CONVEX HULL PROBLEMS

by

Raimi A. Rufai
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Dr. Dana S. Richards, Dissertation Director
_____	Dr. Fei Li, Committee Member
_____	Dr. Jyh-Ming Lien, Committee Member
_____	Dr. Walter D. Morris, Jr., Committee Member
_____	Dr. Sanjeev Setia, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2015
George Mason University
Fairfax, VA

CONVEX HULL PROBLEMS

by

Raimi A. Rufai
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

Dr. Dana S. Richards, Dissertation Director

Dr. Fei Li, Committee Member

Dr. Jyh-Ming Lien, Committee Member

Dr. Walter D. Morris, Jr., Committee Member

Dr. Sanjeev Setia, Department Chair

Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: April 30, 2015

Spring Semester 2015
George Mason University
Fairfax, VA

Convex Hull Problems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Raimi A. Rufai
Master of Science
King Fahd University of Petroleum & Minerals, 2003
Bachelor of Science
University of Ilorin, 1998

Director: Dr. Dana S. Richards, Professor
Department of Computer Science

Spring 2015
George Mason University
Fairfax, VA

Dedication

To my beloved mother, who recently passed on to the great beyond. And to my generous father, who had departed before her. May they be reunited in heavenly splendor.

Acknowledgments

I am extremely grateful to my advisor Dr. Dana Richards for his support, encouragement, generosity, and superb advising. I feel extremely privileged to have had the chance to work closely with him. I am also indebted to my committee members, Dr. Fei Li, Dr. Jyh-Ming Lien, and Dr. Walter Morris for valuable discussions, countless feedback, and generosity with their time.

I am greatly indebted to my friends Aliou Sylla and Mayowa Aregbesola, who hosted me during my sojourns to the Fairfax area to complete my dissertation.

I am also indebted to my current employer, SAP Labs, Inc., for supporting me in many ways through the course of this project. I am equally indebted to my work colleagues and team mates – Angelo, Aqib, Jean-Nicolas, Joel, Mitch, Mourad, Nasir, Nicola, Qiu Wen, Robin, Theodore, and Yasamin – for their support and encouragement.

I am also indebted to my previous employer, Sonex Enterprises, Inc. I am particularly indebted my supervisor, Dr. Moataz Ahmad and his family, and the company president Mr. Dean Xenos.

I would be remiss if I do not express my unreserved gratitude to Dr. David C. Rine, who was my first advisor when I arrived at GMU. Thank you Dr. Rine for your generosity to and caring for my family and me, at a time when we were totally new to America and to the Fairfax area. You essentially guided us every step of the way, from getting our visas to renting our first apartment, and the list goes on.

I am equally indebted to the staff at the international programs office, especially Amy Moffit, Cynthia Tasaki, and Brian Lenius.

I would also like to thank Ms. Therese Michael at the Computer Science Department office for the many things she helped to take care for me, being a remote student for more than half of the time, and for her attention to detail and her genuinely palpable concern for our success.

My debt of gratitude to my parents, both of whom have now passed on, is incalculable. They gave my siblings and me everything they had, so we could attain our dreams. They taught us discipline, morality, kindness, justice, and generosity by doing.

My thanks also go to my siblings, who have always been there for me. My first computer, compiler and programming book were gifts from by brother Latif. That marked the beginning of my journey into computers. I sometimes wonder how things would have turned out without these gifts.

Finally, I would like to thank my family – my generous spouse, Ganiat, and my kids, Mariam and Adam, who endured my several long absences from home. I cannot thank them enough.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

CONVEX HULL PROBLEMS

Raimi A. Rufai, PhD

George Mason University, 2015

Dissertation Director: Dr. Dana S. Richards

The convex hull problem is an important problem in computational geometry with such diverse applications as clustering, robot motion planning, convex relaxation, image processing, collision detection, infectious disease tracking, nuclear leak tracking, extent estimation, among many others.

The convex hull is a well-studied problem with a large body of results and algorithms in a variety of contexts. In this thesis, we consider three contexts: when only an approximate convex hull is required, when the input points come from a (potentially unbounded) data stream, and when layers of concentric convex hulls are required.

The first context applies when input point sets may contain errors from noise or from rounding, or when the accuracy provided by exact algorithms are simply not required. This thesis proposes a framework for examining convex hull approximation algorithms so that they can be better compared. The framework is then used to assess a number of existing algorithms and new algorithms proposed in the thesis. This framework can help an engineer to select the most appropriate algorithm for their scenario and to analyze new algorithms for this problem. Moreover, our new algorithms exhibit better space, time, and error bounds than existing ones.

The second context applies to a base station in a wireless sensor network that receives incoming input points and must maintain a running convex hull within a memory constraint. This thesis proposes a new streaming algorithm that processes each point in time $O(\log k)$ where k is the memory constraint, while maintaining very good accuracy.

And finally, the last context applies when all the convex layers are sought. This has a variety of applications from robust estimation to pattern recognition. Existing algorithms for this problem either do not achieve optimal $\mathcal{O}(n \log n)$ runtime and linear space, or are overly complex and difficult to implement and use in practice. This thesis remedies this situation by proposing a novel algorithm that is both simple and optimal. The simplicity is achieved by independently computing four sets of monotone convex layers in $\mathcal{O}(n \log n)$ time and linear space. These are then merged together in $\mathcal{O}(n \log n)$ time.

Chapter 1: Introduction

This thesis addresses three problems – convex hull approximation, streaming algorithms for the convex hull, and the convex layers problem. Since the *convex hull* is central to the three problems, we begin by summarizing known results about the convex hull. We then briefly introduce each of the three problems.

1.1 Convex Hull

The *convex hull* of a finite point set S in a Euclidean space, often denoted as $\mathrm{conv}(S)$, can be defined as the intersection of all half-spaces that contain S. An equivalent definition for the convex hull is as the union of all convex combinations[1] of the elements of S. There are more definitions of the convex hull that are all provably equivalent to each other (see for instance [44]).

The *convex hull problem* is to compute the convex hull of a given set of points. The convex hull problem occurs as a subproblem in a large number of computational geometry, computer graphics, computational statistics, image processing and even spatial database and optimization problems.

Several exact algorithms for finding the convex hull have been proposed. These algorithms can be broadly categorized as either *offline*, where all the points in S are available prior to computing $\mathrm{conv}(S)$, or *online*, where the points arrive incrementally.

Efficient offline algorithms typically run in $\mathcal{O}(n \log n)$ [23, 47] . At first, it was thought that this was the best that could be achieved, until optimal output sensitive algorithms were discovered. An algorithm whose complexity depends on the size of the output is

[1]Recall that a convex combination of m points, $p_1, p_2, \cdots, p_m$, is defined as any point p satisfying $p = \sum_{i=1}^{m} \lambda_i p_i$ such that each $\lambda_i \geq 0$ and $\sum_{i=1}^{m} \lambda_i = 1$.

termed *output-sensitive*. One of the earliest output-sensitive algorithms is Jarvis march which runs in $\mathcal{O}(nk)$ time [32], where $k \leq n$ is the number of points on the hull. A few output-sensitive algorithms run in $\mathcal{O}(n \log k)$ [37, 12]. It is easy to see that when the output vertices of a convex hull algorithm are required to be in a sorted order, the sorting problem can be reduced to the convex hull problem. This directly suggests a lower bound of $\Omega(n \log n)$. Yao [58] proved $\Omega(n \log n)$ worst-case lower bound for the general convex hull problem, by showing that ternary decision trees for this problem have $n!$ leaves and thus, $\Omega(n \log n)$ height.

In fact, sorting algorithms have often inspired new convex hull algorithms. For instance, the quickhull [22, 10] algorithm drew inspiration from quick sort and the divide-and-conquer algorithm of Preparata and Hong [47] could have been appropriately named merge hull for its resemblance to merge sort. Jarvis march reminds us of selection sort, while Graham scan is somewhat reminiscent of insertion sort.

There is, however, one crucial difference between the convex hull and sorting. In the sorting problem, every element of the input ends up in the output. This contrasts with the convex hull problem where not every input point necessarily remains in the output.

1.2 Convex Hull Approximation

When dealing with input point sets that are themselves approximate, a fast convex hull approximation algorithm might be more practical as long as the approximation error is reasonably bounded. We shall have a lot more to say about convex hull approximation algorithms and their error bounds in Chapter 2.

1.3 Streaming Algorithms for the Convex Hull

An algorithm is called *online*, if it gets its input data a piece at a time and must compute a partial result, which it updates while the rest of the input comes through. Preparata [46]

proposed a $\Theta(n \log n)$ online realtime algorithm that incrementally updates its current convex hull with each arrival of a new point in $\mathcal{O}(\log n)$ time. Preparata's algorithm improves on Shamos's algorithm, which, though also runs in $\Theta(n \log n)$, could only process new points in time $\mathcal{O}\left((\log n)^2\right)$ [46]. Another optimal online algorithm was later discovered by Kallay [33]. There are also algorithms that run in $\mathcal{O}(n)$ expected time [5, 22, 4] for planar and 3D point sets, if the point sets satisfy certain conditions, such as being uniformly distributed.

Online algorithms that are further restricted by how much memory they are allowed (i.e. a *memory budget*) are called *streaming algorithms*. Thus results that require more memory than the allowed budget must make decisions on what is worth keeping and what must be discarded. An example of a streaming algorithm for the convex hull is Hershberger and Suri's algorithm [28, 30, 29]. Their algorithm maintains extreme points in k uniformly spaced directions and another k extreme points in adaptively sampled directions. Their algorithm has a distance error of $\mathcal{O}\left(1/k^2\right)$. This distance is defined as the height of the tallest uncertainty triangle. The uncertainty triangle of an edge e_i is the triangle formed by extending its immediate neighbor edges e_{i-1} and e_{i+1} until they meet, assuming all such triangles are bounded. Chapter 3 introduces streaming algorithms and proposes a new convex hull streaming algorithm.

1.4 Convex Layers

The *convex layers problem*, also known as the *onion peeling problem*, can be defined as follows: Given a set of points P in the plane, construct a set of non-intersecting convex polygons, such as would be constructed by iteratively constructing the convex hull of the points left after all points on all previously constructed convex polygons are deleted. Figure 1.1 is an example of the convex layers for a set of forty-five randomly-generated points. Convex layers will be dicussed further in Chapter 4.

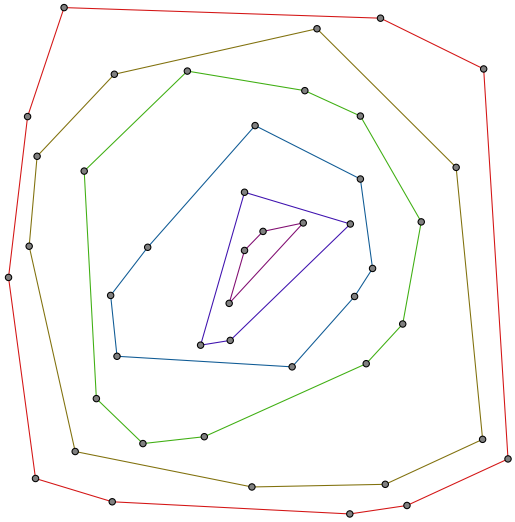

Figure 1.1: Convex Layers

1.5 Organization of Thesis

The next three chapters present the contributions of this thesis. Chapter 2 introduces an analytical framework for describing convex hull approximation algorithms and then applies the framework to a number of published algorithms as well as to new algorithms proposed in this thesis. Chapter 3 presents a new streaming algorithm for the convex hull and analyzes its runtime and error bounds. Chapter 4 presents a new simple algorithm for the convex layers problem, and presents correctness and optimality proofs. Finally, Chapter 5 concludes the dissertation.

