delta, 2\delta, 3\delta, \ldots, N - 2\delta, N - delta\}$. The running time measurements correspond to the y-axis of the plots and the x-axis corresponds to the size of the set contained in the data structure. It is important to remember that n' is not the same as the size of the convex hull or envelope but the amount of elements stored in the tree. This data is saved in JSON format and processed using Python to make plots and approximate asymptotes.

The constants a, b in an asymptote like $O(a * n \log(n) + b)$ are hidden by the definition of big O notation but that does not mean these constants do not impact the running time. These constants matter a lot for visualisations and it is our task to approximate them as good as possible. There exists a Python package *scipy* [1] that contains a function *curve_fit* that takes input data, a function f and parameters for f to fit according to the data. This is exactly what we need to see for asymptotes as $\{a + b \log(n) = O(\log n), a + bn \log(n) = O(n \log n), a + b \log^2 n = O(\log^2 n), a + b \log^3 n = O(\log^3 n), a + bn = O(n)\}$ for parameters a and b are suitable for a certain algorithm.

[1] Documentation is found here.

The parameters are constrained to $0 < a, b < \infty$ but stay close to 0 in practice. Every call of $\log x$ is replaced with $\log(x + \epsilon)$ with ϵ close to 0 to prevent calculating $\log 0$. This method works so well because it will never be possible to fit the wrong asymptote perfectly since these functions differ a lot in shape (see Figure 6.1). The asymptotes are visually inspected to conclude if they are reasonable. *scipy.optimize.curve_fit* can also be used to compute the residuals between the data and the asymptote. The best approximation should have its residuals close to 0 so this is a secondary way of verifying an asymptote for cases like $O(\log^2 n)$ and $O(\log^3 n)$ where the differences are small in practice. These differences are small although $O(\log^3 n)$ is a lot worse than $O(\log^2 n)$ but they might become similar after fitting them to the data.

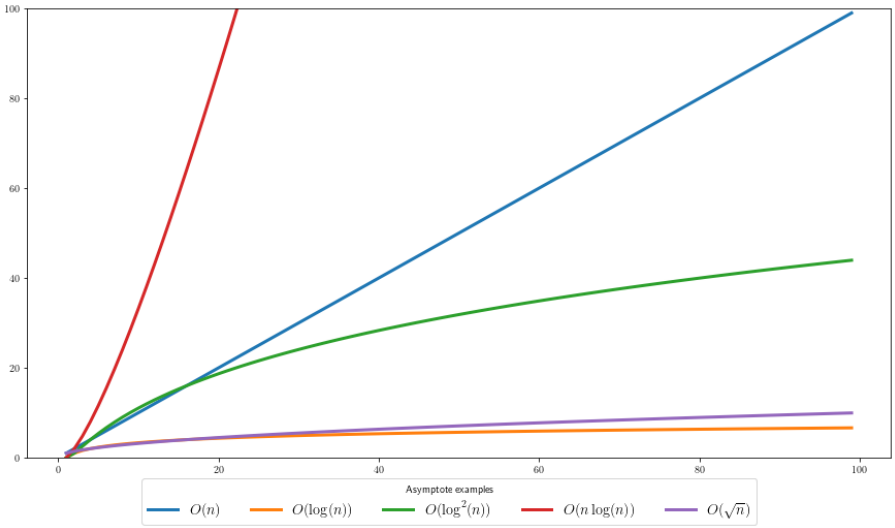


Figure 6.1: Examples of different asymptotes.

6.1.1 Visualising running times

Each figure in Appendix A shows a test with a certain input. Each test contains six subplots divided in two columns and three rows. The first column displays insertions and the second deletions. The first and second row display the running time data but the second one is cropped around the mean while the first one shows all the data points. Both rows contain asymptotes which are fitted using *curve_fit* along with the means of the data to compare the asymptote with the means.

An important note to make is that the representation of the convex hull is different for every variation of maintaining the convex hull. For the Overmars and van Leeuwen versions it is stored in two separate trees or two separate lists and for the Delaunay triangulation we need to traverse the outer face of the triangulation to retrieve the convex hull. In Figure 6.2 we see how traversing the outer face after an insertion or deletion impacts the running time. This difference is small but it would be bigger if the size of the convex hull increases. Each test is run without outputting the convex hull as a list because this gives unnecessary overhead and makes comparing each variation unfair.

6.1.2 Dealing with outliers

The data may contain outliers which may influence the averages (means) negatively. They also have a negative effect on the estimations for asymptotes, so it would be better if those are removed. The challenge is now to find out what an outlier is for each set of data points for every size n.

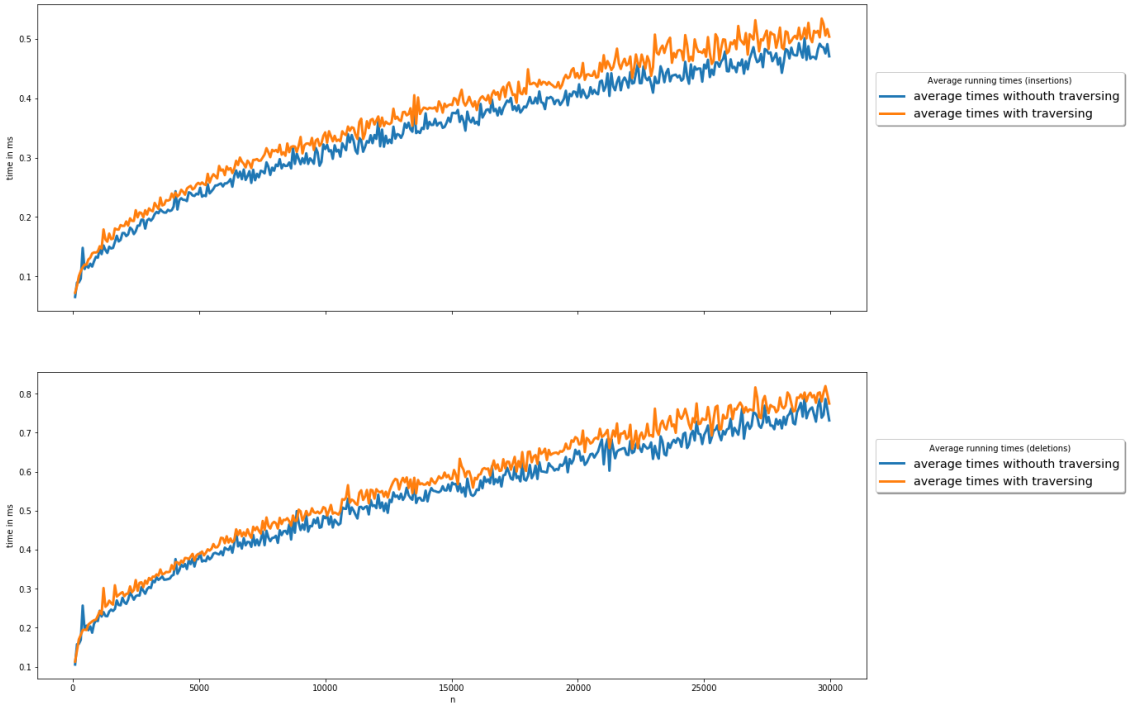


Figure 6.2: Averages of two kinds of tests: one traverses the outer face after an insertion or deletion and the other one does not. Each variation is run five times.

Most outlier detections are based on one or more assumptions and outlier detections are usually based on the assumption that the data is normal distributed. The running time data should be normal distributed according to the central limit theorem [9] which in general can be applied for sample sizes of $n > 30$ but in practice it can be applied sometimes for even smaller sample sizes. However, it seems that a lot of data points were not random distributed as they failed the Anderson-Darling test or the D'Agostini test for normality. A way to find outliers is Tukey's method which relies on boxplots and is also based on the assumption of normality. Regardless whether the data is normal distributed or not, applying it does not improve the asymptote estimations significantly hence this method is not used in general. In most cases it only shifts the averages a bit up or down and it is not useful to make conclusions. There is one case though where it actually is really useful: Figure 6.3 shows an example of running times for maintaining the convex hull of points sampled on a disk using *std::vector*. We clearly see that the running times are really low or really high and there is not much in between. In this case it makes sense to remove the lowest points to fit the upper parts of the data better. The asymptotes on the adjusted data go nicely through the upper part of the data points. This is the only case where it is really clear that outliers should be removed. The other data sets are more spread over the plane or divided into multiple denser regions but not as extreme as here.

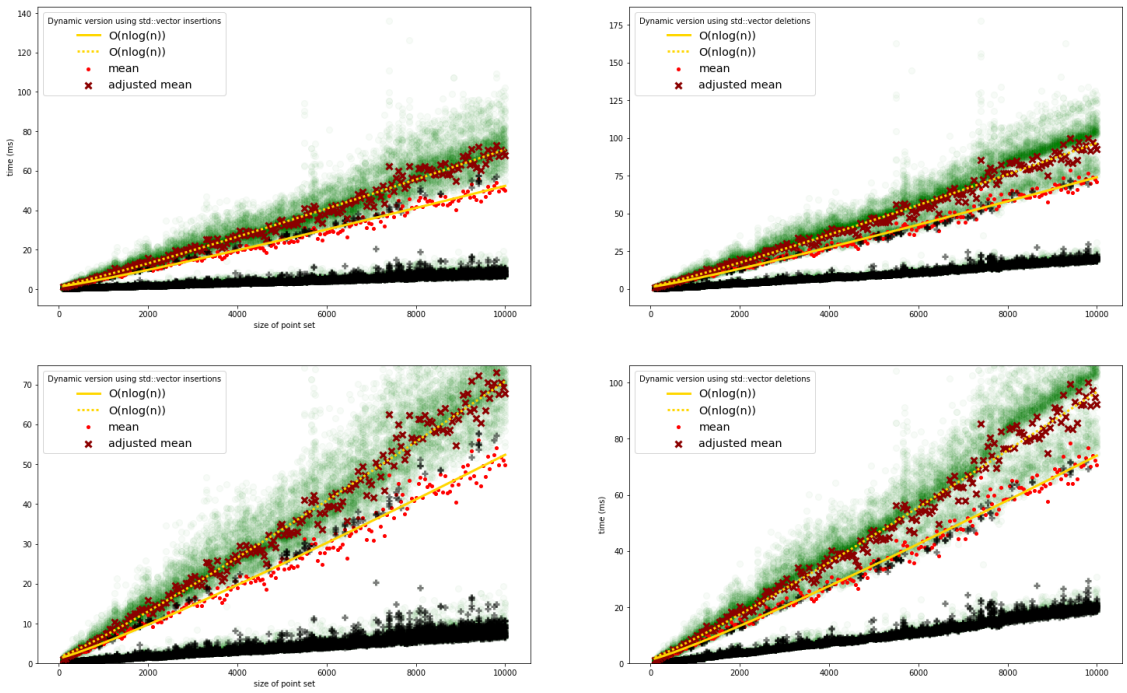


Figure 6.3: The only case where removing outliers made a lot of sense. The convex hull is dynamically maintained using *std::vector* and points are sampled on a disk. The outliers are displayed in black.

6.2 Verifying correctness

Besides the running time it is also important to verify if the computed result is correct. One of the ways to verify if the computed convex hull is actually valid is using a visual debugger 6.4. This method was extremely useful while developing the algorithms because it makes debugging a lot easier. However, this visual method it is not really a good way to test intensively and on large data sets because it takes too much time and will be prone to human errors. A more robust way of verifying is similar to testing running times by comparing the convex hull from the dynamic data structure to the convex hull computed by CGAL, which is always correct, so when the results do not match then the dynamic structure must be wrong.

Convex hulls can be easily compared but validating envelopes is more complex. Envelopes of lines can be validated by taking the upper hull of a Delaunay triangulation and duality (Section 2.3). Parabolas are more difficult to verify using the CGAL algorithm for envelopes although should work on parabolas but it is unclear yet how, so only the visual debugger can be used in this case.

Convex hulls work correct. The dynamic structure contains the minimum and maximum point twice and it is different ordered than the Delaunay triangulation but that is a small thing to keep in mind. The envelopes on the other hand are sometimes incorrect unfortunately and parabolas are more prone to errors as lines. Some examples of these errors are shown in Figure 6.5 where we see that the envelope sometimes has small incorrectnesses. The envelope itself is continuous over the x-axis and does not contain gaps. It is unclear why this happens but it is considered a bug in the code, not an error in the algorithm.

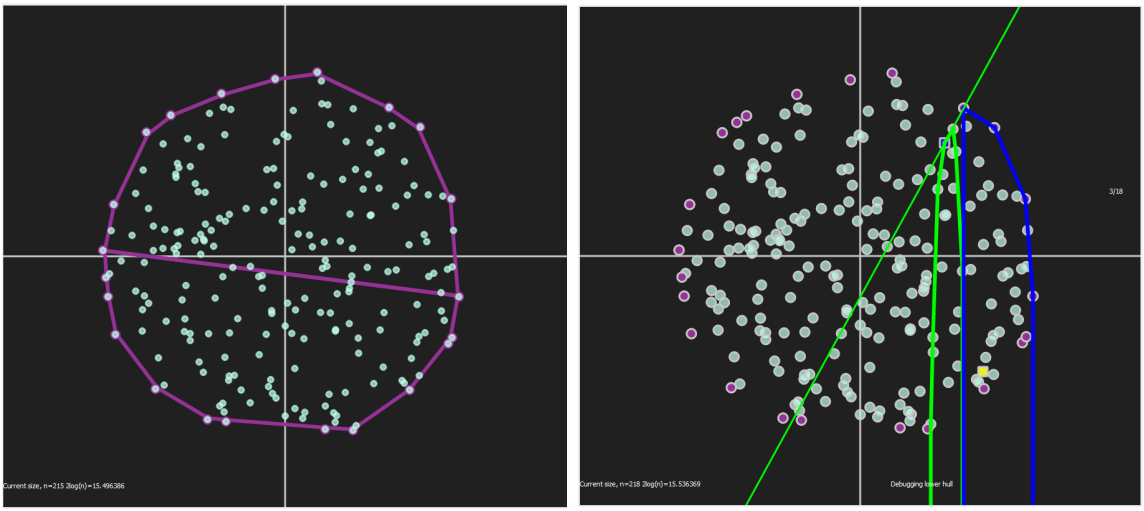


Figure 6.4: An example of how the debugger works. On the left we see a complete convex hull and on the right we see one of the steps of updating the upper hull after an insertion. The lower hull can be debugged in the same way. The steps to find the bridge are also able to be inspected. It displays some information about the current case, like what the average size of the number of steps should be: $2 \log n$ for the current n. A similar debugger was made for envelopes.

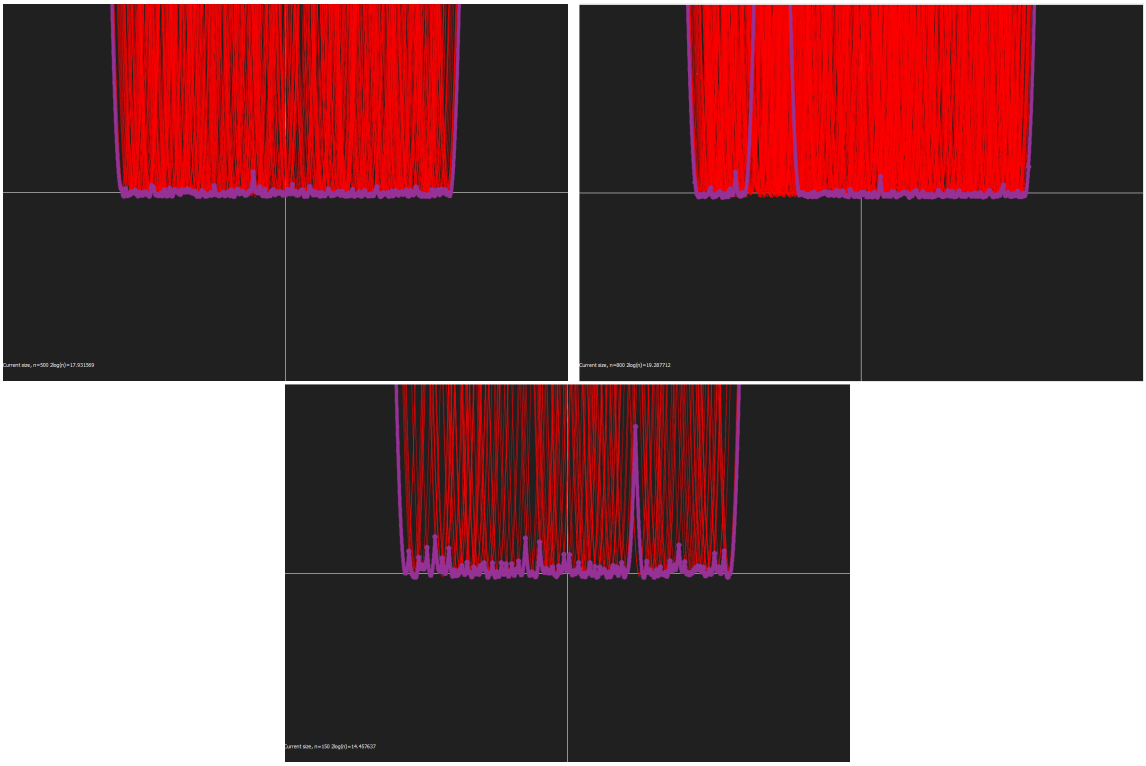


Figure 6.5: Examples of envelopes of a set of parabolas containing small errors in the envelopes.

6.3 Test cases

Testing running times will be done using all combinations of different inputs and different ways of computations. The CGAL algorithms for convex hulls and algorithms are not tested because they are too slow for dynamic applications (except the Delaunay triangulation).

6.3.1 Convex hulls

CGAL provides random generators to sample points in different ways [2]: on a disk, inside a disk, inside a square or on a square. This last one cannot be used because it will generate multiple points with the same x-value which we cannot use so there are three ways of generating data sets. These are combined with the multiple ways of maintaining a convex hull: using *vector*, using trees, CGAL's Delaunay triangulation.

Combining these should tell us how the different data structures are affected by different data because each way of sampling has its own limits on the size of the convex hull. If all points are on a disk, then the size of the convex hull will be the entire set of points. Sampling inside a square will give the lowest size of the convex hull. Sampling in a disk is somewhere in between. So the combinations of different algorithms and inputs can tell us a lot about when each algorithm should be preferred. There are a lot of possible ways to sample points, but only these are chosen because of the focus of this thesis project is about envelopes.

6.3.2 Envelopes of pseudo-lines

There is no built in function in CGAL to generate pseudo-lines, so this requires a bit of work. While generating it is important to make sure the requirements from in Chapter 5 are met, but this still leaves us different possibilities for different inputs. There are two types of pseudo-lines tested: lines in the form of $ax + by + c = 0$ and polynomials like $(x - a)^2 + b$. Polynomials in the form of $ax^2 + bx + c$ where a is fixed and b, c are random are skipped because it is too difficult to spread the curves over the x-axis and this makes it not possible to influence the number of lines on the envelope.

Curves are generated by picking random numbers for each variable. It is easy to generate correct sets in this way and it is possible to influence the size of the envelope by taking a larger sample range for a and the range of c is fixed. Lines are generated by generating sets of points and taking the duals. Inputs that give large envelopes are generated by sampling points on a disk and the size of the envelope will be about $\frac{n}{2}$ since we only consider the lower envelope. Inputs that give a small envelope are generated by sampling points in a disk or in a square.

For envelopes we also test the effect of sorted inputs to see if height balancing trees would be a good feature to add. We insert the lines from a set L if we have either $\{l \in L : l < \min(T)\}$ or $\{l \in L : l > \max(T)\}$. In this way we make sure the tree is highly unbalanced to either the left side or right side.

In Chapter 5 we discussed that an internal node n in the envelope should be augmented with the envelope of $\min(n.right)$ and $\max(n.left)$ as $L(n)$, but we can also traverse the subtree rooted at n instead. Both are evaluated and the latter was implemented first so if there is this is referred to as the 'old way' of traversing n and the first way is referred to as the 'new way' of representing $L(n)$. If there is no reference to the method used then it is the old way.

[2] https://doc.cgal.org/latest/Generator/classCGAL_1_1Random__points__in__disc__2.html

6.4 Expectations and hypotheses

6.4.1 Height balancing of trees

The trees used in the data structures are not height balanced and this probably the cause of high variations of the running times. This also might make it more difficult to find the real asymptote in some cases because the running times are more scattered. Height balancing was not implemented mainly due to time limits. This means that some inputs would perform terribly bad as a consequence, especially for sorted inputs.

6.4.2 Running times for convex hulls

The dynamic convex hull has an $O(\log^2 n)$ update time per insertion and deletion using trees. So it is expected that this will become visible in the plots. The asymptote should be worse for dynamic convex hulls using a vector, since it theoretically takes $O(n)$ time per update, so these should also be slower in practice. We will see if this is also the case in practice. The Delaunay triangulation should take $O(\sqrt{n})$ time or $O(n)$ time in the worst case for insertions and $O(\sqrt{n})$ time for deletions as discussed in Section 2.4.2. It is probably not possible to outperform the Delaunay triangulation, but this is not the goal. The goal is to make a correctly working implementation that can be extended to envelopes.

6.4.3 Running times of envelopes for pseudo lines

We want to know if and how the theoretical running time translates into practice. There are two cases: unsorted inputs and sorted inputs and both are expected to behave different from each other. As discussed in Chapter 5 the running time should be $O(n \log n)$ time for sorted inputs and $O(\log^2 n)$ or $O(\log^3 n)$ time for unsorted inputs.

The variation in input size will likely have an effect on the running time, just like the type of pseudo-line might have. It is also likely that we find a difference in the running times between the old and new way of representing $L(n)$. The new way should be faster, but again the trees are not optimally implemented so the differences might be low.

6.5 Results for convex hulls

In Figure 6.6 we see every combination between inputs and ways of computing convex hulls in a single overview. Each example is displayed more detailed in Appendix A more or less like Figure 6.3 is.

It did not make sense to test inputs for high convex hull sizes for the Delaunay triangulation and the dynamic version using *std::vector* because it is clear how slow it is with the current inputs. There is also a difference between the sizes of the sets in other runs ($n = 80000$ and $n = 160000$) mainly because the time it took to wait for a test, memory usage and its effect: a bigger test does not always make the comparison better.

We see that the Delaunay triangulation is the fastest in general, followed by the dynamic version using *std::vector* and the dynamic version using trees as last. However, the dynamic version using trees is the only that can handle extremely large convex hulls. It is a surprise that the dynamic version using *std::vector* is a bit faster than the dynamic version using trees because the version using *std::vector* has to copy lists a lot which is not efficient. But surprisingly it turned out to be faster than the tree version. It could be due to the fact that the operations on trees are not optimized yet but the Delaunay triangulation implementation and the operations on *std::vector* are.

The Delaunay triangulation has a running time of $O(\sqrt{n})$ for insertions and deletions when the size of the convex hull is low but this increase to $O(n \log n)$ when the size of the convex hull

is extreme. Its asymptotes go from $O(\log n)$ for low sizes of convex hulls to $O(n)$ or $O(n \log n)$ for high sizes of convex hulls. It is a bit difficult how the see the difference between $O(n)$ or $O(n \log n)$.

It is interesting to see how the running time is not really affected by the size of the convex hull although the running time increases a bit when the convex hull is lower. The asymptote of $O(log^2(n))$ time is nicely visible in the plots. But it is the slowest when the size of the convex hull is low.

The Delaunay triangulation works really fast when the convex hull has a reasonable or small size, like when the points are sampled in a disc or in a square. The running time increases rapidly when the size of the convex hull is really large, so in the case of points on a disc. This is probably due to the high number of adjacent vertices that need to be updated after an insertion or deletion. Deletions are even slower. The results in Figures A.1 A.2 A.3 strongly suggest that insertions and deletions run in $O(\sqrt{n})$ time. Deleting points when the size of the convex hull is really large tends to take $O(n^2)$ time. This is the reason why testing with a large convex hull for Delaunay triangulations was done with a much smaller input compared to the other versions.

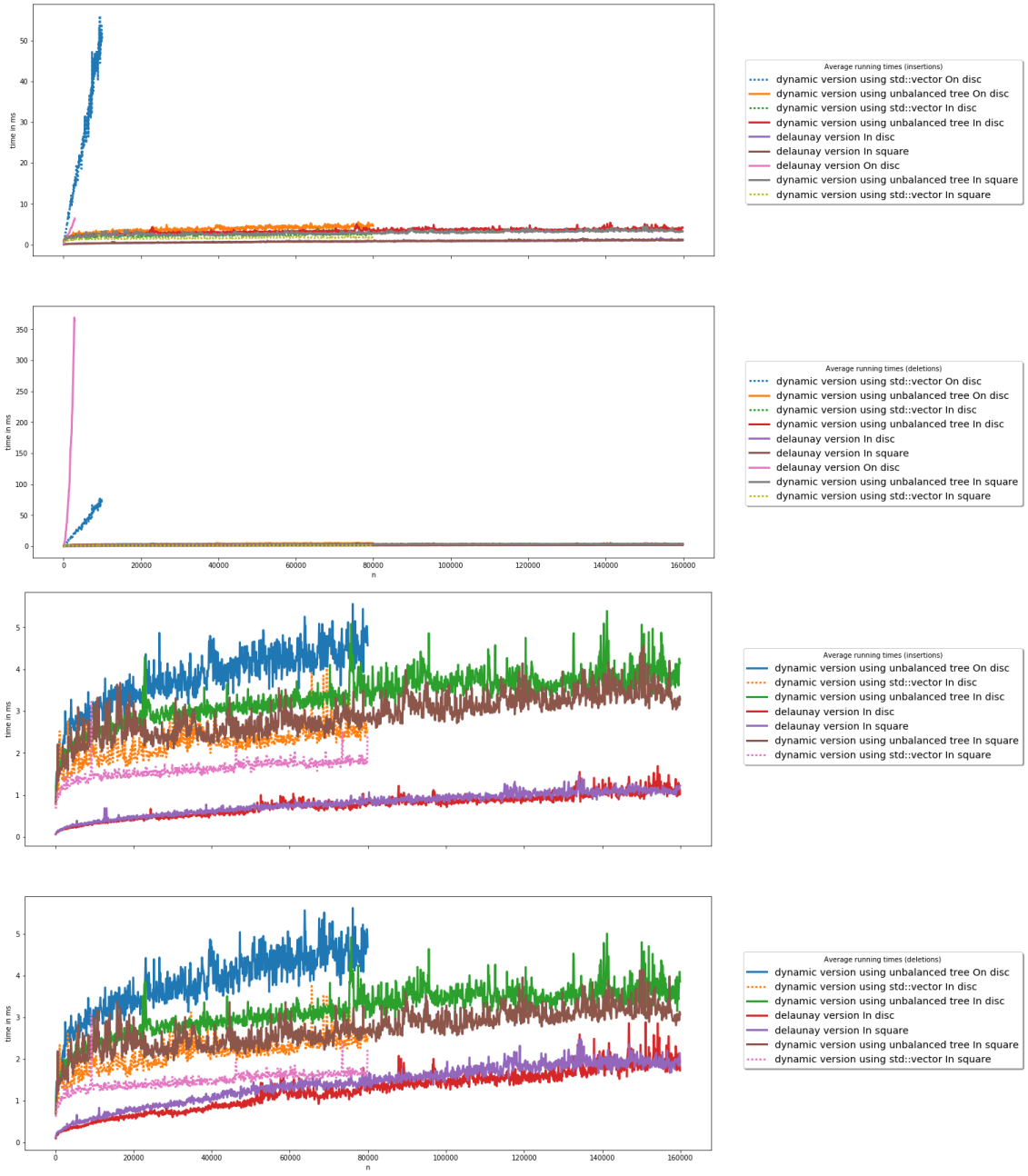


Figure 6.6: Overview of insertions and deletions of the different combinations for conex hulls. The top plot contains all combinations and the bottom lacks Delaunay triangulation and the dynamic version using *std::vector* with points sampled on a disk.