Chapter 2: Convex Hull Approximation

This chapter introduces the problem of approximating convex hulls. The convex hull of a finite set of points $P \subset \mathbb{R}^2$ is the smallest simple polygon that contains P. In Section 2.1, we introduce the problem, present our contributions in Section 2.2, and conclude the chapter in Section 2.3.

2.1 Introduction

In many domains, where the convex hull is applied, point sets may contain errors from noise or from rounding. In either case, it might be more desirable to compute an approximate hull using a very fast algorithm than to compute an exact one using a costlier algorithm as long as the approximation error is reasonably bounded.

2.2 Contributions

The contributions in this chapter are of two types. The first is a proposed analytical framework for describing convex hull approximation algorithms using a common set of attributes. This framework is then applied to a number of algorithms found in the literature with the goal that these algorithms can be better compared, and gaps in published knowledge about them discovered and filled. The second type of contributions proposed are new algorithms with improved runtime and error bounds.

2.2.1 Framework for Approximate Convex Hull Algorithms

This section proposes a common framework for discussing known approximation algorithms for the convex hull, with the goal that these algorithms can be more easily compared.

Underlying Convex Hull Definition. Often, convex hull algorithms derive directly from one of the many equivalent definitions of the convex hull. This attribute is used to map algorithms to definitions that could have inspired them, even if so only in hindsight. The algorithms have been mostly inspired by two of the definitions of the convex hull mentioned above – the convex hull defined as the intersection of half-planes (*intersection idea*) and as the union of convex combinations (*union idea*).

Hull Approximation Type. Does the algorithm compute an inner, an outer, or a mid-hull?

Analogous Sorting Algorithm. Sorting algorithms also often inform convex hull algorithms. Is there such a mapping? If yes, what is the mapping for a convex hull approximation algorithm?

Generalization to d-space. How well does an algorithm generalize to higher dimensions?

Input Space. What assumptions does the algorithm make about the input space? Does the algorithm work only with integer coordinates or does it apply more generally?

Does the algorithm have corner cases, that might lead it to fail for some classes of inputs? What are these classes of inputs and how could such degeneracies be handled?

Complexity. What are the the time- and space-complexities of a convex hull approximation algorithm?.

Accuracy Measures. How good is an algorithm in terms of accuracy under various accuracy measures?

Parallelizability. How easily can the algorithm be parallelized?

Streaming model. Can this algorithm process streaming data, where input points arrive one at a time and memory is limited? If not, how easily can it be adapted to handle streaming data?

All the algorithms discussed below fit into one of two models [1], shown in Algorithm 2.1 and Algorithm 2.2 below. Individual algorithms, however, differ in how each algorithm defines the COMPUTESUBSET(S, k) or the COMPUTEHALFPLANES(S, k) functions.

Note that algorithms that follow the COMPUTESUBSET model tend to be inner hulls, while those following the COMPUTEHALFPLANES model tend to be outer hulls.

Algorithm 2.1: APPROXSUBSET(S, k)

Input : A point set S and a parameter $k \geq 3$
Output: The vertex set of an approximate convex hull of S in sorted order

▷ Compute a subset, L, of S ◁
1 $L \leftarrow$ COMPUTESUBSET(S, k)
▷ Return the convex hull of L, computed with a linear-time convex hull algorithm ◁
2 **return** $\text{conv}(L)$

Algorithm 2.2: APPROXHALFPLANES(S, k)

Input : A point set S and a parameter $k \geq 3$
Output: The vertex set of an approximate convex hull of S in sorted order

▷ Compute the half-planes of S, $H = \{h_i : h_i \text{ is a half-plane containing } S\}$ ◁
1 $H \leftarrow$ COMPUTEHALFPLANES(S, k)
▷ Return the intersection of the half-planes H. ◁
2 **return** $\bigcap_i (h_i)$

[1] Models such as these are sometimes called control abstractions, or meta-algorithms.

2.2.1.1 Accuracy Measures

Approximate convex hull algorithms have been evaluated using a number of accuracy measures. These measures usually come in two forms:

Relative Distance Measure. Given a finite point set S, let P be the vertices of $\mathrm{conv}(S)$ and P' the vertices of $\mathrm{conv}^*(S)$ where $\mathrm{conv}^*(S)$ denotes some approximate convex hull of S. The relative distance measure of P to P' is defined as:

$$err_{\delta,\mathrm{diam}}(P, P') = \delta(P, P')/\,\mathrm{diam}(P) \tag{2.2.1}$$

i.e. the distance between the true hull and the approximate hull relative to the diameter of S. The distance $\delta(\cdot,\cdot)$ most commonly used is the Hausdorff distance[2].

Relative Extent. Let P and P' be similarly defined as above. We define the relative extent measure between P and P' with respect to an extent measure g as follows:

$$err_g(P, P') = \frac{|g(P) - g(P')|}{g(P)} \tag{2.2.2}$$

where $g(\cdot)$ is some extent measure of the given point set, such as diameter, area, or even cardinality. For instance, if we define the function g as the area, then the relative area measure for approximating a convex polygon P by P' can be expressed as follows:

$$err_{\mathrm{area}}(P, P') = \frac{|\mathrm{area}(P) - \mathrm{area}(P')|}{\mathrm{area}(P)} \tag{2.2.3}$$

[2]The Hausdorff distance from a finite point set P to another Q, $\delta(P, Q)$ is the maximum distance between any point in P to its nearest point in Q, i.e. $\delta(P, Q) = \max(\max_{p\in P} \min_{q\in Q} \|p - q\|, \max_{q\in Q} \min_{p\in P} \|q - p\|)$.

Another interesting question immediately comes to mind here: Can we analyze these algorithms using a common template, including a common set of accuracy measures?

2.2.2 Approximate Convex Hull Algorithms

This section uses the framework presented above to discuss several published convex hull approximation algorithms.

2.2.2.1 Klette's Algorithm [38]

Klette [38] describes two kinds of approximate convex hulls: an outer and an inner hull. The inner hull, true to its name, is wholly contained in the exact hull. Its vertices form a subset of those of the exact convex hull. The outer hull always contains the exact convex hull. Klette's algorithm, presented in Algorithm 2.3 below[3], takes an integer k and a point set S as input parameters. It starts out by constructing k directions $\Delta = \{0, \frac{1}{k}2\pi, \frac{2}{k}2\pi, \cdots, \frac{k-1}{k}2\pi\}$. An extreme point p_i for a direction $\alpha_i \in \Delta$ is a point such that the line l_i, perpendicular to the direction α_i, passing through p_i divides the plane into two half-planes one of which wholly contains S. The set of extreme points for all the directions forms the vertex set for the approximate inner hull A_k. The intersection of the half-planes forms the outer hull H_k. Because the extreme points p_i found this way are not necessarily distinct, both A_k and H_k might actually have cardinalities smaller than k.

Below, we discuss Klette's algorithm using the framework given earlier in Section 2.2.2.

Underlying Convex Hull Definition. Klette's inner hull approximation algorithm uses the idea of union of convex combinations (simplices), while his outer hull approximation uses that of intersection of half-planes.

Hull Approximation Type. Klette defined both an inner and an outer hull approximation in his paper.

[3]Note that Algorithm 2.3 along with Algorithm 2.1 only compute the inner hull

Algorithm 2.3: COMPUTESUBSET(S, k)

Input : A point set S and a parameter $k \geq 3$
Output: A subset of S

$L \leftarrow \emptyset$
foreach $\alpha \in \{\frac{2\pi}{k} i | i \in [0, k-1]\}$ **do**
    Get the points P that are extreme in direction α
    $L \leftarrow L \cup P$
▷ Return the subset L ◁
return L

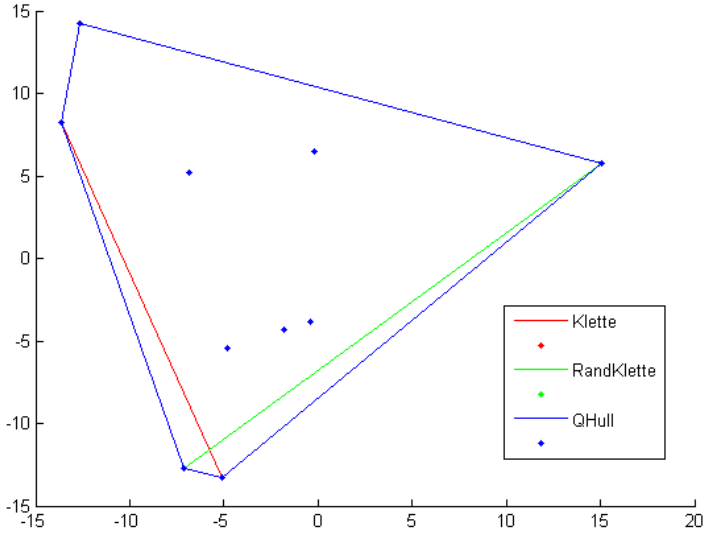

Figure 2.1: A sample run using $|S| = 10$, $k = 4$.

Analogous Sorting Algorithm. The operation of selecting the maximum length point along each direction reminds one of selection sort. However, the consideration of k directions is reminiscent of bucket sort. It seems this algorithm does not fit into a single sorting "bucket".

Generalization to d-space. Klette's algorithm was rediscovered some fifteen years later by Xu et al [57] in 1998 and generalized to higher dimensions and to real inputs.

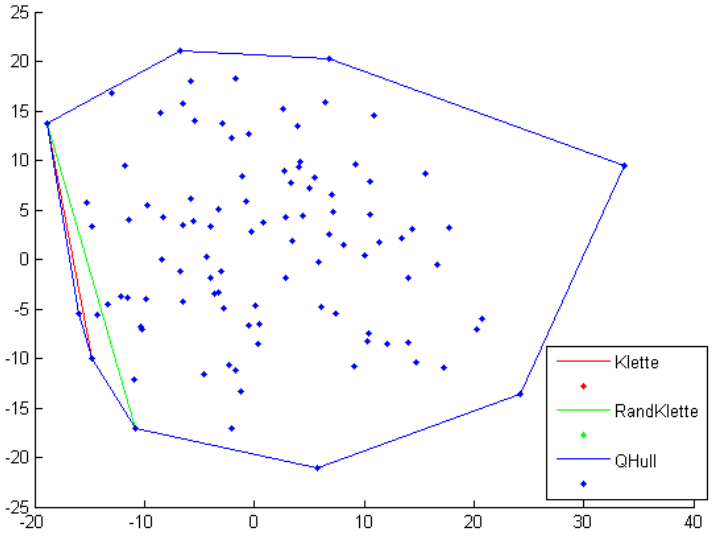

Figure 2.2: A sample run using $|S| = 100$ and $k = 10$

Input Space. As originally presented, Klette's algorithm assumes the input points have integer coordinates[4], however, the algorithm can be made to work with arbitrary precision points as demonstrated by Xu et al [57] in their generalization of the algorithm to higher dimensions.

Complexity. Klette's algorithm runs in $\mathcal{O}(nk)$ time. Kim and Stojmenovic [36] suggest ways to improve it to $\mathcal{O}(n \log k)$ worst-case and $\mathcal{O}(n + \sqrt{n} \log k)$ average-case time. The approach is to start with $d = 2$ directions and double the number of directions repeatedly until $d \geq k$.

Klette proved that the inner hull always converges to the true hull given a large enough number of directions. Žunić [55] found that m^2 directions are necessary to guarantee convergence, where m is the diameter of the point set (i.e. the maximum number of grid cells orthogonally spanned by the point set). Another interesting property of the inner hull is that all the vertices of the inner hull $V(A_k(S))$ are vertices

[4]Point sets with integer coordinates are sometime called *grid* or *digital* point sets.

of the exact hull $V(\text{conv}(S))$ (i.e. $V(A_k(S)) \subseteq V(\text{conv}(S))$).

Accuracy Measures. Klette reports that empirically, the inner hull algorithms perform extremely well under the area measure

$$\frac{\text{area}(A_k(S))}{\text{area}(\text{conv}(S))} = .977$$

on average for $k = 8$ and point set of sizes ranging from 11 to 5175. A sample output from our implementation of this algorithm is shown in Figure 2.1, in comparison with an exact convex hull algorithm and a randomized version where the k directions are randomly generated. For this particular run, we found the area ratios 0.996862 and 0.976057 for the Klette and the randomized Klette respectively.

Notwithstanding these empirical results, the worst-case relative distance error is $\tan(1/k)$ [36]. Since $1/k \in (0, 1]$, the relative distance error is no greater than than $\tan(1.0) \approx 0.0174$.

The area error bound is $\mathcal{O}(1)$ [36].

Parallelizability. The computations for each direction α in Algorithm COMPUTESUBSET Algorithm 2.3 can be done independently and thus handed off to a different processor. This will result in a parallel runtime of $T_p = \mathcal{O}\left(\frac{nk}{p}\right)$, where p is the number of processors in a PRAM model of computation.

Streaming model. Klette's algorithm assumes that the whole input point set is available to the algorithm, so it does not fit into the streaming model.

2.2.2.2 Bentley et al.'s Algorithm [3]

The algorithm of Bentley, Faust and Prepaprata is one of the earliest published for this problem. Given a point set $S \in \mathbb{R}^2$ and a parameter k, the algorithm finds the two points

with the minimum and maximum x-coordinates (ties split using the y-coordinates) and adds them to its subset L. It then splits the point set into k vertical strips, each of width $\mathrm{diam}(S)/k$, where $\mathrm{diam}(S)$ denotes the diameter of S. Within each strip, the algorithm finds the two extreme points with the minimum and maximum y-coordinates and adds them to L. Finally, the algorithm computes the convex hull of L using a version of Graham's scan that skips the sorting step. The pseudocode for the algorithm is given in Algorithm 2.4.

Algorithm 2.4: COMPUTESUBSET(S, k) in Bentley et al.'s Algorithm

Input : A point set S and a parameter $k \geq 3$
Output: A subset of S

$L \leftarrow \emptyset$
$pmax = \underset{p_i=(x_i,y_i)\in S}{\operatorname{argmax}} \; x_i$
$pmin = \underset{p_i=(x_i,y_i)\in S}{\operatorname{argmin}} \; x_i$
$L = L \cup \{pmin, pmax\}$
▷ Initialize each strip's extreme points ◁
foreach $i \in [1, 2, \cdots, k]$ **do**
    $pmin_{s_i} = (\infty, \infty), pmax_{s_i} = (-\infty, -\infty)$
foreach $p \in S$ **do**
    $i = \lfloor \frac{1}{k}(p.x - pmin.x)(pmax.x - pmin.x) \rfloor$
    $pmin_{s_i} \leftarrow \underset{v\in\{p,pmin_{s_i}\}}{\operatorname{argmin}} \; y_v$
    $pmax_{s_i} \leftarrow \underset{v\in\{p,pmax_{s_i}\}}{\operatorname{argmax}} \; y_v$
foreach $i \in [1, 2, \cdots, k]$ **do**
    $L \leftarrow L \cup \{pmin_{s_i}, pmax_{s_i}\}$
▷ Return the subset L ◁
return L

Underlying Convex Hull Definition. Union idea – union of convex combinations.

Hull Approximation Type. This algorithm is clearly an inner hull algorithm.

The authors also suggested an outer hull version as well as a "mid"-hull version. The outer hull version essentially replaces each extreme point $p = (x, y)$ with two new points $p_l = (x_l, y)$ and $p_u = (x_r, y)$, where x_l is the x-coordinate of the left boundary of the strip containing p and x_r that of the right boundary. This amounts to essentially adding the four corners of the minimum enclosing box for each strip to the subset.

The "mid"-hull is constructed by shifting each extreme point horizontally so that they lie in the center of their respective strip. This variant is noted to have a slightly smaller distance error ratio of $1/2k$ rather than $1/k$.

Analogous Sorting Algorithm. Clearly, this algorithm resembles bucket sort, since it splits the point into vertical strips, which is similar to the idea of buckets in bucket sort.

Generalization to d-space. Bentley et al. discussed a generalization of their algorithm to d-space. The key idea here is to generate $k + 2$ strips along each of the dimensions $1, 2, \cdots, d - 1$ and then find the extreme points in the d dimension to obtain L.

Input Space. The input space is $\mathbb{R}^2$, but can be generalized to support $\mathbb{R}^n$.

Complexity. The time and space complexity of Bentley et al.'s algorithm is $\mathcal{O}(n + k)$ and $\mathcal{O}(n)$ respectively.

When generalized to d-dimensional space, it runs in $\mathcal{O}(n + g(k, d))$ time and $\mathcal{O}\left(k^{d-1}\right)$ storage, where $g(k, d)$ is the time-complexity for computing the convex hull of k points in d-space. In 3-dimensional space, $g(k, 3) = \mathcal{O}\left(k^2 \log k\right)$. For $d > 3$, the best worst-case $g(n, k)$ known is due to Chazelle [16] and runs in $\mathcal{O}\left(n \log n + n^{\lfloor d/2 \rfloor}\right)$ time. Unlike in the planar case, there is no known algorithm that takes advantage of the existing ordering in L to compute the convex hull faster than $\mathcal{O}\left(n \log n + n^{\lfloor d/2 \rfloor}\right)$, which is optimal in the worst-case [16].

Accuracy Measures. Bentley et al. showed a relative distance error bound of $\mathcal{O}(1/k)$. The area error bound is $\Theta(1)$ [36], as shown in Figure 2.3.

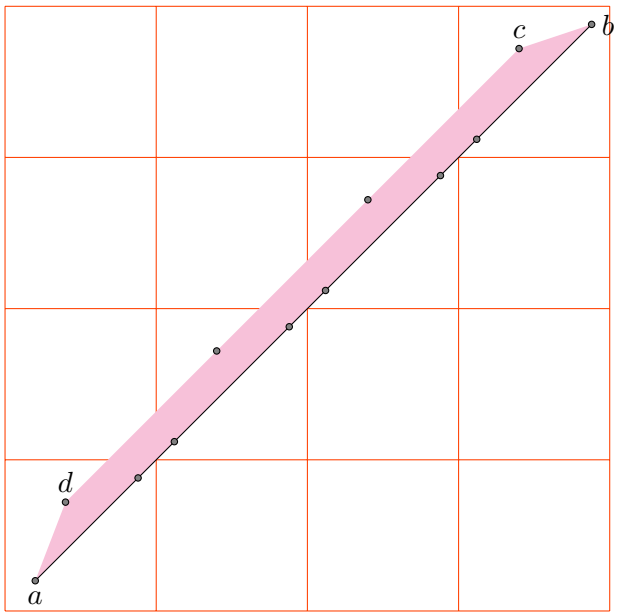

Figure 2.3: Area Approximation Error of Bentley et al.'s Algorithm

Note that the algorithm of Bentley et al. as well as that of Soisalon-Soininen's algorithm [52, 53] will both return the line segment ab as the approximate hull, whereas the exact hull is the trapezoid $abcd$, thus giving a worst-case relative area error of $\Theta(1)$

Parallelizability. This algorithm is easy to parallelize since the processing of each strip is independent and can thus be delegated to a different processor.

Streaming model. Bentley's algorithm as defined assumes that the whole point set is known at the start of the algorithm. However, we only need the point set in order to compute the x-range of the input point set. In the streaming model, this information

is not fully known until all the points have been seen. A two-pass streaming algorithm can be devised however, where the first pass computes the extrema of the point set, so that the strip width can be computed and the second pass computes the subset two per strip and finally the approximate hull is then computed from the subset. The parameter k can be understood to be the memory budget of the streaming algorithm.

Miscellaneous Issues. Kim and Stojmenovic [36] proposed two approximate hull algorithms — one of which is an extension of Bentley et al.'s. The only difference between this algorithm and that of Bentley et al. is that in computing the extreme points within each vertical strip, it takes the points that are farthest above or farthest below the line segment connecting the horizontal extreme points.

Soisalon-Soininen's algorithm [52, 53] improved slightly on Bentley et al.'s algorithm by splitting the point set both vertically into k_1 strips as well as horizontally into k_2 strips. The algorithm starts by finding the two extreme points along the horizontal axis x_{min} and x_{max}, and the two along the vertical axis y_{min} and y_{max}. Then it splits the point set into k_1 vertical strips and then into k_2 horizontal strips. Next, it computes the vertical subset L_1 as the set of extreme points within the vertical strips. It also computes L_2 as the extreme points along the horizontal strips. The intricate part of the algorithm is the merging of L_1 and L_2 to form the subset for the entire point set L in sorted order. This is achieved by first finding a common point between the L_1 and L_2 at the corners and then filtering out the points that are farther in to make it into the convex hull of the merged set. Finally, it computes $\mathrm{conv}(L)$ using an exact algorithm just as is done in Bentley et al's.

This algorithm runs in time $\mathcal{O}(n+k)$ where $k = \max(k_1, k_2)$ and uses $\mathcal{O}(k)$ space, not counting the input. The paper proved that the Hausdorff distance from the true hull to the approximate hull produced by this algorithm is no more than $\sqrt{2}/2k$ of the diameter of the input point set.

2.2.2.3 Žunić's Algorithm [55]

This algorithm is an extension of Jarvis march. It constructs both an outer and an inner hull approximation. It starts out by constructing an axis-parallel bounding box of the point set S. Denote the four corners of the bounding box by t_i where $i = 1, 2, 3, 4$. Thus, in the first iteration, the vertices of the approximate convex hull, V_1, consist of these four corners as well as any other points lying on the four sides of the bounding box and adjacent to the four corners. In other words, $V_1 = \{t_1, \cdots, t_4\} \cup \{l_1, \cdots, l_4, r_1, \cdots, r_4\}$, where l_i (r_i) is the point adjacent to vertex t_i on the left (right). Note that some of the l_i's might coincide with the r_i's.

The algorithm proceeds by successively replacing each t_i with three points, l_i', t_i', r_i', from within the triangle $l_i t_i r_i$. The point l_i' (r_i') is the point that maximizes the angle $\sphericalangle(l_i', l_i, r_i)$ ($\sphericalangle(r_i', r_i, l_i)$). The point t_i' is the point of intersection of the two line segments $l_i l_i'$ and $r_i r_i'$.

Underlying Convex Hull Definition. Intersection of half planes.

Hull Approximation Type. Since the approximate convex hull computed by the above algorithm contains points t_i, which are not necessarily points from S, it is clearly an outer hull approximation. In order to compute an inner hull, the algorithm simply takes the outer hull V_k and computes the inner hull as $\mathrm{conv}(V_k \backslash \{t_i\}), i \in [1, \cdots, 4]$.

Analogous Sorting Algorithm. Underlying sorting algorithm is selection sort.

Generalization to d-space. The generalization to arbitrary dimension follows from the generality of the gift-wrapping paradigm that informs the Jarvis march [14].

Input Space. The original algorithm was designed for grid points, so the input space is $\mathbb{Z}^2$, but can be generalized to support $\mathbb{R}^n$.