6.6 Results for envelopes

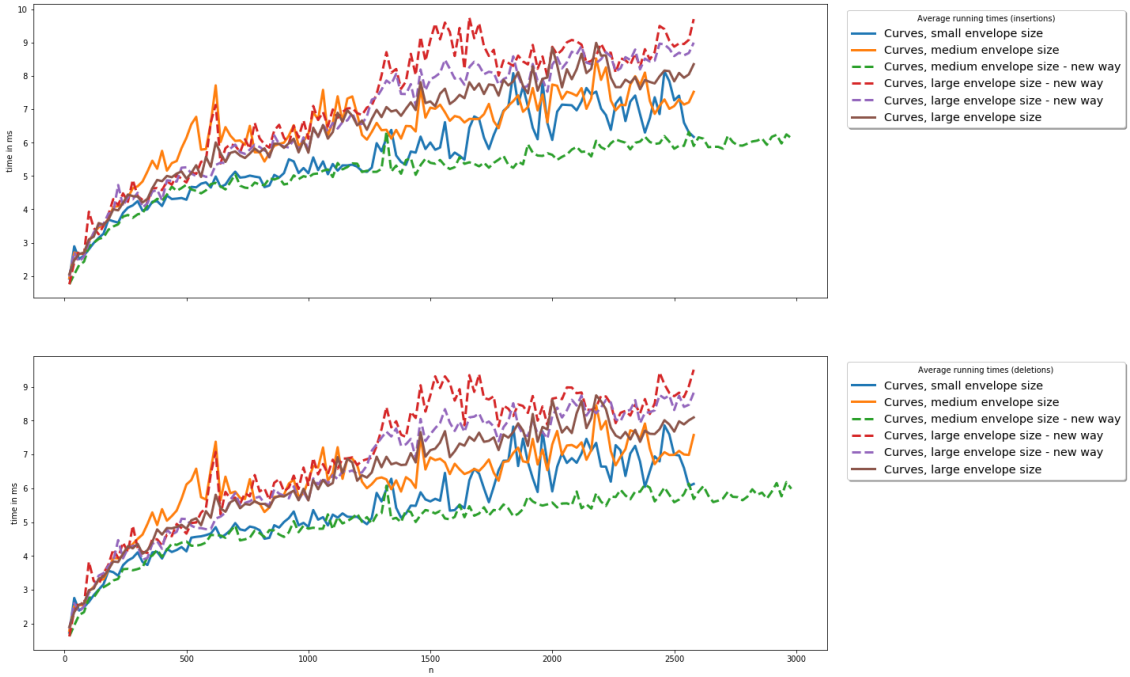
Figure A.16 shows test results of all the combinations between inputs (lines or curves) and the size of the envelope. All of them are also tested with theoretically the inefficient way (using $E(v)$ in the intersection algorithm) and the theoretical efficient way (using $v.L$: the envelope of $\max(v.left)$ and $\min(v.right)$). The first is referred to as 'old way' or is not explicitly named at all while the latter is always referred to as 'new way'. Appendix A contains plots of every test in more detail.

It turns out it is near impossible to see a difference in the asymptotes $O(\log^2 n)$ and $O(\log^3 n)$ because these are really close to each other after fitting them. There is a small difference visible in the beginning of the plot where the input size is still low and removing outliers with extremely high running times does not really help either. The only thing we can do is look at the plots in Figure A.16 where we can compare the differences. It turns out that there is no real difference in running times except that the efficient way (storing only a part of the envelope) has a more stable mean.

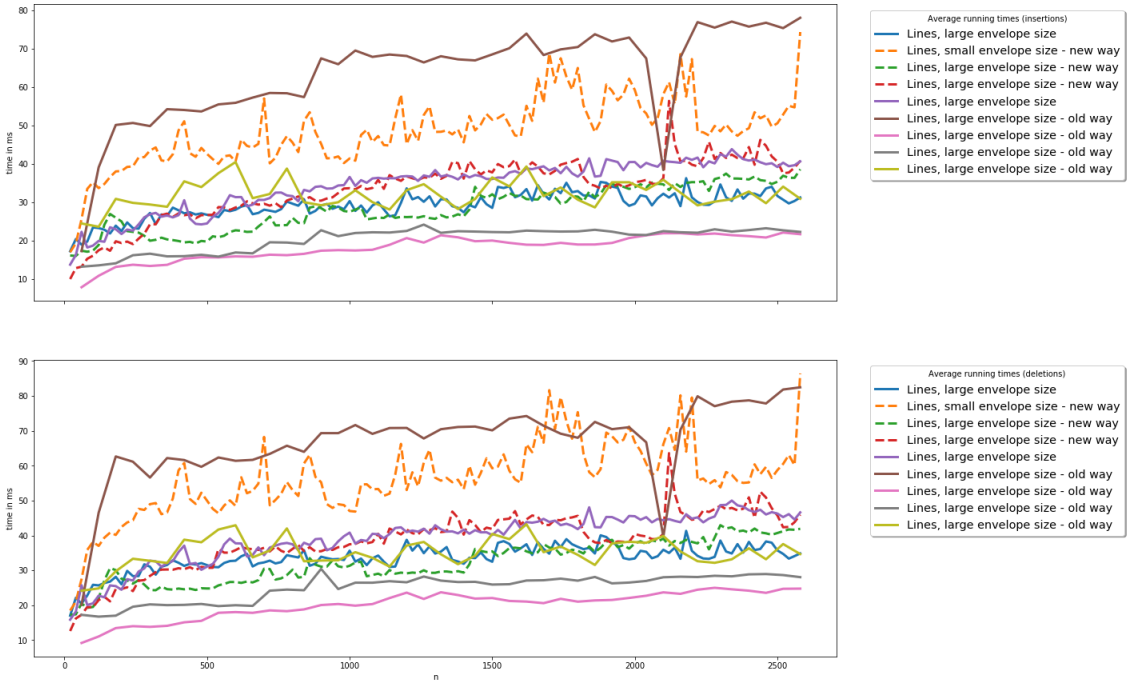
For some unknown reason we see that lines are a lot slower than curves and that curves are less influenced by the size of the envelope. We can see this in Figure A.16 and 6.8.

The evaluation whether using $L(n)$ or traversing n is better uses multiple runs with lines as pseudo-lines. These results are visible in Figure 6.7b and as we see there is no clear difference between the two methods. Similarly, curves are tested which we see in Figure 6.7a and there is no real difference visible.

As expected, we see that the sorted input increases the running time enormously, and they tend to run in linear time or more. Because this way of testing takes a lot of time it was not possible to test larger inputs and even worse: the insertions and deletions started generating stack overflow errors because these are defined recursively. So this is a more serious issue than time is.



(a) curves (range of $a = 100$).



(b) Comparing old and new way for all combinations of inputs for lines.

Figure 6.7: An overview of several experiments to compare the old and new variation of the intersection algorithm using unsorted inputs.

Figure 6.8: An overview of different types inputs but all are sorted.

6.7 The code

The code for this thesis project is written in C++ and Python. C++ is used for the data structures and testing their efficiency. Python is used to process the results, generate plots of the data and generate example plots throughout this report. The main reasons to use C++ are efficiency and the use of the CGAL library, which is also written in C++. CGAL is used because it provides a lot of computational geometric algorithms and data structures and it can be used to provide floating-point precision [3] as much as possible, although this is not used yet in practice.

CGAL requires several other libraries to operate. One of them is *Boost* and is used to create a GUI which is used to create the visual debugging tools. *Qt* is mainly used to generate JSON files. All the C++ is developed with Visual Studio 2019 and CMake. GitHub is used for version control and to publish the algorithms, data structures and results. [4] The Python is developed in so called Python notebooks in *Jupyter lab*. Important Python packages are *mathplotlib*, *numpy*, *scipy* and *pandas*.

[3] Documentation is found here.
[4] Implementation can be found on GitHub.

Chapter 7

Conclusions

In this chapter we present the conclusions about dynamically maintaining convex hulls, envelopes of lines or pseudo-lines based on the results from the experimental evaluation (Chapter 6). We also present recommendations for future work.

7.1 Convex hulls

We have seen multiple ways of maintaining a convex hull dynamically, the most important one being the implementation of Overmars and van Leeuwen. It works correct and fast but it has some room for improvement. Right now it is only possible to store points with a unique x-coordinate so changing this would be a useful adaption. And the bridging algorithm can still be sped up: the implementation for dynamic convex hulls made in Java by Yun Chi traverses the subhulls both at the same time (it can take two steps in total: one in the left subhull and one in the right tree) but the current implementation does either left or right (so one step at a time in one of the trees). The figures showing the running times for dynamic data structure using a tree (Figures A.8 A.6 A.7) all suggest that the $O(\log^2 n)$ running time is correct in practice.

The maintenance of the convex hull using the Delaunay triangulation is faster than the Overmars and van Leeuwen implementation using trees, so it is natural to prefer the Delaunay triangulation over the Overmars and van Leeuwen implementation unless the size of the convex hull is extremely large or when the efficiency is improved in the future. Even the Overmars and van Leeuwen implementation using lists is faster than the version using trees although the running time is worse in theory. The reason behind this is likely that the trees are not optimised, while the implementations of the Delaunay triangulation and *std::vector* are. There are also no tree balances applied, which might slow down operations in some cases.

7.1.1 Running time evaluation

The size of the convex hull clearly has a strong influence on the running time in every different variation of maintaining the convex hull, especially the Delaunay triangulation and dynamic version using a *std::vector* perform awful when the size of the convex hull is extreme while the dynamic version using a tree is surprisingly stable and only affected by the size of the convex hull in a small amount.

Merging and splitting a *std::vector* take both $O(n)$ time in the worst case. This translates to a bad theoretical running time: insertions and deletions take $1 + 2 + 4 + \cdots + \frac{n}{2} + n = 2^0 + 2^1 + 2^2 + \cdots + 2^{\log(n)-1} + 2^{\log n} = \sum_{i=0}^{\log n} 2^i = O(n \log n)$ time in the worst case, when a convex hull is huge. However: the size of the convex hull k is not likely to be around n points, but a lot less so the splitting is merging is not bounded to n, but to k. Hence, a more realistic asymptote would be $\sum_{i=0}^{\log n} \min(2^i, k) = O(n \log(\min(n, k))) = O(n \log k)$ time per insertion or deletion for

$O(1) \leq k \leq n$. This is visible in the results (Figures A.4, A.5 and 6.3). It can be the case that the final convex hull is smaller than intermediate results in which case k is not a good representation but in general it should work. The Delaunay triangulation varies between $O(\sqrt{n})$ and $O(n^2)$ time in practice. This last one is unexpected, because the worst case running time of the Delaunay triangulation should be $O(n)$ time in the worst case.

The effect of convex hull size is less dramatic for the Overmars and van Leeuwen implementation using trees. We saw that insertions or deletions take $O(\log^2 n)$ time in theory (Section 3) but the results (Figures A.6 A.8 A.7) suggest that the size of the convex hull k influences the running time mildly. The difference can be explained in the same way as with the *std::vector* version: the size of the tree to split runs up to k, not to n so now we can change the original running time of $\sum_{i=0}^{\log n} \log(2^i) = O(\log^2 n)$ to $\sum_{i=0}^{\log n} \log(\min(2^i, k)) = O(\log(n)\log(k))$ for $O(1) \leq k \leq n$.

7.1.2 Outlier detection

The only reason why testing for normality was necessary was to test which data points were outliers. What we did now was just testing for normality and detecting outliers using boxplots but there were only one or two cases were removing outliers made sense. Some test results had multiple dense regions and that might be the problem why the data was not normal distributed. This could be checked using histograms and the data could be transposed to a normal distribution using Box-Cox transformations [3] if this is really desired. These methods were not used because outlier detection was not as relevant as expected, but it might be in the future.

7.2 Envelopes

We saw in Chapter 4 that the Overmars and van Leeuwen proposal for envelopes is not practical although it works in theory. It is better to use a convex hull of the duals of the lines with for example a Delaunay triangulation. This is only possible for normal lines though, but it is also possible to maintain the envelope of pseudo-lines now.

7.2.1 Correctness

Although the CGAL algorithm is not suitable for dynamic applications it still can be used to verify the correctness of the dynamic algorithms. This is unfortunately not implemented yet, mainly because it was not straightforward to use. Hence the only method of checking for correctness is the debugger.

The implementation of the intersection algorithm still contains some errors unfortunately which show up sometimes after a number of insertions. It is unknown where these errors come from but it appears that envelopes of parabolas are more error prone than envelopes of lines. The only way to test this is by visual inspection unfortunately so a good first step would be to validate envelopes automatically, most likely using CGAL.

Although the intersection algorithm is not perfect yet, we can conclude that it works in practice as well and not only in theory. The only thing left to do now is to remove its sharp edges (inaccuracies, bugs and other possible errors), optimise it and add support of more types of pseudo-lines. Then it becomes possible to extend the structures with other functions to answer queries.

7.2.2 Running time evaluation

It is difficult to see a difference between the $O(\log^2 n)$ and $O(\log^3 n)$ asymptotes in the plots (see Appendix A). There is also barely a difference between the two different ways of finding the cases using $v.L$ or $E(v)$. And if we look at the differences between smaller envelope sizes (approximately $20 \leq k \leq 50$) and larger envelope sizes ($k \approx \frac{n}{2}$), we see that the differences of the running times

are small and this is similar to what we saw for the Overmars and van Leeuwen data structures to maintain convex hulls. But the difference in running times exist so this implies that there is an influence of the size of the envelope, although it is small.

If we apply the same logic for the convex hull running time analysis here, we have that the running time depends on the size of the envelope k. This makes sense because the small difference we saw actually suggests this. For the case of $v.L$ we get the same asymptote of $O(\log(n)\log(k))$ time for insertions and deletions by the same reasoning. For the case of $E(v)$ we have $\sum_{i=0}^{\log n} \log^2(\min(2^i, k)) = O(\log(n)\log^2(k))$ time for insertions and deletions. But the differences between $O(\log(n)\log^2(k))$ and $O(\log(n)\log(k))$ are either small or insignificant in practice because the experiments do not show a clear difference between the two variations.

7.3 Future work

Height balancing trees should decrease the variations in the running times and improve the average running time. A lot of the running times (Figures A.4 A.12 for example) vary a lot which means that the true asymptote is harder to approximate. Some even show multiple dense regions which weakens the approximations of asymptotes. This is especially the case for convex hulls with the dynamic versions using lists but less for trees. Balancing the trees should decrease these effects and make it more predictable how the running times behave and what the true asymptote is. It also solves the running time issue with sorted inputs and the stack overflows. The stack overflows occurred once the tree had a depth of around 220. This tree would not have this depth in the first place if it was height balanced but now imagine a height balanced tree with a depth of 220. The depth of a tree is $\log n$ where n is the size of the set stored in the tree, so if $\log n = 220$ then we have that $n = 2^{220} \approx 1.68 * 10^{66}$ and this kind of input sizes are completely unrealistic in practice and that is why height balancing is important.

There is more optimisation possible, like merging of envelopes is implemented in a simple but inefficient way. Updating $n.lmax$ and $n.rmax$ in a smarter way might save some unnecessary tree traversals and thus save time. These are updated right now using a tree traversal, but there might be smarter solutions. But most of all it is important to start with making the intersection algorithm work correctly. CGAL also can help with verifying if the dynamic result is correct.

CGAL can be used to create polynomial equations of various forms [1]. However, it was not initially clear how to solve x for $f(x) = g(x)$. It is of course possible to figure out how to solve this by hand but it would just be more convenient if this is possible as it makes it a lot easier to add different kinds of pseudo-lines. Unit disks and x-monotone lines should also work now once they are implemented. This report provides a guideline to do this.

The problem with large numbers (while comparing pseudo-lines) should be solved further. Generating the data sets and running experiments were difficult in the beginning because this issue kept popping up. One of the causes was the sampling of inputs from a large sample space, so reducing the sample space worked. It could also be that large numbers are not the issue but small ones, so floating point errors might be another issue that not has been tackled or that lines close to each other intersect far away from the origin.

[1] Documentation can be found here

Bibliography

[1] Pankaj K. Agarwal, Ravid Cohen, Dan Halperin, and Wolfgang Mulzer. Dynamic Maintenance of the Lower Envelope of Pseudo-Lines. *35th European Workshop on Computational Geometry (EuroCG'19)*, pages 1–7, 2019. iii, 1, 2, 3, 32, 34

[2] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. *Annual Symposium on Foundations of Computer Science - Proceedings*, 14186:617–626, 2002. 24

[3] R. J. Carroll and David Ruppert. On prediction and the power transformation family. *Biometrika*, 68(3):609–615, 1981. 54

[4] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(4):361–368, 1996. 9

[5] Timothy M. Chan. Three problems about dynamic convex hulls. In *International Journal of Computational Geometry and Applications*, volume 22, pages 341–364. World Scientific Publishing Company, aug 2012. 24

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2010. 7

[7] Mark de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry*. Springer, 3rd edition, 2008. 8, 9, 12, 26

[8] Mashhood Ishaque and Csaba D. Tóth. Relative convex hulls in semi-dynamic arrangements. *Algorithmica*, 68(2):448–482, aug 2014. 24

[9] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley & Sons, 6th edition, 2014. 42

[10] Eunjin Oh and Hee Kap Ahn. Dynamic geodesic convex hulls in dynamic simple polygons. *Leibniz International Proceedings in Informatics, LIPIcs*, 77(51):51:1–51:15, 2017. 24

[11] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. iii, 1, 2, 14, 23, 29

[12] F. P. Preparata. An Optimal Real-Time Algorithm for Planar Convex Hulls. *Communications of the ACM*, 22(7):402–405, 1979. 24

[13] Franco P. Preparata and Michael Ian Shamos. *Computational geometry. An introduction*. Springer, 1985. 8, 9

[14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R.J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex

Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T. Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vinícius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, and Yoshiki Vázquez-Baeza. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, mar 2020. 24

Appendix A

Running time plots

A.1 Convex hulls

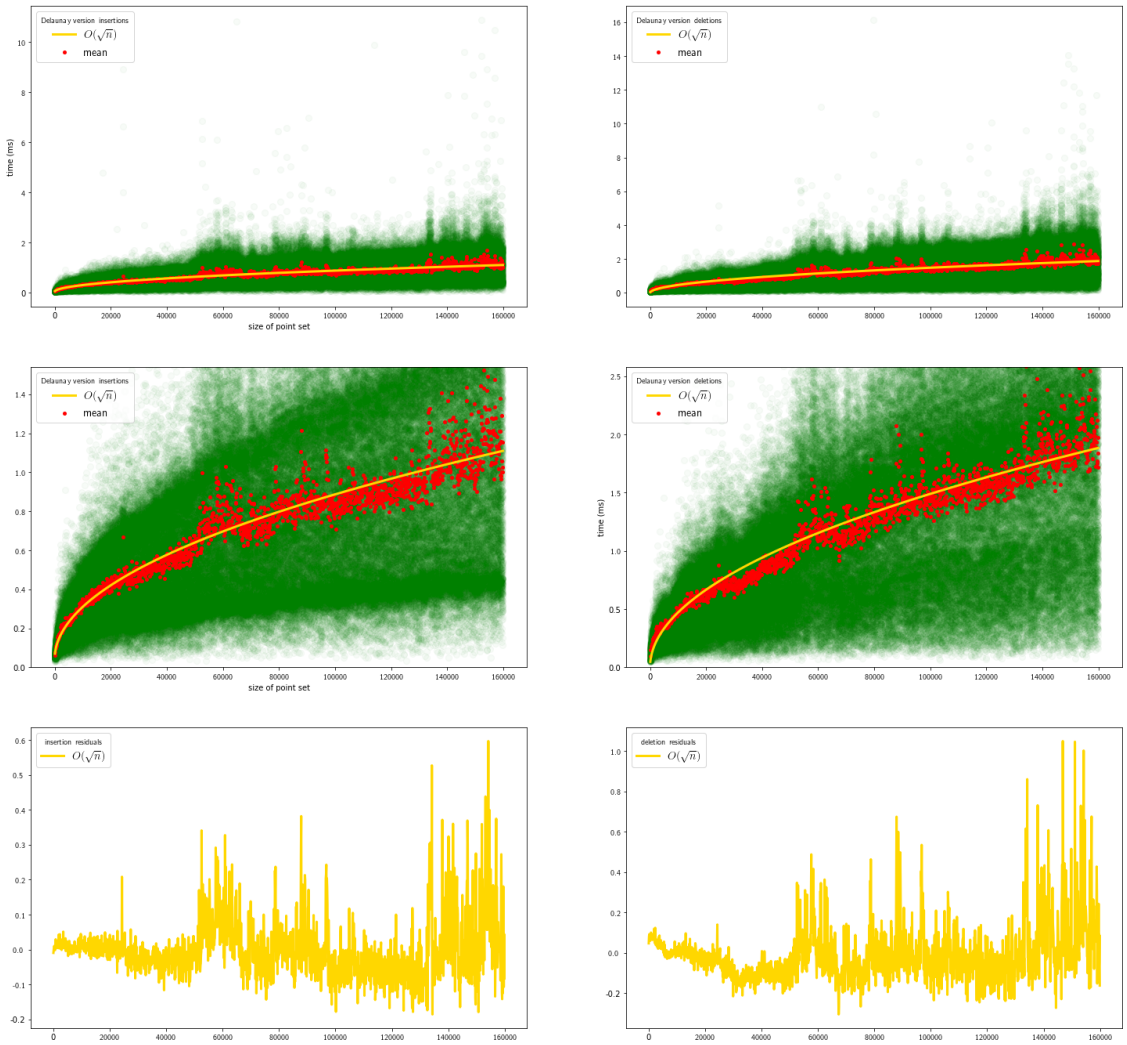


Figure A.1: Results of testing insertions and deletions using a Delaunay triangulation, points are sampled in a disc.

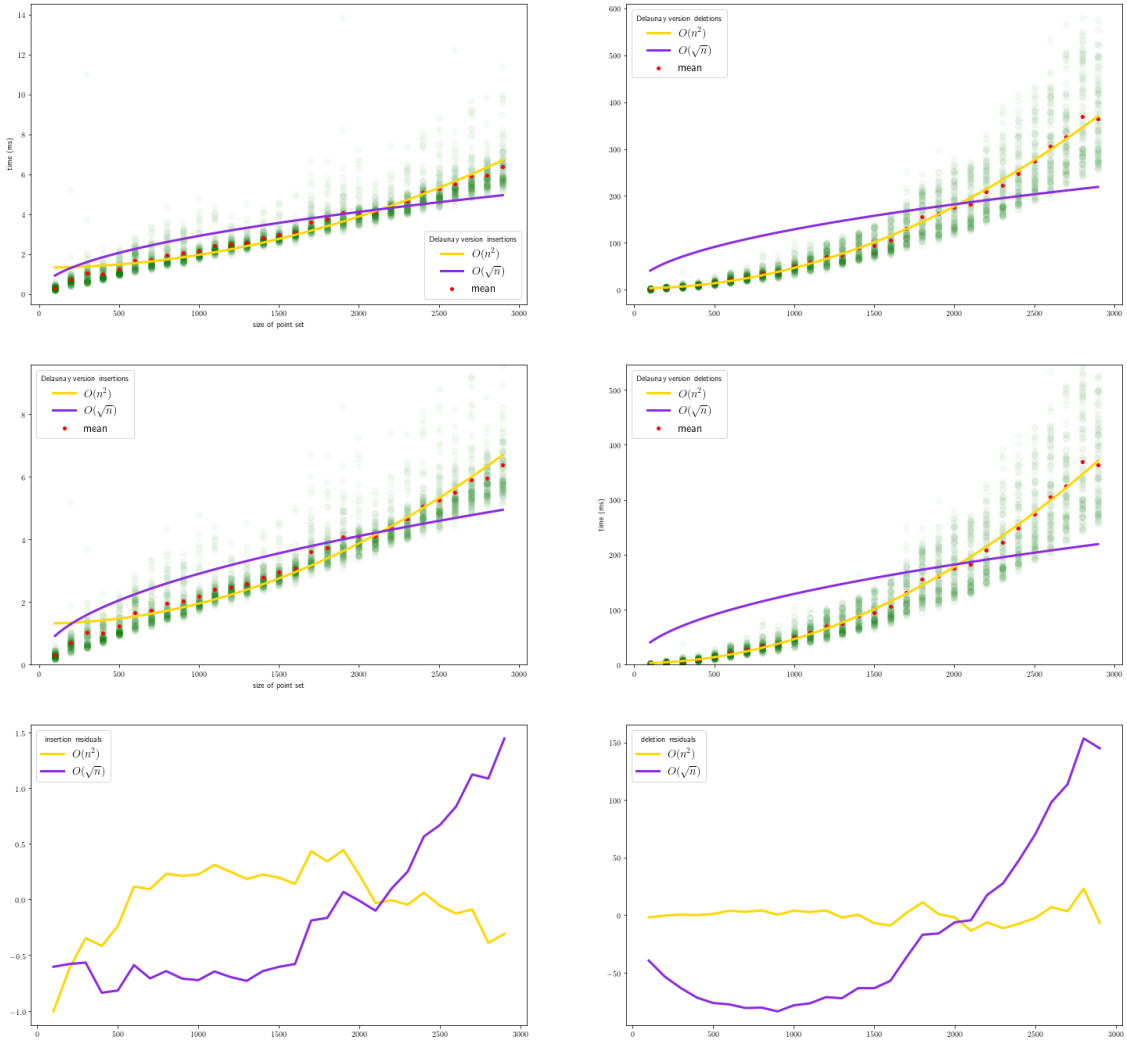


Figure A.2: Results of testing insertions and deletions using a Delaunay triangulation, points are sampled on a disc.

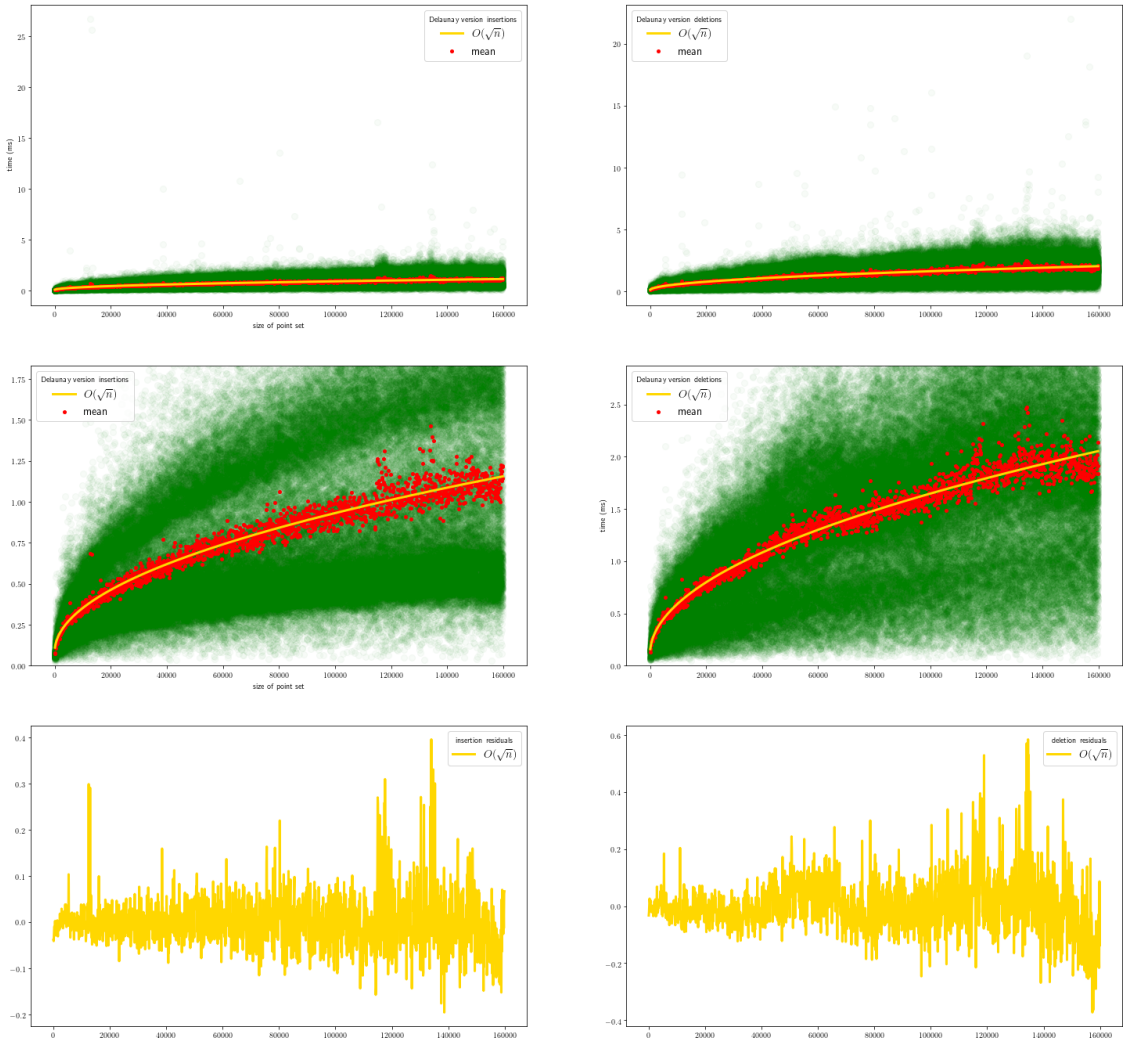


Figure A.3: Results of testing insertions and deletions using a Delaunay triangulation, points are sampled in a square.