Complexity. This algorithm runs in $\mathcal{O}(kn)$ time and $\mathcal{O}(n)$ space, where k is the number of refinement iterations taken by the algorithm.

Accuracy Measures. The relative distance error and the area error are shown to be $\mathcal{O}(1)$ by Kim and Stojmerovic [36].

Parallelizability. The gift-wrapping paradigm is inherently sequential, as each iteration depends on the output of the previous one.

Streaming model. The algorithm assumes that the entire point set is available at the outset, so that its bounding box can be computed, and then its bounding 8-gon, and so on. In the streaming model such structures cannot be reliably computed until all the points have been seen.

2.2.2.4 Kim and Stojmenovic's Algorithms [36]

Kim and Stojmenovic [36] proposed two approximate hull algorithms — one of which is an extension of Bentley et al.'s. The only difference between this algorithm and that of Bentley et al. is that in computing the extreme points within each vertical strip, it takes the points that are farthest above and below from the line segment connecting the horizontal extreme points.

The second algorithm in the work of Kim and Stojmenovic [36] is an adaptation of quickhull [22, 10]. This algorithm is essentially the quickhull algorithm, but breaking out at the k-th iteration or recursive depth. This second algorithm is analyzed using our framework below.

Underlying Convex Hull Definition. Intersection of half-planes.

Hull Approximation Type. Inner hull.

Analogous Sorting Algorithm. Quick sort.

Generalization to d-space. Generalizable to d-space, since it is essentially quickhull, with fewer iterations or recursive depth.

Input Space. The input space is the real plane, $\mathbb{R}^2$.

Complexity. $\mathcal{O}(n \log k)$ worst-case time.

Accuracy Measures. Both the relative distance and area error bounds are $\mathcal{O}\left(\frac{1}{k^2}\right)$.

Parallelizability. Parallel version of quickhull [41] can be adapted to obtain a parallel version of this algorithm.

Streaming model. Algorithm assumes the entire point set is available from the outset.

2.2.2.5 Kavan et al.'s Algorithm [35]

The algorithm proposed by Kavan, Kolingerova, and Zara [35] works roughly as follows. Given a point set S and an integer k, it splits the point set into the k sectors of a circle with an arbitrary center $c \in S$. Each sector s_i has an angle of $2\pi/k$. The algorithm projects all the points in a sector onto a half-line l_i that originates from c and bisects s_i. The trivial extreme point p_i for a sector s_i is defined as follows:

$$p_i = \underset{p \in s_i \cap S}{\operatorname{argmax}}\, p \cdot \overrightarrow{l_i}, \tag{2.2.4}$$

where $\cdot$ is the dot-product and $\overrightarrow{l_i}$ is the unit vector pointing away from c and collinear with l_i. Unfortunately, when such extreme points are used to generate an approximate hull, the distance from a point outside the hull to the hull is unbounded. So, the authors compute another set of extreme points p_i^*, from the p_i's above as follows:

$$p_i^* = \underset{p \in P}{\operatorname{argmax}}\, p \cdot \overrightarrow{l_i}, \tag{2.2.5}$$

where $P = \bigcup_{j \in \{1,2,\cdots,k\}} p_j$. Next, for each of the extreme points p_i^*, a half-space h_i is defined with the normal l_i^* to l_i passing through p_i^*. The approximate hull is defined by the intersection of these half-spaces.

Underlying Convex Hull Definition. Intersection of half-spaces.

Hull Approximation Type. Note that the vertices of hulls thus constructed are not necessarily the extreme points. Rather, the extreme points would lie on the edges of the hull. Thus, these approximate hulls are neither inner nor outer hulls. On the one hand, they resemble outer hulls in that they result from an intersection of half-planes. On the other, they also resemble inner hulls as they are sometimes wholly contained within the exact hull.

Analogous Sorting Algorithm. Since points are divided into sectors just as keys are split into buckets, the analogous sorting algorithm is clearly bucket sort.

Generalization to d-space. Algorithm can be extended to arbitrary dimensions since most computations involve computing norms and partitioning the input space into sectors. In d-dimensional space, the surface of the smallest enclosing d-ball, $\mathbb{B}$ can be partitioned into k zones. The half-space that passes through an extremal point for a zone, defined similarly, that is orthogonal to the ray emanating from the center of $\mathbb{B}$ and passing though the zonal center is computed for each zone. The intersection of these half-spaces define an approximate convex hull in d-space.

Input Space. The input space is $\mathbb{R}^2$.

Complexity. Kavan et al's algorithm takes $\mathcal{O}\left(n + k^2\right)$ time and $\mathcal{O}\left(n + k\right)$ space.

Accuracy Measures. It was shown in [35] is that the distance δ to any point external to the hull produced by this algorithm is bounded by the inequality,

$$0 \leq \delta \leq \max(r \tan \frac{\pi}{k}, 2r \sin \frac{\pi}{k}), \tag{2.2.6}$$

where $r < \mathrm{diam}(S)$ is the distance between c and its farthest neighbor in S.

The relative area error for Kavan's algorithm is also $\Theta\left(1\right)$ when it underestimates. To see why, consider Figure 2.4 below.

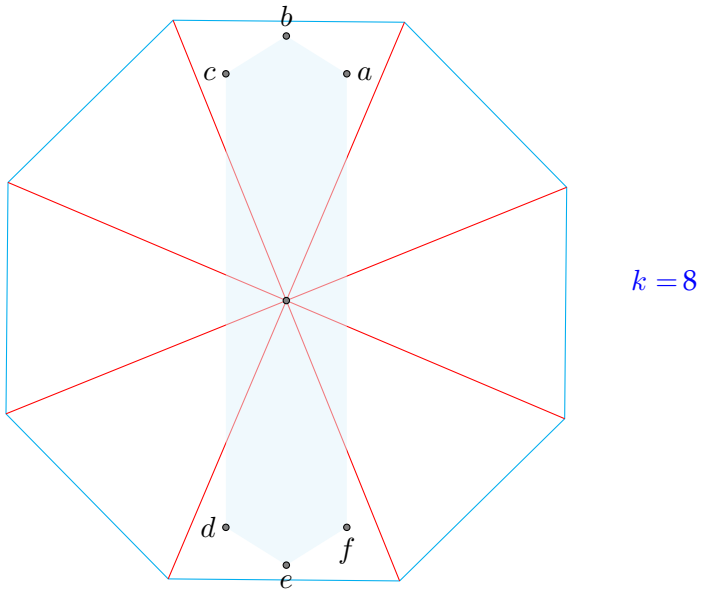

Figure 2.4: Area Approximation Error of Kavan's Algorithm – Underestimate

For the point set in Figure 2.4, Kavan's algorithm will return the segment be as the approximate hull, whereas the exact hull is the shaded polygonal area, $abcdef$. Here the algorithm grossly underestimates the true hull.

However, when it overestimates, Kavan's algorithm has an unbounded relative area error as shown in Figure 2.5. Here the true hull is the segment ac, and thus has zero area, but Kavan's algorithm will return a region, making the relative area error unbounded in this case.

Parallelizability. The computation within each sector can be handed off to a different processor. So, a parallel version is conceivable.

Streaming model. Another positive aspect of this algorithm is that it is also an online algorithm with an update time complexity of $\mathcal{O}(k)$. When implemented as an online algorithm, the space complexity can be reduced to $\mathcal{O}(k)$, since interior points can be discarded as soon as they are discovered.

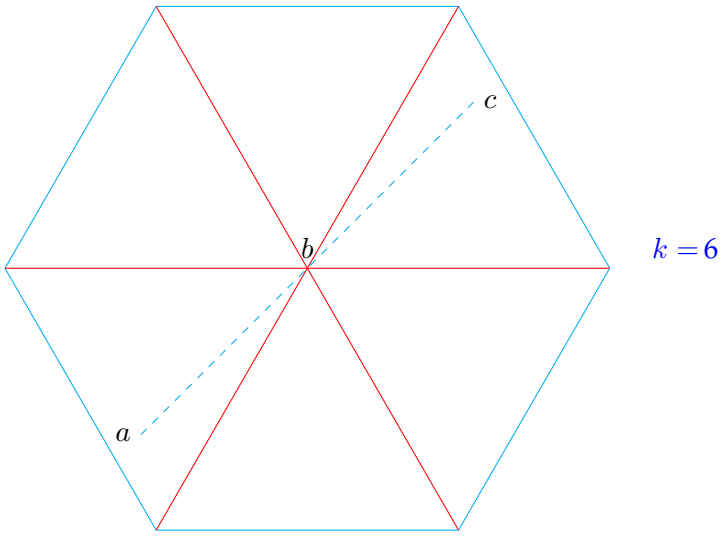

Figure 2.5: Area Approximation Error of Kavan's Algorithm – Overestimate

2.2.3 New Convex Hull Approximation Algorithms

This section discusses three new approximation algorithms for the convex hull problem. The first one is based on a bucketing technique – split the point set into sectors and then select candidate points from each sector. The convex hull of these candidate points then becomes the approximate convex hull. The second algorithm combines bucketing with ideas from the quickhull algorithm. The third is an enhancement of Kavan's algorithm.

2.2.3.1 Radial Bucketing

In combination with Algorithm APPROXSUBSET on page 7, Algorithm COMPUTESUBSET on page 23, presents the pseudocode for a new convex hull approximation algorithm. It essentially consists of two steps. Step 2.1 invokes COMPUTESUBSET to compute a subset and Step 2 computes and returns the convex hull of the subset. The algorithm depends on a control parameter k – the bigger it is, the better the approximation.

Underlying Convex Hull Definition. Union of convex combinations.

Hull Approximation Type. Inner hull approximation.

Algorithm 2.5: COMPUTESUBSET(S, k)

Input : A point set S and a parameter $k \geq 3$
Output: A subset of S

▷ Compute the centroid, c, of S ◁

1 $c \leftarrow \dfrac{\sum_{i=0}^{|S|-1} p_i}{|S|}$
2 Select a point $\hat{p} \in S$ farthest from c
3 **for** $i \leftarrow 0$ **to** $k-1$ **do**
4 $\quad L_i \leftarrow c$
5 **foreach** $p \in S$ **do**
$\quad$ ▷ Let ϕ_p be the polar angle of point p about c ◁
6 $\quad \phi_p \leftarrow \tan^{-1} \dfrac{\| \operatorname{proj}_y \overrightarrow{p-c} \|}{\| \operatorname{proj}_x \overrightarrow{p-c} \|}$
7 $\quad i \leftarrow \lfloor \dfrac{\phi_p k}{2\pi} \rfloor$
8 $\quad S_i \leftarrow S_i \cup \{p\}$
9 $\quad$ **if** $\|c-p\| > \|c-L_i\|$ **then**
10 $\quad\quad L_i \leftarrow p$
11 **for** $i \leftarrow 0$ **to** $k-1$ **do**
$\quad$ ▷ Let l_b (l_a) be the ray originating from L_i and passing through $L_{(i-1) \bmod k}$ (respectively through $L_{(i+1) \bmod k}$) ◁
12 $\quad$ add $\operatorname{argmax}_{p \in S_i} dist(p, l_b)$ to L if it exists
13 $\quad$ add L_i to L
14 $\quad$ add $\operatorname{argmax}_{p \in S_i} dist(p, l_a)$ to L if it exists
15 **return** L ▷ Return the subset L ◁

Analogous Sorting Algorithm. Bucket sort.

Generalization to d-space. Because of the bucketing step, this algorithm is not easily generalized to higher dimensions.

Input Space. The input points for this algorithm are arbitrary points from the plane, $\mathbb{R}^2$.