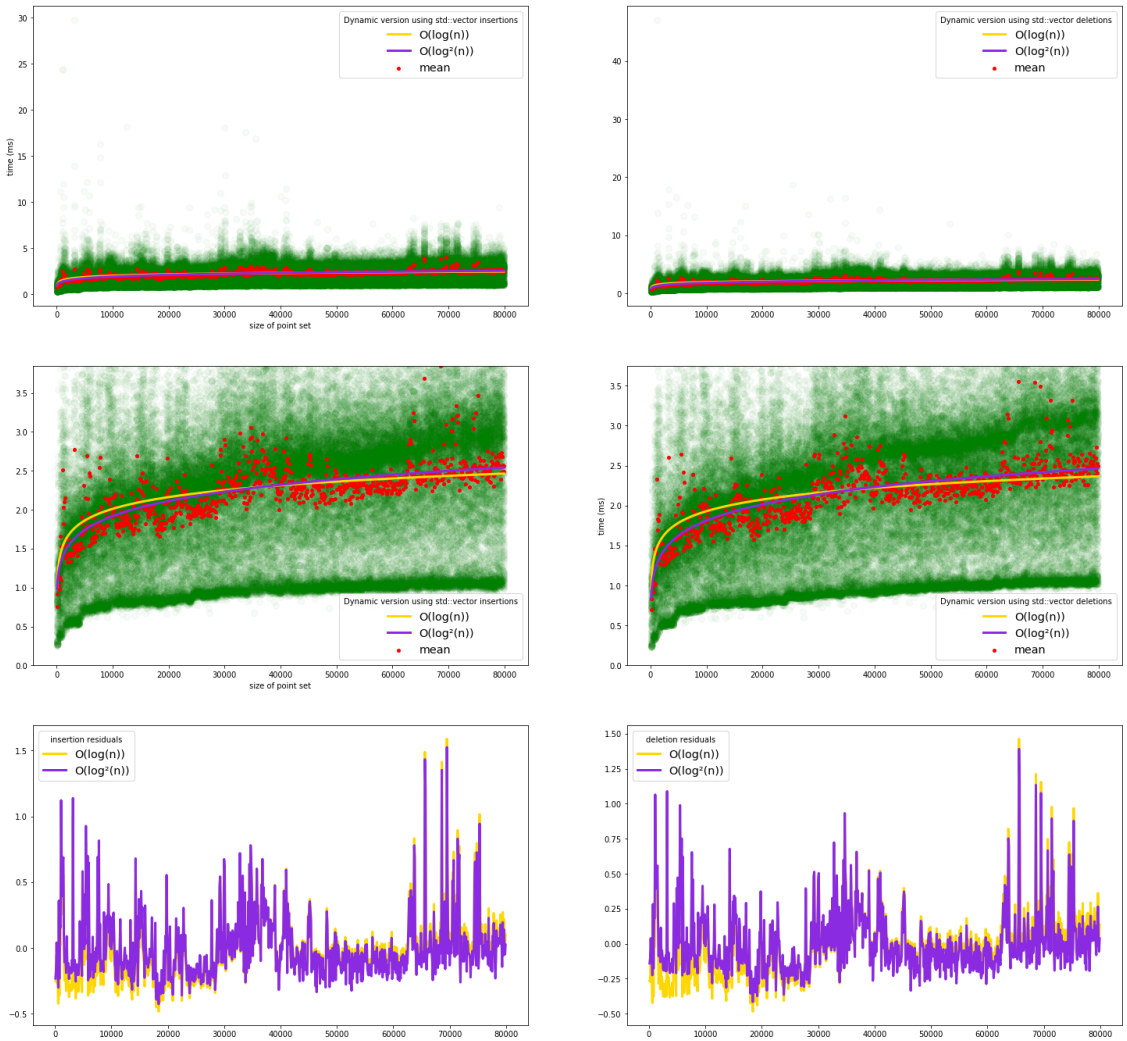


Figure A.4: Results of testing insertions and deletions in the dynamic data structure using a vector, points are sampled in a disc.

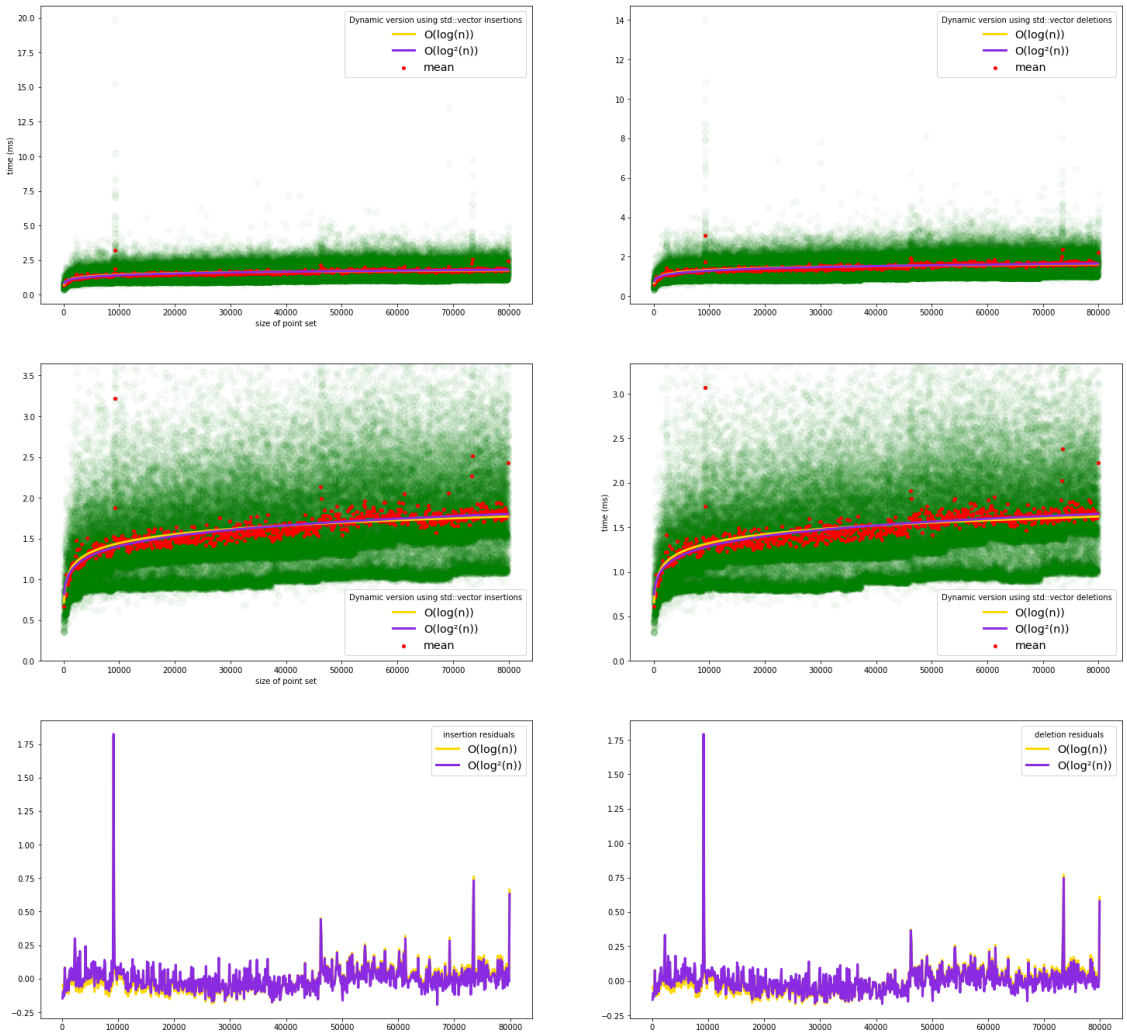


Figure A.5: Results of testing insertions and deletions in the dynamic data structure using a vector, points are sampled in a square.

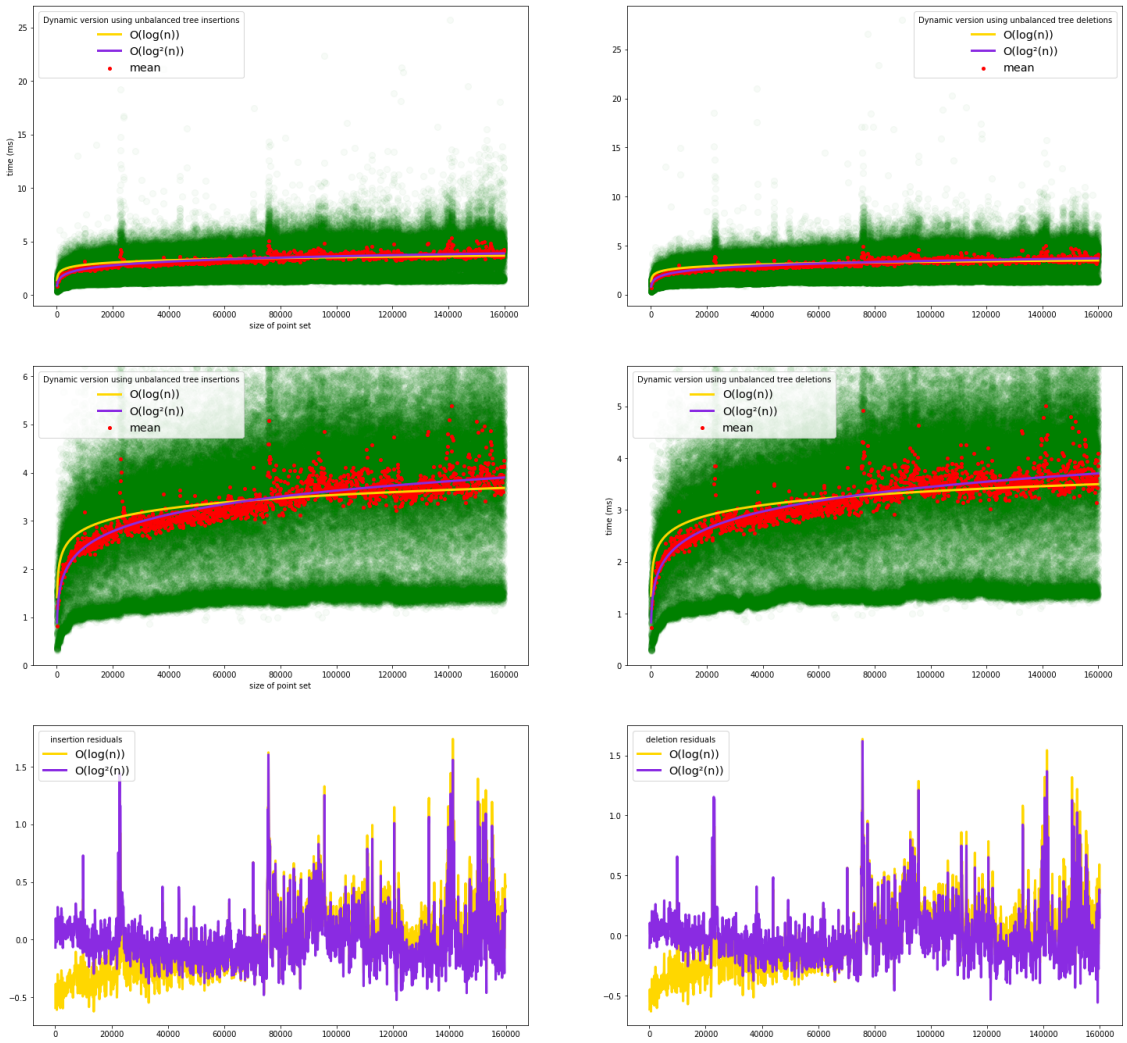


Figure A.6: Results of testing insertions and deletions in the dynamic data structure using an unbalanced tree, points are sampled in a disc.

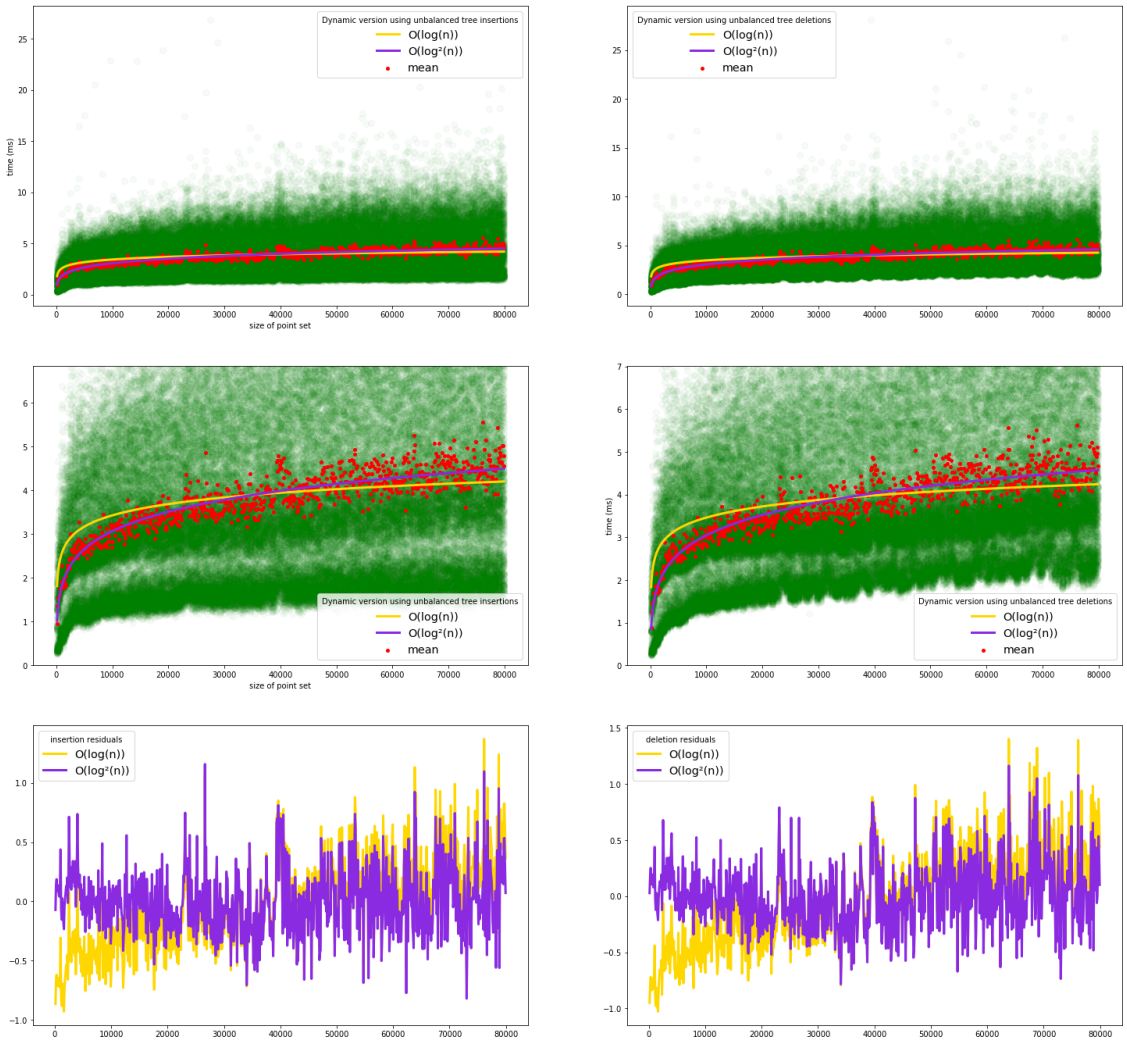


Figure A.7: Results of testing insertions and deletions in the dynamic data structure using an unbalanced tree, points are sampled on a disc.