Complexity. Clearly, Algorithm COMPUTESUBSET runs in linear time, since all the iterations over the input point set either take $\mathcal{O}(n)$ or $\mathcal{O}(k)$ as indicated below.

Step 1 $\mathcal{O}(n)$

Step 2 $\mathcal{O}(n)$

Step 3 – 4 $\mathcal{O}(k)$

Step 5 – 10 $\mathcal{O}(n)$

Step 11 – 14 $\mathcal{O}(n)$

It is not hard to see that Algorithm APPROXSUBSET with this implementation of COMPUTESUBSET runs in $\mathcal{O}(n+k)$ worst case time. Step 1 of APPROXSUBSET, which simply invokes (COMPUTESUBSET), thus also runs in linear time, $\mathcal{O}(n)$. The second and last step 15 of APPROXSUBSET simply invokes an exact convex hull algorithm such as Graham's scan for a sorted list of points (polygonal vertices), such as L is. This only takes linear time $\Theta(k)$ [24]. Algorithm APPROXSUBSET then clearly runs in $\Theta(n+k)$ and is thus comparable to Bentley et al.'s in runtime.

Accuracy Measures. The relative distance error and the area error bounds for this algorithm are both $\mathcal{O}(1)$. This is achieved when all the points fall within a narrow rectangular band, such that only two sectors are populated.

Parallelizability. Each of the loops in COMPUTESUBSET can clearly be handed off to a different processor as each iteration is independent.

Streaming model. In a streaming model, we would not have the whole input point set in order to compute the centroid, so the algorithm must be adjusted to accommodate this fact. The resulting streaming algorithm is described in Section 3.3.

2.2.3.2 Quickhull + Bentley et al.'s Algorithm

This algorithm combines the preprocessing step of the quickhull algorithm with the idea of splitting the point set into k vertical strips from Bentley et al.'s algorithm. Given a point set P, the algorithm starts out by finding the two points p_l and p_r with the minimum

and maximum x-coordinates. Next, the algorithm finds the two points p_t and p_b respectively farthest above and below from the segment $p_l p_r$. Together these four points define a convex quadrilateral Q. Similarly to quickhull, it discards all the points in the interior of the quadrilateral Q. The algorithm then constructs a rectangle $abcd$ with sides parallel or perpendicular to the segment $p_l p_r$ with sides ab, bc cd, and da passing through the four extreme points p_t, p_r, p_b, and p_l respectively. Next, it computes the following four sets:

- $P_{lt}^{(1)}$ – points falling in the interior of the triangle $p_l a p_t$
- $P_{tr}^{(1)}$ – points falling in the interior of the triangle $p_l b p_t$
- $P_{br}^{(1)}$ – points falling in the interior of the triangle $p_l c p_t$
- $P_{lb}^{(1)}$ – points falling in the interior of the triangle $p_l d p_t$

Each of these triangles is then partitioned into k strips perpendicular to the side of the quadrilateral $p_l p_t p_r p_b$ that defines it. Next, the algorithm selects the point within each strip farthest from its side of the quadrilateral Q. It then constructs the list P' using these points as well as the corners Q in order. Finally, it computes the convex hull $\mathrm{conv}(P')$ using a linear time convex hull algorithm such as Graham's scan for polygonal points.

One possible way to enhance this algorithm's accuracy is to let the algorithm do a few more iterations of quickhull until the subproblem size reduces to a certain threshold value, that is a function of n and k, before applying the bucketing step. It is however not clear how to achieve this while still maintaining a linear runtime.

Underlying Convex Hull Definition. Quickhull and bucket sort.

Hull Approximation Type. Inner hull approximation.

Analogous Sorting Algorithm. The initial partition part of the algorithm is reminiscent of quick sort's PARTITION step. However, the subsequent step of splitting the point into slabs is more analogous to bucket sort.

Generalization to d-space. Because of the bucketing step involved, generalization to higher dimensions is not obvious.

Input Space. The input points for this algorithm are arbitrary points from the plane, $\mathbb{R}^2$.

Complexity. The first step in the algorithm finds the four points with maximum and minimum x- and y-coordinates. Together, these define a quadrilateral Q. This step takes linear time. Similarly, discarding points bounded by the quadrilateral Q takes linear time. Finally, splitting each group of remaining points into k buckets also takes linear time. So, does the call to Graham scan, since the points are already sorted. So, overall, the algorithm takes linear time, $\mathcal{O}(n+k)$ and linear space, $\mathcal{O}(n)$.

Accuracy Measures. Both the relative distance and area error bounds are $\mathcal{O}\left(\frac{1}{k^2}\right)$.

Parallelizability. This algorithm is parallelizable. Its initial steps can start off with a parallel version of quickhull. The bucketing steps that follow are also parallelizable as well.

Streaming model. It is possible to devise a streaming version, where the quadrilateral Q is constantly being updated as new points arrive. Points outside of Q are retained in memory, until the memory budget is reached. At that moment, the algorithm partitions them into slabs and finds the list P'.

2.2.3.3 Enhancement to Kavan's Algorithm

The main reason for the poor relative error of Kavan's algorithm is that it only computes one extremal half-plane per sector. By increasing the number extremal half-planes to include two additional neighboring directions on either side, three half-planes are produced for each sector. One is orthogonal to the sector's directional vector, and the other two are respectively orthogonal to the directional vectors of its adjacent sectors.

In other words, rather than compute,

$$p_i = \underset{p \in s_i}{\operatorname{argmax}}\, p \cdot \overrightarrow{l_i} \tag{2.2.7}$$

we compute instead

$$p_{i,j} = \underset{p \in s_i}{\operatorname{argmax}}\, p \cdot \overrightarrow{l}_{i+j}, \;\; j = -1, 0, 1 \tag{2.2.8}$$

and similarly,

$$p_i^* = \underset{p \in P}{\operatorname{argmax}}\, p \cdot \overrightarrow{l_i} \tag{2.2.9}$$

with P redefined as $P = \bigcup_{i \in \{1,2,\cdots,k\}, j \in \{-1,0,1\}} p_{i,j}$.

2.3 Conclusion

This chapter has presented an overview of the convex hull problem, and three new approximation algorithms for the convex hull. The chapter has also given complexity and error analyses for these algorithms. Further, future work will expand on their error analysis by use of empirical tools.

Future work will also attempt to unify the error analysis for the algorithms presented. The following questions will also be pursued further:

- Can an algorithm be devised that takes advantage of the ordering inherent in the subset S' of the input set S produced by Bentley et al.'s algorithms in higher dimension ($d > 2$) to compute a convex hull faster?
- Can inspiration be drawn from other sorting algorithms to find better exact or approximate convex hull algorithms?
- Can other definitions of the convex hull be used to devise better algorithms?

Chapter 3: Streaming Algorithm for the Convex Hull

This chapter introduces the problem of computing the convex hull from a stream of points arriving in arbitrary order. A streaming algorithm is an approximation algorithm constrained to work within a memory budget. Thus results that require more memory than the allowed budget must make decisions on what is worth keeping and what must be discarded. In Section 3.1, we introduce the problem, relate relevant literature in Section 3.2, present our contributions in Section 3.3, and Section 3.5 concludes the chapter.

3.1 Introduction

A streaming algorithm, typically limited in the amount of resources it is allowed, essentially has three parts: an initialization part, a processing part, and a query answering part.

Initialization. In this part, counters and data structures are initialized. This is the bootstrap for the algorithm and is executed only at the onset of the streaming process.

Process. This part computes an intermediate structure that can be easily updated with a new input as well as easily queried to obtain an answer based on the inputs seen so far.

Query. This part responds to queries using the latest state of the intermediate structure built in the process step above.

3.2 Related Work

In the case of a finite stream of points P, our algorithm behaves similarly to Preparata's exact online algorithm [46] when $k \geq |\operatorname{conv}(P)|$.

The streaming algorithm proposed by Hershberger and Suri [28, 30, 29] maintains extreme points in k uniformly spaced directions and another k extreme points in adaptively sampled directions. Their algorithm has a distance error of $\mathcal{O}\left(1/k^2\right)$. This distance is defined as the height of the tallest uncertainty triangle. The uncertainty triangle of an edge e_i is the triangle formed by extending its immediate neighbor edges e_{i-1} and e_{i+1} until they meet, assuming all such triangles are bounded. No area measure was reported.

Lopez and Reizner [40] proposed two algorithms for approximating an n-gon P by a k-gon Q. Their first algorithm builds an inscribed k-gon by repeatedly removing an ear of minimum area until only k vertices remain. So, it does bear some resemblance to our algorithm, however, it differs from our algorithm in at least two respects. Firstly, their algorithm is not online, as all the vertices of the n-gon are known ahead of time. So, the minimum area ear in their algorithm is truly globally minimum. In a streaming scenario, the minimum area ear is only minimum among the vertices remembered by the algorithm at an instant of time. Secondly, their algorithm does not and need not ensure that directional extrema are remembered.

Lopez and Reizner's second algorithm [40] similarly builds a circumscribing k-gon of minimum area to approximate an n-gon. At each iteration of the algorithm, a side of the polygon with minimum-area *outer cap* is chosen. The outer cap of a side s is the triangle formed by extending the neighboring sides until they meet. Their meeting point is then a new vertex of the polygon. So, each iteration eliminates a side, until there are only k sides left.

3.3 Contributions

3.3.1 Streaming Algorithm

Let $C = (p_1, p_2, ..., p_n)$ be a sequence of vertices of a convex polygon in counter-clockwise order. Each contiguous 3-sequence (p, q, r) in C defines a measure $\Delta_q = \text{GOODNESS}(p, q, r)$,

which is associated with the vertex q. We shall call the measure Δ_q the *goodness* of q. Note that Δ_q is a local measure and depends only on q and its two direct neighbors in C. Thus, whenever this contiguity relationship is violated, say by deletion of a direct neighbor or insertion of a new one, q's goodness must be recomputed. Similarly, when q is deleted, the GOODNESS of both p and r must be recomputed. By varying the definition of the function GOODNESS as the area, the perimeter of the triangle $\triangle pqr$, the length of the segment pr, the height of the triangle pqr relative to base pr, or even the angle $\angle q$ in $\triangle pqr$, we obtain different variants of the same algorithm. We shall mainly address ourselves to the area variant in this section.

3.3.1.1 INITIALIZE

The procedure INITIALIZE in Algorithm 3.1 initializes a height-balanced binary search tree T and a priority queue H to store the NODE references using two different keys. While points in T are ordered by their polar angles, points in H are keyed on their goodness value.

The structure T could be implemented as a left-leaning red-black tree [50, 51] and supports ordered sequence operations such as PRED, SUCC in addition to regular dictionary operations of INSERT, DELETEKEY and LOOKUP. It also supports the search operations of PRED and SUCC. Given an input key k, PRED (SUCC) returns the node with key immediately preceding (succeeding) k in T.

The priority queue H could be implemented as a binary min-heap and supports the heap operations of INSERT, DELETEMIN, and CHANGEKEY each in $\mathcal{O}(\log n)$ time [51, 18]. Each point is inserted into H with its goodness as key, thus the DELETEMIN operation on H will always return the vertex with the least goodness.

The structure L in Step 1 is a cyclic array and supports PRED and SUCC operations. The function NODE$(p, \Delta_p, \Theta_p$, deleted) creates a new node (a 4-tuple), whose attributes can be accessed using the attribute names POINT, GOODNESS, POLAR, and DELETED respectively.

Algorithm 3.1: INITIALIZE(S_k, k)

Input : The first k input points S_k and parameter k.
Output: T: height-balanced BST with vertices of $\mathrm{conv}(S_k)$ sorted by polar angles about centroid c, H: binary min-heap of vertices $conv(S_k)$ using GOODNESS as priority.

$L \leftarrow \mathrm{conv}(S_k)$
$c \leftarrow$ CENTROID(L)
$(N, W, S, E) \leftarrow$ DIRECTIONALEXTREMA(L, c)
foreach $p \in L$ **do**
    $\Theta_p \leftarrow$ POLAR(p, c)
    if $p \in (N, W, S, E)$ **then**
        $\Delta_p \leftarrow \infty$
    else
        $\Delta_p \leftarrow$ GOODNESS($L.$ PRED(p), p, $L.$ SUCC(p))
    node $\leftarrow$ NODE(p, Δ_p, Θ_p, false)
    $T.$ INSERT(Θ_p, node)
    $H.$ INSERT(Δ_p, node)
return (T, H, c, k)

3.3.1.2 PROCESS

Algorithm 3.2: PROCESS(T, H, c, k, p)

Input : T: height-balanced BST with $\leq k$ of $\mathrm{conv}(S)$ where S is the point stream, H: binary min-heap of $\leq k$ of $\mathrm{conv}(S)$, p: new point, k: memory budget
Output: T: a height-balanced BST update with p if on the hull, H: a binary min-heap updated with p if on the hull.

$n \leftarrow$ NODE($p, 0,$ POLAR(p, c), false)
$(T, H) \leftarrow$ UPDATEHULL(T, H, c, n)
if $|T| > k$ **then**
    $(T, H) \leftarrow$ SHRINKHULL(T, H)
return (T, H)

Procedure PROCESS is invoked each time a new point arrives. A new node n is created and used to update current hull by invoking procedure UPDATEHULL. The call to UPDATEHULL(T, H, c, n) in line 2 of Procedure PROCESS updates the structures T and H with a new node n. If the point associated with the new node, n.POINT, falls within the interior of the current convex hull or on its boundary, it is discarded. This test can be done in Steps 1 through 3 of UPDATEHULL.

Whenever the number of nodes in T exceeds k, the procedure SHRINKHULL is called to choose one vertex for eviction. This is done by calling the DELETEMIN() on the min-heap structure H to obtain the node q that should be evicted. The procedure then updates q's neighbor's GOODNESSes and deletes q from T.

3.3.1.3 QUERY

Algorithm QUERY is invoked to obtain the current hull at any point in the streaming process. It simply traverses T to return the hull vertices in a cyclic list.

The algorithm described is sensitive to the order in which points arrive in the stream. Consider the six points A, B, C, D, E, F shown in Figure 3.1 and Figure 3.2 below.

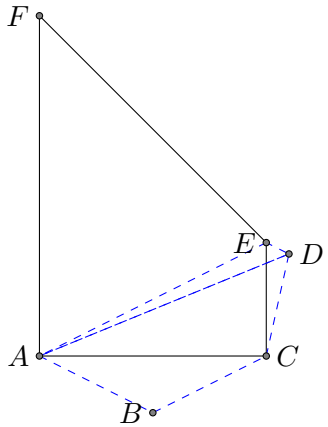

Figure 3.1: $k = 4$, arrival sequence: A, B, C, D, E, F. D is evicted after E arrives, and B after F.

Algorithm 3.3: UPDATEHULL(T, H, c, n)

Input : T: height-balanced BST with $\leq k$ of $\text{conv}(S)$, H: binary min-heap of $\leq k$ of $\text{conv}(S)$, n: new node.

Output: T: height-balanced BST updated with n if on the hull, H: binary min-heap updated with n if on the hull.

```
1  p ← T.FLOOR(n)
2  r ← T.CEILING(n)
3  if not CONTAINS(△prc, n) then
4      (s, t) ← TANGENTS(T, n)
5      x ← T.SUCC(s)
6      while x ≠ t do
7          x.deleted ← true
8          H.CHANGEKEY(x, −∞)
9          T.DELETEKEY(x.polar)
10         x ← T.SUCC(s)
11     q ← H.MINIMUM()
12     while q.deleted do
13         q ← H.DELETEMIN()
14     n.Δ_p ← GOODNESS(T.PRED(n), n, T.SUCC(n))
15     if n.Δ ≥ q then
16         T.INSERT(n.polar, n)
17         H.INSERT(n.Δ_p, n)
18         H.CHANGEKEY(s, GOODNESS(T.PRED(s), s, T.SUCC(s)))
19         H.CHANGEKEY(t, GOODNESS(T.PRED(t), t, T.SUCC(t)))
           ▷ Update extrema if needed                                  ◁
20         (N, W, S, E) ← UPDATEDIRECTIONALEXTREMA(T, c, n)
21         foreach n ∈ (N, W, S, E) do
               ▷ To prevent eviction of direction extrema              ◁
22             H.CHANGEKEY(n, ∞)
23 return (T, H)
```

3.3.2 Complexity Analysis

Theorem 3.1. *Procedure* INITIALIZE *runs in time* $\mathcal{O}(k \log k)$ *and uses* $\mathcal{O}(k)$ *space.*

Proof. Step 1 of Procedure INITIALIZE runs in time $\mathcal{O}(k \log k)$ using an optimal output sensitive planar convex hull algorithm [37, 13]. This step dominates the procedure. □

Algorithm 3.4: SHRINKHULL(T, H)

Input : T: height-balanced BST with $k+1$ vertices of $\text{conv}(S)$, H: binary min-heap of $k+1$ vertices of $\text{conv}(S)$.
Output: T: height-balanced BST with k vertices of $\text{conv}(S)$, H: binary min-heap of k vertices of $\text{conv}(S)$.

1 $q \leftarrow H.\,\text{DELETEMIN}()$
2 $p \leftarrow T.\,\text{PRED}(q.polar)$
3 $r \leftarrow T.\,\text{SUCC}(q.polar)$
4 $T.\,\text{DELETEKEY}(q.polar)$
5 $H.\,\text{CHANGEKEY}(p, \text{GOODNESS}(T.\,\text{PRED}(p), p, T.\,\text{SUCC}(p)))$
6 $H.\,\text{CHANGEKEY}(r, \text{GOODNESS}(T.\,\text{PRED}(r), r, T.\,\text{SUCC}(r)))$
7 **return** (T, H)

Algorithm 3.5: QUERY(T)

Input : T: height-balanced BST with k vertices of $\text{conv}(S)$
Output: A cyclic list of the vertices in T

1 **return** TOCYCLICLIST(T)

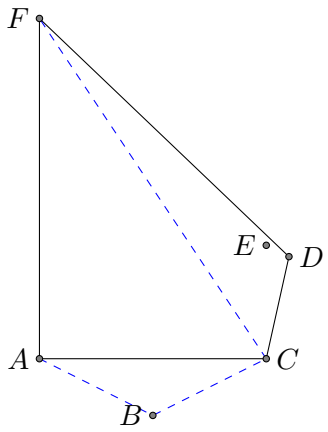

Figure 3.2: $k = 4$ with arrival sequence: A, B, C, D, F, E. B is evicted after F arrives. E is discarded as an interior point.

Lemma 3.1. *Procedure* UPDATEHULL *runs in time* $\mathcal{O}(\log k)$ *per point in the input stream* S.

Proof. Steps 1 through 2 of Procedure UPDATEHULL take $\mathcal{O}(\log k)$ time since they involve a binary search on T. Step 3 takes $\mathcal{O}(1)$ time. The call to TANGENTS takes $\mathcal{O}(\log k)$ time [46]. The rest of the procedure – Steps 5 – 13 – deletes a vertex chain that no longer belongs to the hull. Since these vertices are only deleted once per point in S, the total cost over all invocations of the procedure UPDATEHULL is $\mathcal{O}(n \log k)$, where n is the length of S. □

Lemma 3.2. *Procedure* SHRINKHULL *runs in time* $\mathcal{O}(\log(k))$.

Proof. Every step of Procedure SHRINKHULL takes $\mathcal{O}(\log(k))$. □

Theorem 3.2. *Procedure* PROCESS *runs in time* $\mathcal{O}(\log k)$ *time.*

Proof. Each invocation of PROCESS makes a single call to SHRINKHULL and at most a single call to SHRINKHULL. Thus, by Lemma 3.1 and Lemma 3.2, procedure PROCESS also runs in $\mathcal{O}(\log k)$ time. □

Theorem 3.3. *Procedure* QUERY *runs in time* $\mathcal{O}(k)$.

Proof. Procedure QUERY only does a depth-first (in-order) traversal of T to construct a cyclic list of its k vertices. □

Lemma 3.3. *Let* T_{i-1} *be the convex hull computed so far at the moment just before invoking Algorithm* UPDATEHULL. *Let* T_i *be resulting hull after* UPDATEHULL *returns. Then the following is invariant holds:*

$$|T_{i-1}| \leq |T_i| \tag{3.3.1}$$

Proof. Consider the invocation of UPDATEHULL on an arbitrary point p_i. The fate of p_i is one of two:

p_i **lies in the interior of** T_{i-1}**.** UPDATEHULL ignores p_i, in which case the hull does not grow and $T_i = T_{i-1}$

p_i **lies in the exterior of** T_{i-1}**.** UPDATEHULL expands T_{i-1} by adding p_i to the hull and therefore T_i has a bigger area than T_{i-1}.

□

A direct consequence of Lemma 3.3 above is the following statement.

Corollary 3.1. *The centroid can never become external to the hull interior, even after an invocation of procedure* SHRINKHULL.

Proof. This follows since the centroid is computed precisely once in Line 2 of INITIALIZE and never updated afterwards, but the directional extrema are recomputed with each input point if required in Algorithm UPDATEHULL. Note that the quadrilateral formed by these extrema will always contain the centroid, since it never gets smaller, since the extrema are protected from eviction as they have infinite GOODNESS. The only time an extreme point gets deleted from the hull is when a newly arrived point becomes more extreme than one of the extrema in one direction, as the extrema p is now being deleted in favor of q in Figure 3.3. This scenario, however, does not threaten the centroid c. □

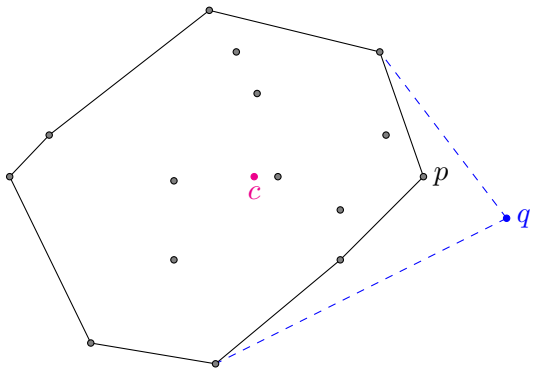

Figure 3.3: Convex Hull

Lemma 3.4. *When* $k \geq |\mathrm{conv}(S)|$ *the algorithm computes the exact convex hull of* S *.*

Proof. The algorithm then is equivalent to that of Preparata [46]. □

3.3.3 Error Analysis

Recall from Equation (2.2.3) that relative area error is defined as:

$$err_{\text{area}}(P, P') = \frac{|\text{area}(P) - \text{area}(P')|}{\text{area}(P)} \tag{3.3.2}$$

Lemma 3.5. *Each eviction from a convex $(k+1)$-gon by Algorithm* SHRINKHULL *introduces an error no worse than $\mathcal{O}\left(1/k^3\right)$.*

Proof. Let $m = k + 1$. Let Q be a convex m-gon and let $e_1, e_2, ..., e_m$ be its ears. Denote by $|e_i|$ the area of e_i. Let $Q'_i = Q - e_i$ denote the k-gon that would result if e_i were evicted. Therefore, the ratio $|e_i|/|Q|$ represents the area error that would result from deleting e_i. Further, let R_m denote a regular m-gon with unit area.