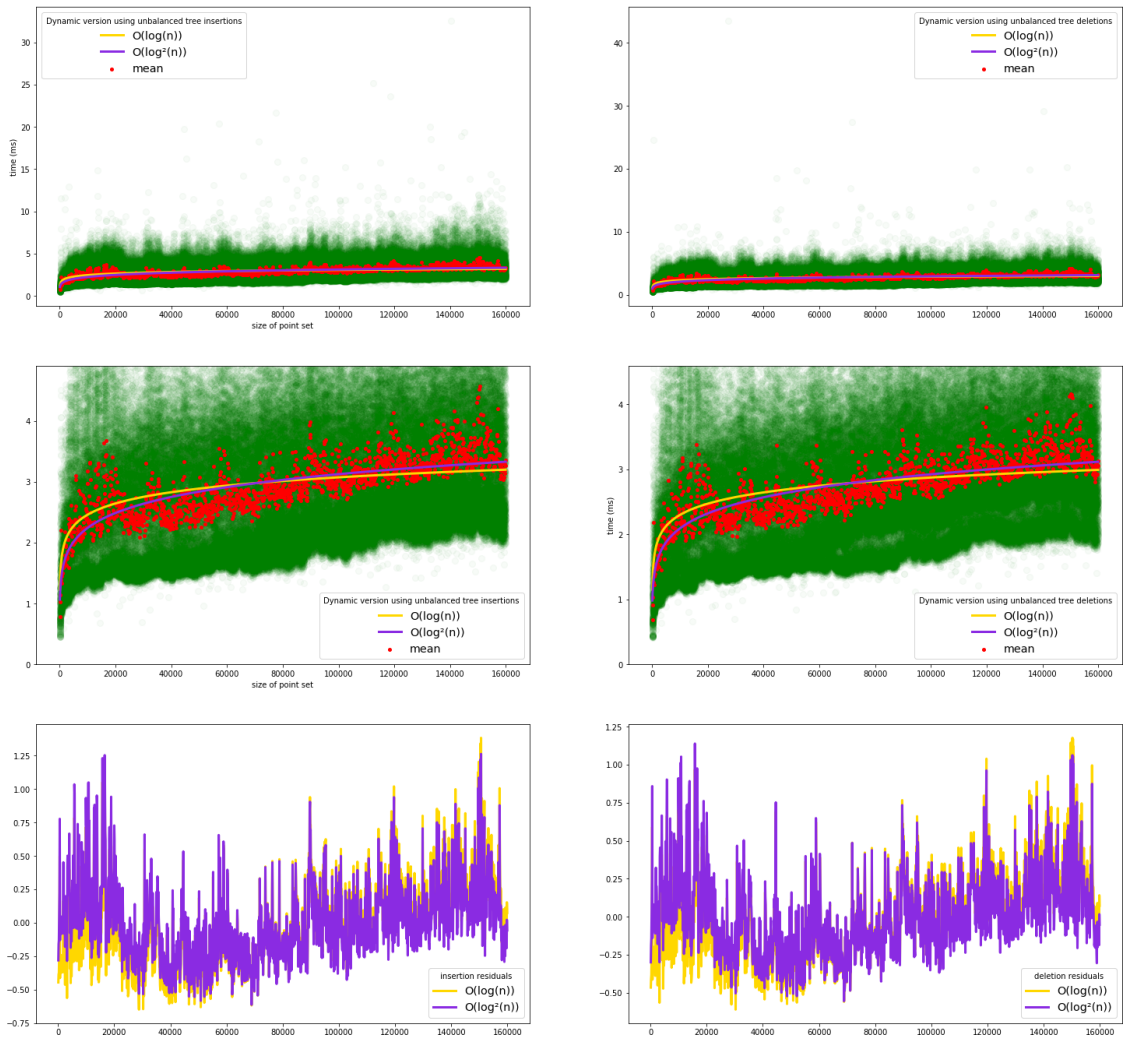


Figure A.8: Results of testing insertions and deletions in the dynamic data structure using an unbalanced tree, points are sampled in a square.

A.2 Envelopes

A.2.1 Unsorted input

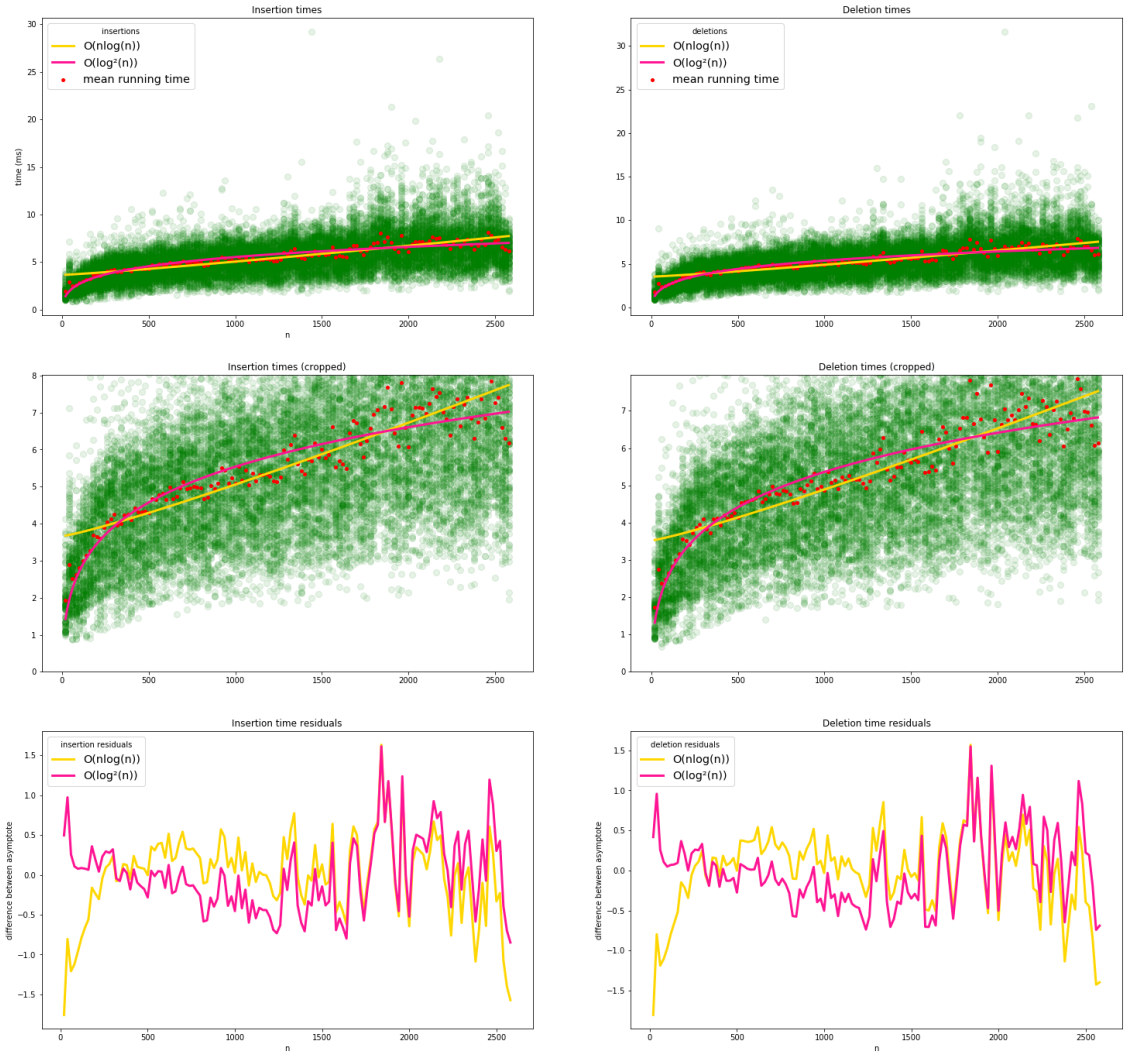


Figure A.12: Results of testing insertions and deletions of curves with a small size of envelope.

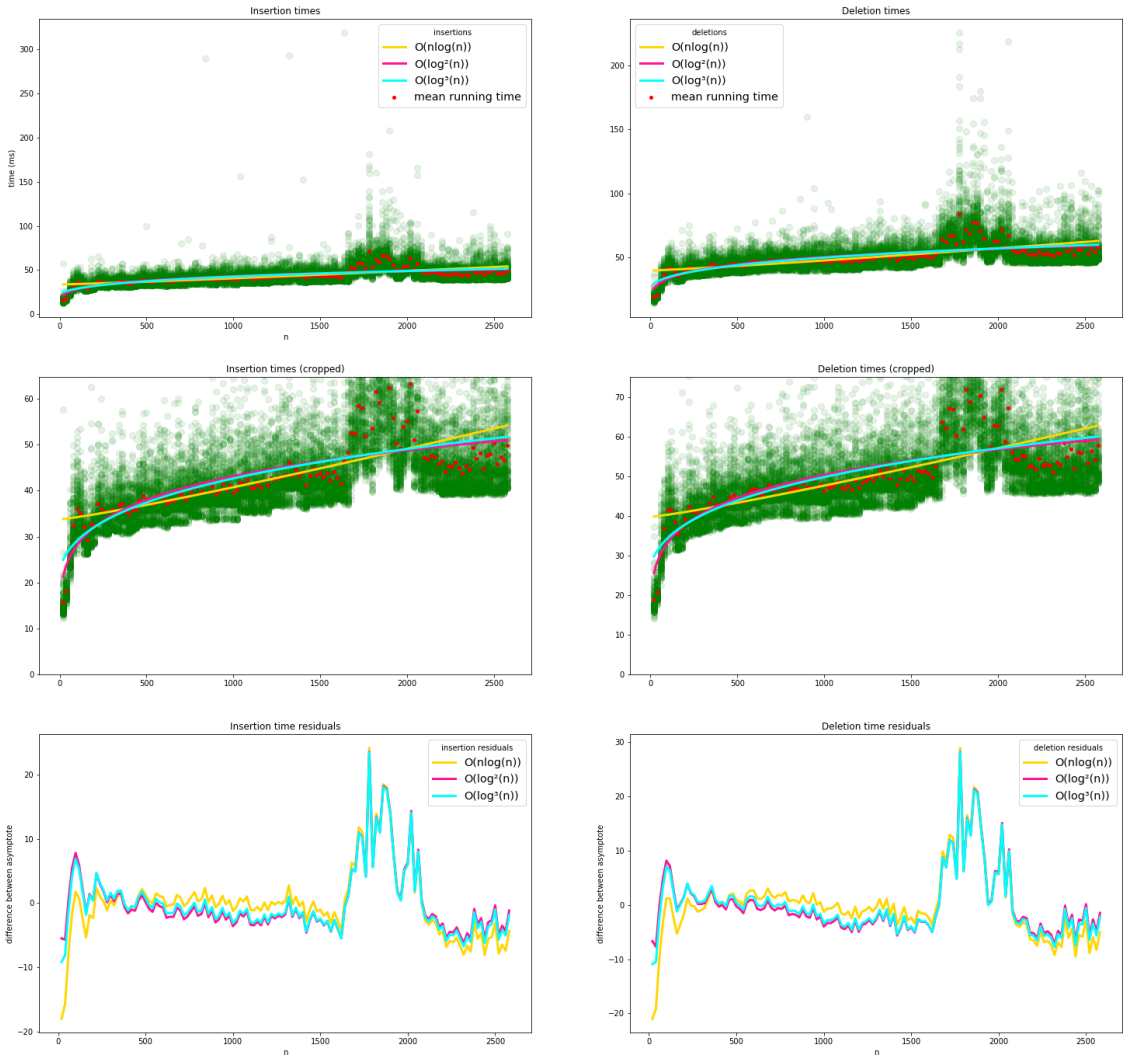


Figure A.9: Results of testing insertions and deletions of lines with a small size of envelope.

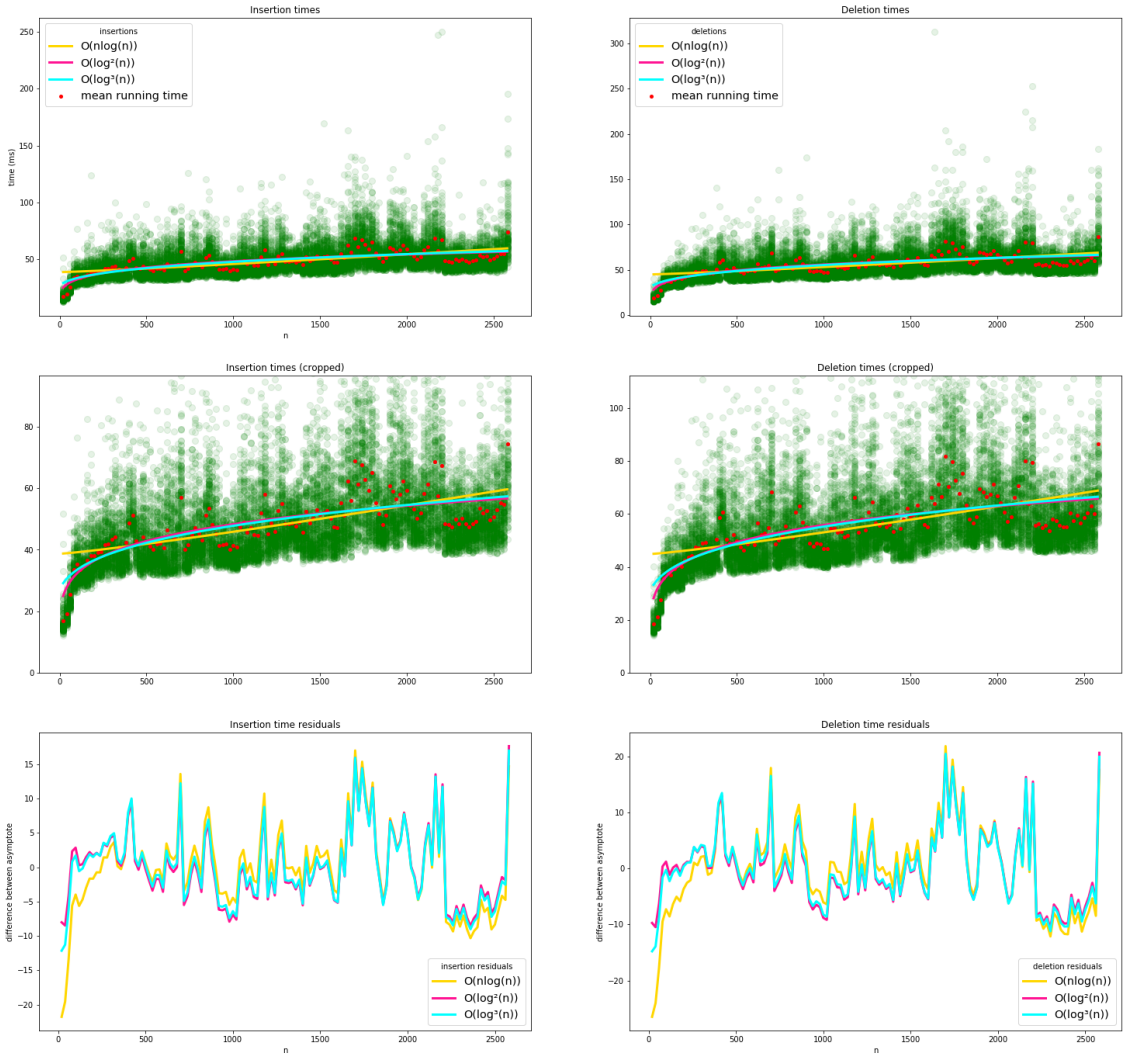


Figure A.10: Results of testing insertions and deletions of lines with a small size of envelope, new way.