Renyi and Sulanke [49] proved the following result

$$\frac{1}{|Q|^m} \prod_{i=1}^{m} |e_i| \leq |r|^m \tag{3.3.3}$$

where r is an ear of R_m.

By taking logarithms and invoking the mean-value theorem, it is clear that there must exist at least one ear e_j in Q such that $\frac{|e_j|}{|Q|} \leq |r|$. The following inequality involving the ear r of a regular m-gon can be easily shown:

$$|r| = 4R^2 \frac{\pi^3}{m^3} \left[1 - \frac{\pi^2}{m^2} + \mathcal{O}\left(\frac{1}{m^4}\right) \right] \tag{3.3.4}$$

and thus:

$$\frac{|e_j|}{|Q|} < 4R^2 \frac{\pi^3}{m^3} \tag{3.3.5}$$

$$= \mathcal{O}\left(\frac{1}{(k+1)^3}\right). \tag{3.3.6}$$

□

Lemma 3.6. *Let $e_1, e_2, ..., e_m$ denote the sequence of ears evicted by the streaming algorithm. The following inequality holds*

$$|e_i| \leq |e_{i+1}| < H.\text{MINIMUM} \text{ for all } i = 1, 2, ..., m-1. \tag{3.3.7}$$

Proof. Recall that Algorithm UPDATEHULL only inserts a new node if its goodness is greater than $H.\text{MINIMUM}$. By definition, $H.\text{MINIMUM}$ increases with each eviction. So, just before the the i-th eviction, $H.\text{MINIMUM} = |e_i|$, but increases to $|e_{i+1}|$ right afterwards. □

We shall need a new term, *outer ear*, to make sense of the next lemma.

Definition 3.1. *Let P be a convex k-gon with sides $s_1, s_2, ..., s_k$ where each side $s_i = p_{i-1}p_i$. We associate to each side s_i of P, a triangle t_i defined by s_i and the extension of its neighboring sides s_{i-1} and s_{i+1} such that they meet on the side of s_i that is exterior to P. We call t_i a* finite outer ear *of P. Note that s_{i-1} and s_{i+1} may not meet on the side of s_i exterior to P, in which case call t_i an* infinite outer ear.

Lemma 3.7. *Let P be a convex k-gon returned by a call to Algorithm* QUERY. *Any vertex evicted in the course of the streaming process, must lie in the interior of P or in one of its outer ears.*

Proof. Suppose for the sake of contradiction that there was some vertex q that was evicted, but does not fall within P or any of its outer ears. This means q must lie within a wedge defined by two half-lines obtained by extending two successive sides of P, say s_i and s_{i+1}.

Note that the (inner) ear defined by q is now bigger than that of p_i, but only a minimum area ear could have been evicted by Lemma 3.6 – a contradiction. □

Lemma 3.8. *All evictions from within a finite outer ear o_i of a convex k-gon P must lie within an area no greater than $2H.\text{MINIMUM}()$.*

Proof. Let s_i be the side of P associated with the outer ear o_i. Suppose $a = H.\text{MINIMUM}()$. Since each one of these evicted ears must fit within o_i and have an area no greater than a. The possible range of all such ears is bounded by a trapezoid A with s_i as its base and a height h:

$$h \leq \frac{2a}{s_i} \tag{3.3.8}$$

Since the top of A is less than its base, otherwise it could not have been enclosed in the finite outer ear o_i. Thus, it fits within a parallelogram M of base s_i and height h. The area of M is at most $2a$, by Equation (3.3.8).

□

Lemma 3.9. *Let S be the stream of points processed in a streaming process. Let P be the convex k-gon created after processing S. The directional extrema of P, (N, W, S, E), maintained by Algorithm* UPDATEHULL *define an axis-parallel bounding box B that contains $\text{conv}(S)$.*

Proof. Note that these directional extrema are extreme over all of S in the four axis-parallel directions. Suppose there were some point p in S not contained in B. Further suppose, without loss of generality, that p lies above B, then p must be more extreme than N in the positive y direction – a contradiction. □

Lemma 3.10. *Let $s_i = p_i p_{i+1}$ be the side of P adjacent to an infinite ear of P. Then both p_i and p_{i+1} are extreme points.*

Proof. Suppose, without loss of generality, that p_i is not an extreme point and that it is closer to the W extreme point than to N. Then, since the chain $W, \cdots, p_{i_1}, p_i, p_{i+1}, \cdots, N$ is an xy-monotone chain, the outer ear associated with $s_i = p_i p_{i+1}$ is finite – a contradiction.

□

Lemma 3.11. *All evictions from within an infinite outer ear o_i of a convex k-gon P must lie within an area no greater than $2H.\,\text{MINIMUM}()$.*

Proof. Again, let $a = H.\,\text{MINIMUM}()$. Consider the set of all evictions that have taken place from the infinite outer ear o_i associated with a side s_i of P. Each one of these evictions has area less than a, since they could not have been evicted otherwise.

Let $s_i = p_i p_{i+1}$. By Lemma 3.10, both p_i and p_{i+1} are extreme points. Also, by Lemma 3.9, the bounding box B must contain these points and all points ever evicted from the o_i. Thus, the intersection of B and o_i define a triangle Δ that contains all ears evicted from o_i.

Similarly to Lemma 3.8, the possible range of these evicted ears is bounded by a trapezoid A with s_i as its base and a height h:

$$h \leq \frac{2a}{s_i} \tag{3.3.9}$$

Since the top of A is also smaller than its base, being contained in triangle Δ, the area of A is at most $2a$, by Equation (3.3.9).

Thus, the area of A is bounded above by $2a$. This completes the proof.

□

The following theorem gives an upper bound on the area error for processing $n \gg k$ points.

Theorem 3.4. *The total area error incurred in the streaming process is bounded above by $\mathcal{O}\left(1/k^2\right)$.*

Proof. We consider two cases.

Case 1. *Evictions from within a finite outer ear.*

By Lemma 3.8, the total area of all the evictions within one finite outer ear is bounded above by $2H.\,\text{MINIMUM}$.

Case 2. *Evictions from within an infinite outer ear.*

By Lemma 3.11, the total area of all the evictions within one infinite outer ear is bounded above by $2H.\,\text{MINIMUM}$.

By Lemma 3.5, $H.\,\text{MINIMUM}$ is at most $\mathcal{O}\left(1/k^3\right)$ and since there are k outer ears, the total error is $\mathcal{O}\left(1/k^2\right)$. This completes the proof. □

Note that in general not all evictions would have an impact on the final k-gon returned at the end, after processing all points in the stream. However, when an adversary could provide a stream of points that all lie on the convex hull, such as the vertices of a regular n-gon, the above error bound, being a worst-case bound, would still apply.

Theorem 3.5. *Given an adversarial input, the total area error accumulated by all the evictions is at least*

$$2\pi^2\left[\frac{1}{k^2}-\frac{1}{n^2}\right]. \tag{3.3.10}$$

Proof. This bound was obtained by [40], but in their case, they had access to all the vertices offline as discussed earlier in Section 3.2. □

3.3.4 Empirical Results

A stream S of ten thousand random points lying on a common circle is generated. We then feed thirty three random shuffles of S to the streaming algorithm and take the mean distance and area relative errors. These are then used to compute the lower and upper bounds as defined in Theorem 3.4 and Theorem 3.5. The empirical area error is neatly sandwiched between the two bounds as expected.

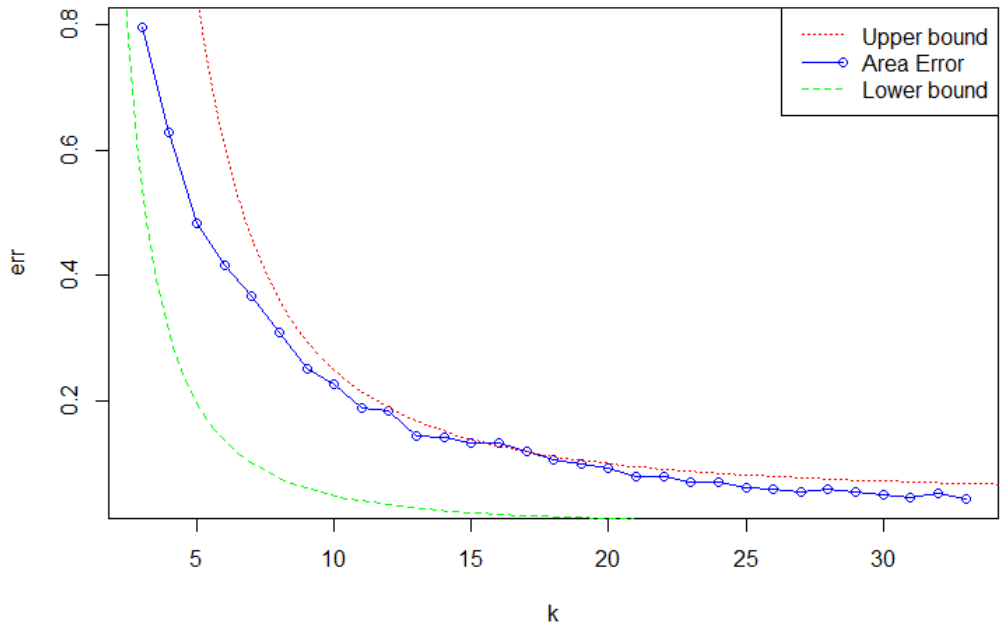

Figure 3.4: Empirical Area Error sandwiched between Lower and Upper Bound curves

For completion, Figure 3.5 also shows the distance and area relative errors.

3.4 Refinement

We consider a refinement of Algorithm 3.2 given below, which uses the idea from Lopez and Reisner [39]. The essential difference is that rather than invoke SHRINKHULL every time the k-gon grows into a $(k+1)$-gon, the Algorithm waits until the it grows into a mk-gon for some small constant m before invoking SHRINKHULL. This only works, of course, if the memory constraint allows use of $(m-1)k$ extra memory for processing. The main benefit of this enhancement is that the effect of order in the point sequence depicted earlier in Figure 3.1 and Figure 3.1 is minimized, while keeping the same overall asymptotic time bounds.