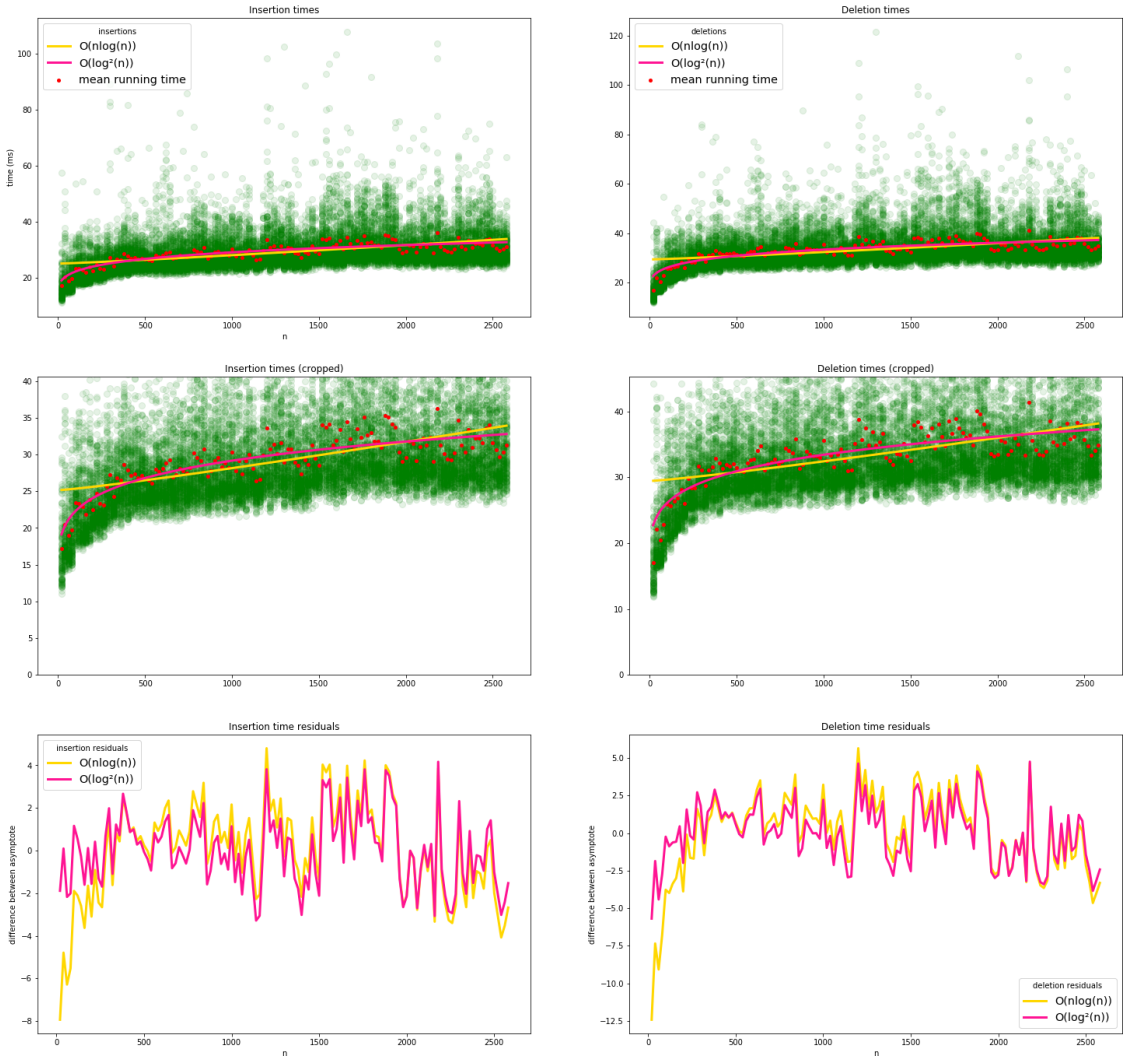


Figure A.11: Results of testing insertions and deletions of lines with a large size of envelope.

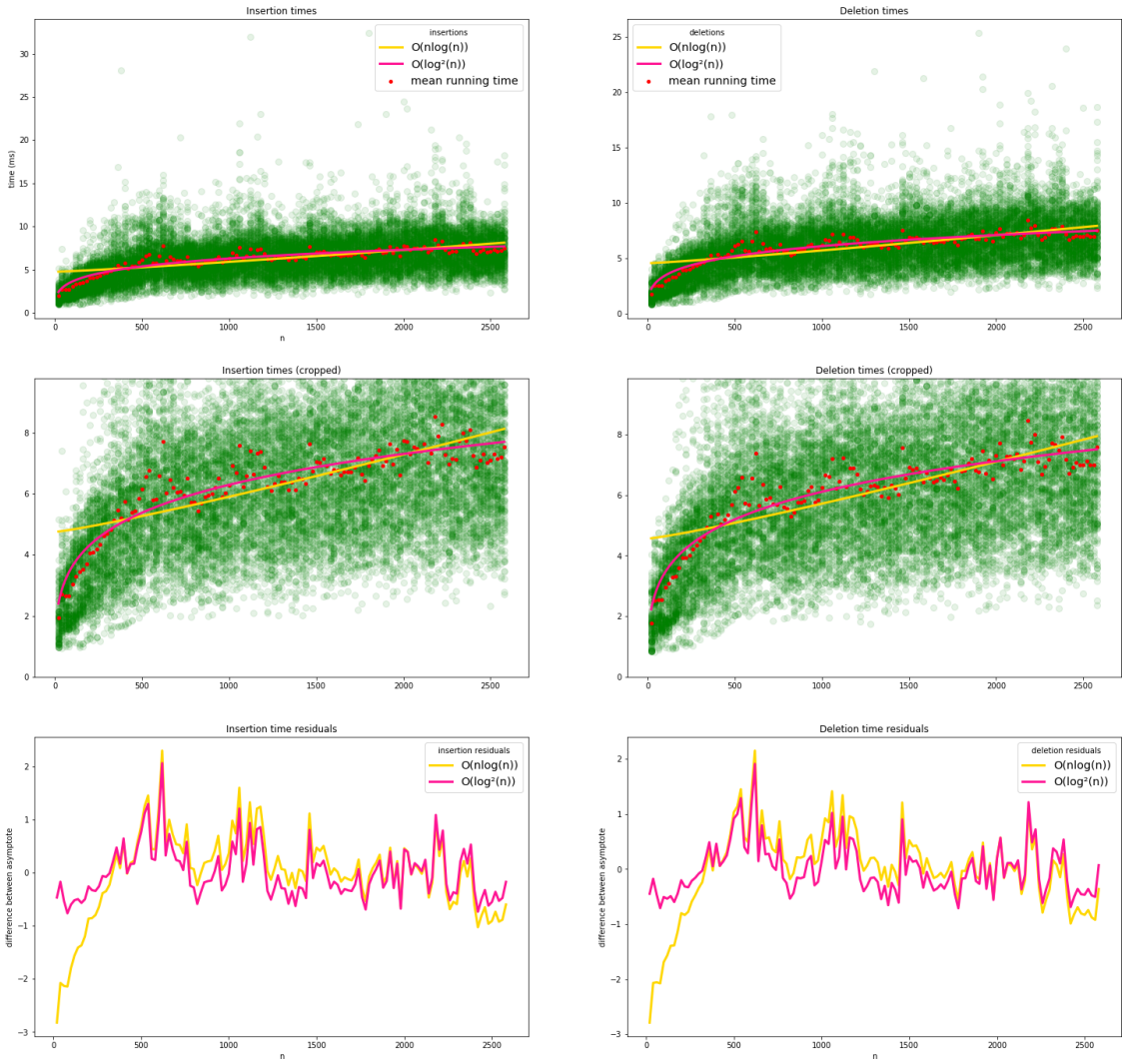


Figure A.13: Results of testing insertions and deletions of curves with a medium size of envelope.

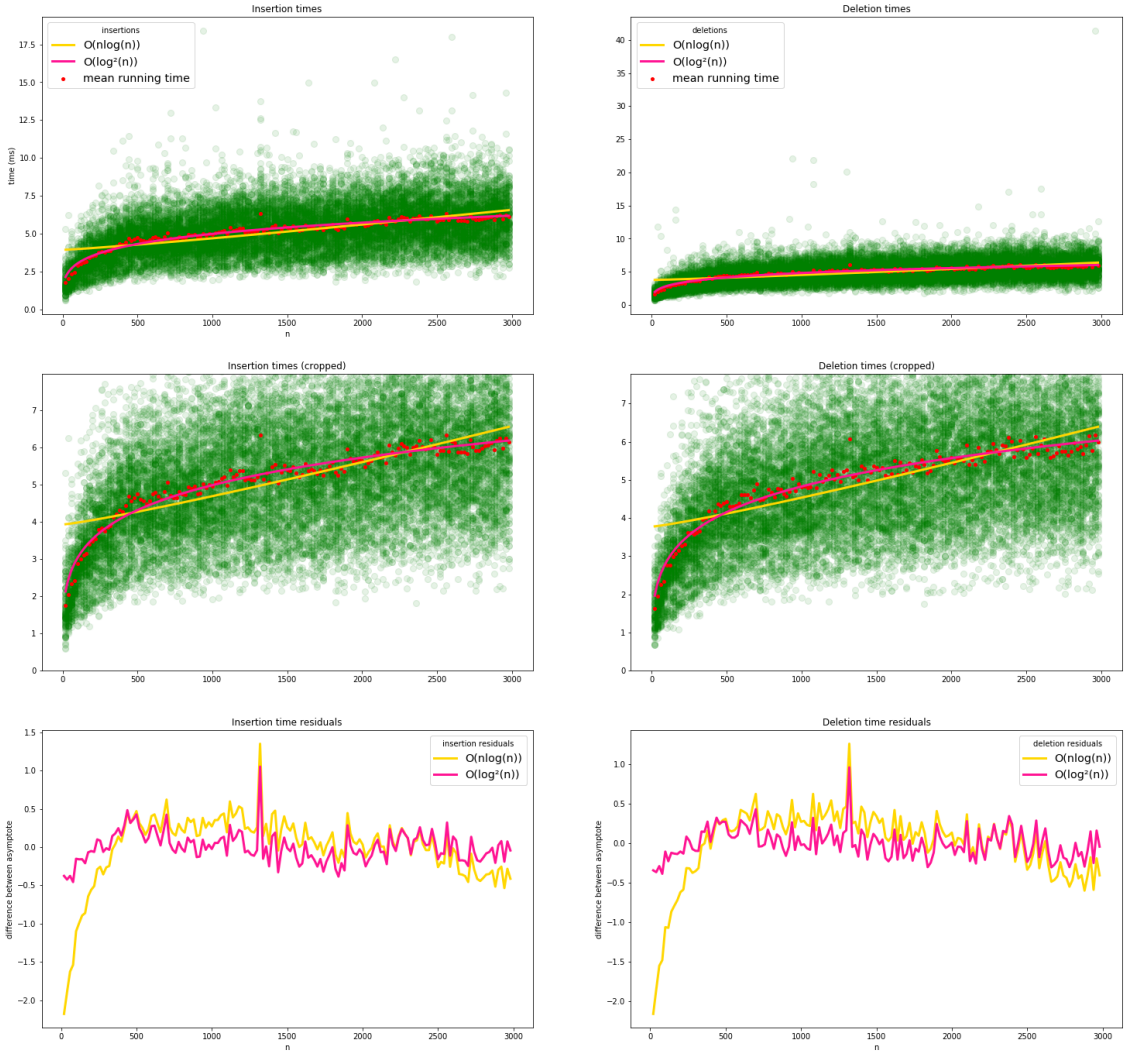


Figure A.14: Results of testing insertions and deletions of curves with a medium size of envelope, new way.

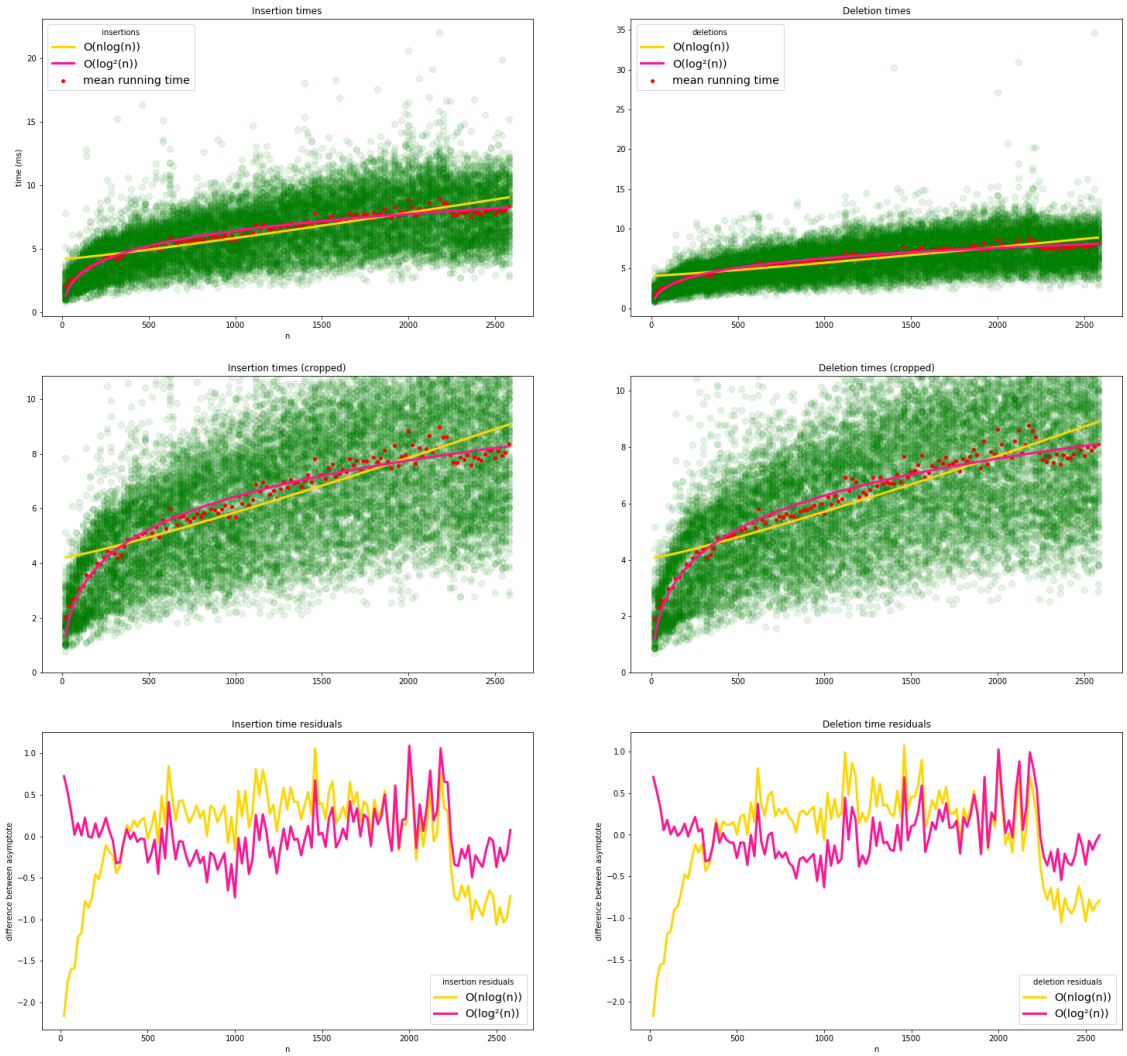


Figure A.15: Results of testing insertions and deletions of curves with a large size of envelope.

Figure A.16: Comparing old and new way for all combinations of inputs.

A.2.2 Sorted input

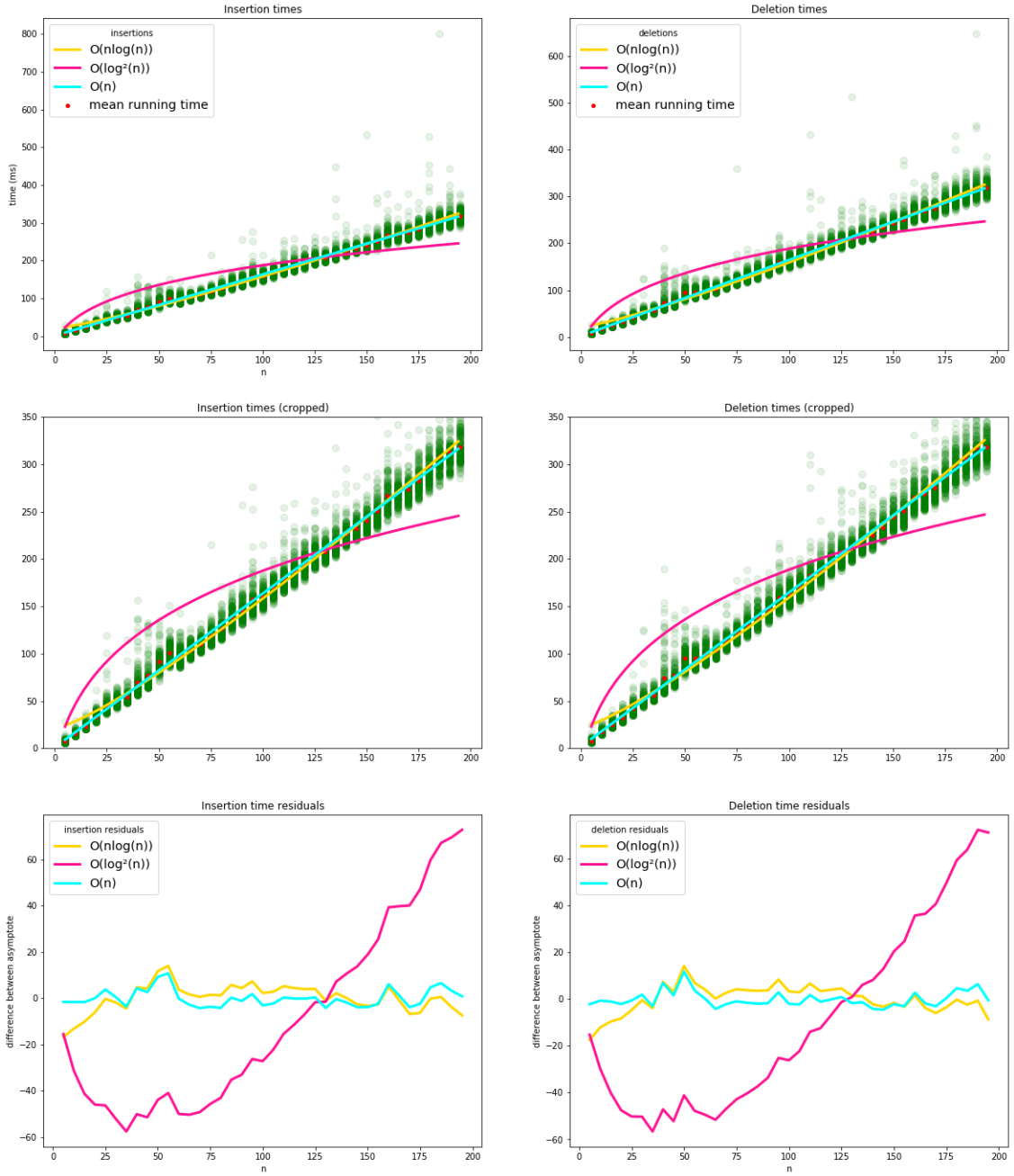


Figure A.17: Results of testing insertions and deletions of lines with a small size of envelopeusing a sorted input.

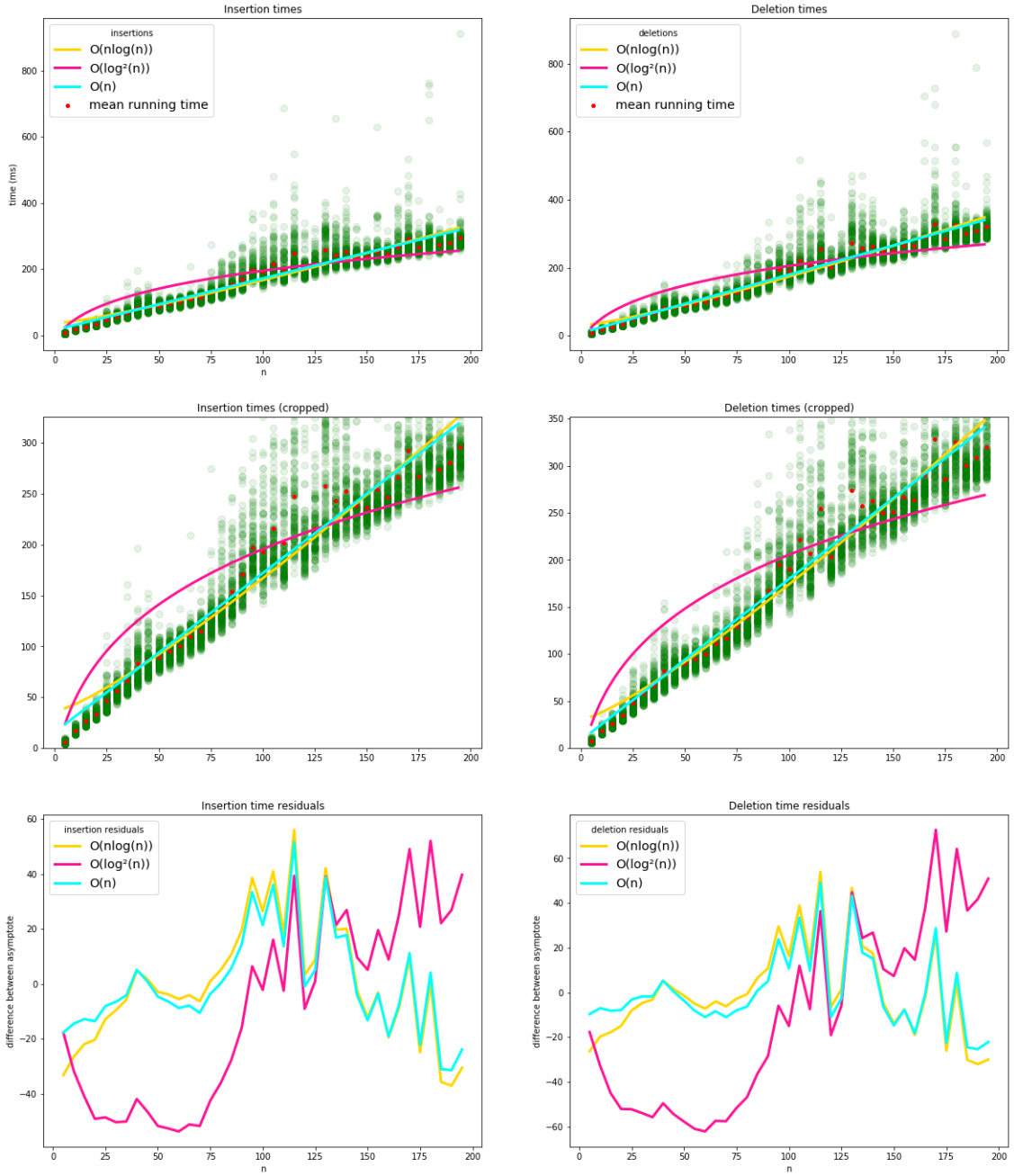


Figure A.18: Results of testing insertions and deletions of lines with a large size of envelopeusing a sorted input.

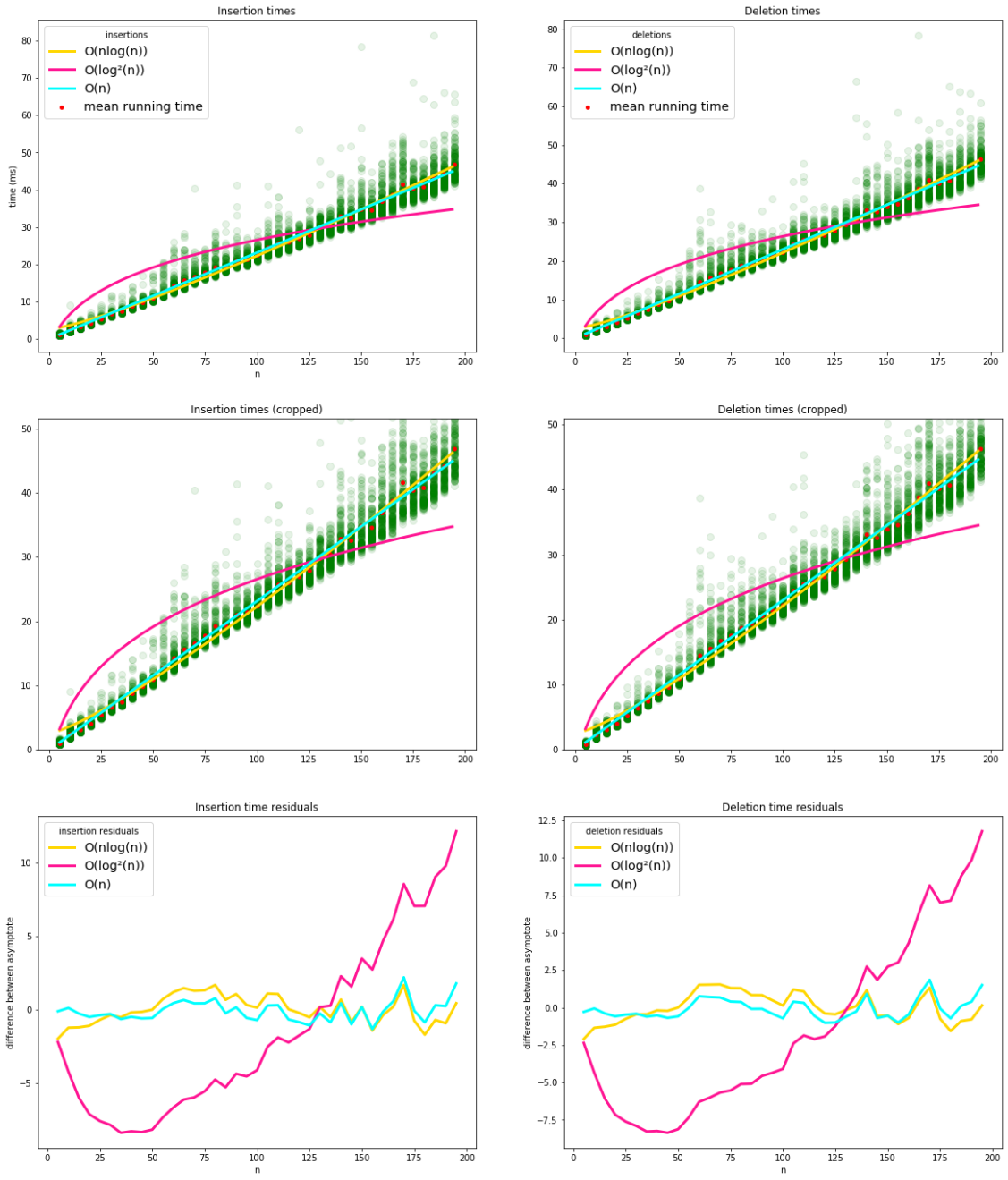


Figure A.19: Results of testing insertions and deletions of curves with a small size of envelopeusing a sorted input.