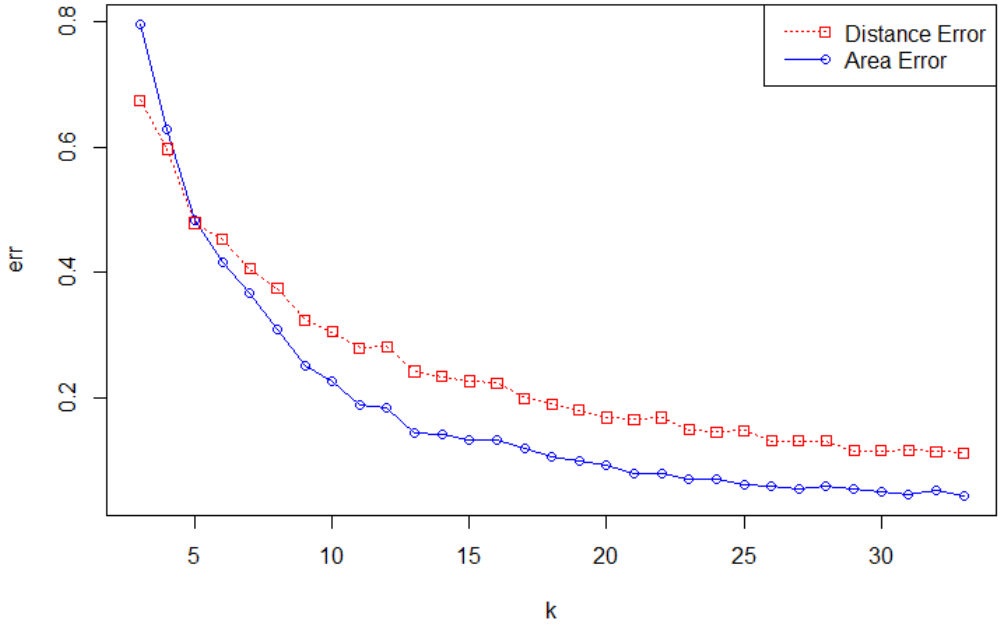

Figure 3.5: Distance and Area Relative Errors

Algorithm 3.6: PROCESS(T, H, c, k, p)

Input : T: height-balanced BST with $\leq k$ of $\text{conv}(S)$ where S is the point stream, H: binary min-heap of $\leq k$ of $\text{conv}(S)$, p: new point, k: memory budget

Output: T: a height-balanced BST update with p if on the hull, H: a binary min-heap updated with p if on the hull.

```
n ← NODE(p, 0, POLAR(p, c), false)
(T, H) ← UPDATEHULL(T, H, c, n)
if |T| > mk then
    while |T| > k do
        (T, H) ← SHRINKHULL(T, H)
return (T, H)
```

3.5 Conclusion

A new streaming algorithm for the convex hull is presented. Its runtime and error bounds are analyzed. The gap between the lower and the upper bound can be further explored in a future work. Hershberger and Suri [28, 30, 29] only provided a distance error bound for their algorithm. One line of future work will be to derive an area bound of their algorithm.

Chapter 4: Convex Layers

To date, published sequential algorithms for the convex layers problem that achieve optimal time and space complexities have tended to be involved. In this chapter, we give a simple $\mathcal{O}(n \log n)$-time and linear space algorithm for the problem. Our algorithm computes four quarter convex layers using a plane-sweep paradigm as a first step. The second step then merges these together in $\mathcal{O}(n \log n)$-time.

The *convex layers problem*, also known as the *onion peeling problem*, can be defined as follows: Given a set of points P in the plane, construct a set of non-intersecting convex polygons, such as would be constructed by iteratively constructing the convex hull of the points left after all points on all previously constructed convex polygons are deleted. This chapter briefly describes the convex layers problem (Section 4.1) and some of its applications (Section 4.3), relates relevant literature (Section 4.4), presents our contributions (Section 4.5), and concludes with a list of open problems (Section 4.6).

4.1 Introduction

One can compute the convex layers of a point set P by taking the convex hull of P to obtain its first layer L_1. These points are then discarded from P and the convex hull of the remaining points are taken to obtain the second layer L_2. Those are then discarded and we continue this process until we run out of points. So, in general a point p belongs to layer L_i, if it lies on the convex hull of the point set $P - \bigcup_{j=1}^{i-1}\{L_j\}$.

Definition 4.1. *The* convex layers, $\mathbb{L}(P) = \{L_1, L_2, \cdots, L_k\}$, *of a set P of $n \geq 3$ points is a partition of P into $k \leq \lceil n/3 \rceil$ disjoint subsets L_i, $i = 1, 2, \cdots, k$ called* layers, *such that each*

layer L_i is an ordered[1] *set of the hull vertices of the set $\bigcup_{j=i,\cdots,k}\{L_j\}$.*

Thus, the outermost layer L_1 coincides exactly with the convex hull of P, $\mathrm{conv}(P)$. Next, we define the *convex layers problem*.

Definition 4.2. *Given a point set P, the* convex layers problem *is to compute $\mathbb{L}(P)$.*

A related concept is the notion of the *depth* of a point in a point set.

Definition 4.3. *The* depth *of a point p in a set P is the index $i \in [1 \cdots k]$, such that $p \in L_i$. The depth of P is $k = |\mathbb{L}(P)|$[48].*

It would appear that the convex layers problem can be defined as the the problem of computing the depths of all the points in P. This is not quite right, however, as each convex layer must be a (counter-clockwise) ordered sequence of the points that have the same depth. We now define the depth problem below.

Definition 4.4. *Given a point set P, the* depth problem *is to compute the mapping $D : P \to \{1, 2, \cdots k\}$ that assigns a depth $D(p)$ to each point p in P such that all the points having a common depth i also belong the same layer L_i.*

It is not hard to see that this problem can be solved easily for a point set P once we have its convex layers $\mathbb{L}(P)$.

4.2 Layering Problems

The convex layers problem belongs to a class of problems called *layering problems*. Each problem in this class uses an appropriate notion of depth to partition a set of objects into subsets, called layers, such that objects in the same layer have a common depth. Examples of more problems from this class are given below.

[1] One convention is to have the points sorted in the counterclockwise order starting with the one with the smallest x coordinate, breaking ties by choosing the one with the smallest y.

Upper envelope layers problem. The geometric objects to be partitioned are line segments. The upper envelope of a set of n line segments are exactly the set of segments that are visible (even if partially so) from a position above the line segments. The upper envelope layers can be obtained by repeatedly computing and discarding the upper envelopes until no segments remain. The upper envelope can be computed in $O(n \log n)$ time [26]. Thus, the lower bound for the envelope layers problem is $O(n \log n)$. Hershberger [27] obtained an optimal $O(n \log n)$ algorithm for the upper envelope layers when the segments are disjoint and an $O(n\alpha(n) \log^2 n)$ for the case when the segments intersect, where $\alpha(n)$ is the inverse Ackermann function. It is still open whether a faster algorithm can be found for the latter.

Layers of maxima problem. The maximal elements of a point set are the set of points that are not dominated by any other point. Given a point set $P \subseteq \mathbb{R}^d$, a point $p \in P$ is maximal if there exists no point $q \in P$ such that $q_i > p_i$ for all $i = 1, \cdots, d$, where p_i (q_i) is the i-th coordinate of p (q). The layers of maxima problem is to compute the maximal points, assigning them to layer 1, deleting them and then repeating the process until no points remain. This is well-studied problem and there are many published algorithms that are optimal [34, 9, 6]. The general approach is to maintain a dynamic data structure from which layers are extracted until elements run out. Nielsen uses a grouping trick to obtain an output-sensitive algorithm for computing the first k maximal layers in time $O(n \log H_k)$, where H_k is the number of points appearing in the first k layers. It is also not hard to imagine a plane sweep algorithm for this problem.

Multi-list Layering problem. Given k lists, $l_1, l_2, \cdots, l_k$, where each l_i contains a list of integers, it is required to assign each distinct integer in the set $\cup_i l_i$ to a layer as follows. During iteration i of the algorithm, the integer at the top of each list is extracted and assigned to layer i. Integers in layer i are then deleted from each of the lists. This is

repeated until all the lists are empty. This is an easier problem. It can be transformed into a matrix transpose problem with adequate preprocessing to ensure distinct elements and padding shorter lists with suitable sentinels.

4.3 Applications of Convex Layers

Convex layers have several applications in various domains, including robust statistics, computational geometry, and pattern recognition. The following list is not meant to be exhaustive.

Robust estimation [15]. In statistics, finding an estimator that is not sensitive to slight deviations from an assumed distribution is known as robust estimation. A good example is the α-trimmed mean. Consider a set S of size n. Let S' be the subset of S where the smallest and the biggest α fractions of the data have been taken out. The α-trimmed mean is the mean of S'. When this is generalized to 2 or higher dimensions, Tukey and others have suggested peeling off the convex layers until only $(1 - 2\alpha)n$ of the points remain. The mean of these is then taken [48].

Another application of convex layers to robust estimation mentioned by Green and Silverman [25] is the multivariate analog of rank-based statistics [31] called *depths* [48]. The depth of a point is the index of the convex layer it belongs to. For instance, points on the outermost convex layer have depth 1, points on the next layer depth 2, and so on.

Half-plane range search problem [15]. The *half-plane range search problem* can be stated as follows: Given a point set and a query half-plane, report all the points lying within that half-plane. Chazelle et al. [17] were able to derive an optimal solution to this problem using convex layers.

Pattern Recognition [54]. Suk and Flusser [54] described a technique for matching two

images by first mapping each image to a point set. Then, the convex layers of each point set are computed. Finally, the convex layers are compared using a matching function. Suk and Flusser reported that their technique works even when the images are taken at different camera angles. They found that when no points are occluded in both images, the algorithm can match the images in time $O(n \log n)$ time, whereas the best known algorithm for the general point set recognition problem under arbitrary deformation (including occlusion) is $O(n^5)$.

4.4 Related Work

A brute-force solution to the convex layers problem is obvious – construct each layer L_i as the convex hull of the set $P - \bigcup_{j<i} L_j$ using some suitable convex hull algorithm. The brute-force algorithm will take $O(kn \log n)$ time where k is the number of the layers. It essentially computes one convex layer at a time by peeling off the points on that layer. Abstractly, one can think of this algorithm as peeling off layer vertices one layer at a time from some geometric structure such as a point set, a Delaunay triangulation, or a Voronoi diagram. This *peeling* approach is reminiscent of many convex layers algorithms. Another general approach to this problem is the *plane-sweep* paradigm. We shall review algorithms in both categories below.

4.4.1 Peeling-based Techniques

One of the earliest works that takes this approach is Green and Silverman [25]. Their algorithm is a repeated invocation of quickhull to extract the convex layers, one layer per invocation. This algorithm runs in $O(n^2)$ worst-case time.

Overmars and van Leeuwen [45] proposed an algorithm for this problem that runs in $O(n \log^2 n)$ based on a fully dynamic data structure for maintaining a convex hull under arbitrary deletions and insertion of points. Each of these update operations takes $O(\log^2 n)$

time, since constructing the convex layers can be reduced to inserting all the points into the data structure in time $O(n \log^2 n)$, marking points on the current convex hull and deleting them off and then repeating this for the next layer. Since each point is marked exactly once and deleted exactly once in the life of the algorithm, these steps together take no more than $O(n \log^2 n)$ time. Thus, the whole algorithm runs in $O(n \log^2 n)$.

Chazelle [15] proposed an optimal algorithm for this problem that runs in $O(n \log n)$ time and $O(n)$ space, both of which are optimal.

A new algorithm that belongs to this class is discussed in Section 4.5.

4.4.2 Plane-Sweep Technique

The first algorithm on record that uses this technique is a modification of Jarvis march proposed by Shamos [48]. The algorithm works by doing a radial sweep, changing the pivot along the way, just as Jarvis march does, but does not stop after processing all the points. It proceeds with another round of Jarvis march that excludes points found to belong to the convex hull on the last iteration. This way, the algorithm runs in $O(n^2)$.

A natural thought process would lead one to wonder if Chan's modification of Jarvis march [11] that uses a grouping trick can help find an optimal solution to this problem. This is exactly the approach taken by Nielsen [42] to obtain yet another optimal algorithm for the convex layers problem. Nielsen's algorithm is output-sensitive in that it can be parametrized by the number of layers k to compute. It runs in $O(n \log H_k)$ time where H_k is the number of points appearing on the first k layers.

4.4.3 Other results

Dalal [19] showed that the expected number of convex layers for a set of n points uniformly and identically distributed within a smooth region such as a circle is $\Theta(n^{2/3})$. *For a polygonal region, however, the expectation is* $\Theta(\frac{n}{\log n})$.

The envelope layers problem [27] and the multi-list layering problem [20] have been shown to be P-complete. It is still not known whether the convex layers problem belongs to the class NC [2, 27]. Dessmark et al. [20] reported a reduction of the convex layers problem to the multi-list layering problem, but this reduction does not bring us any closer to resolving the status of the convex layers problem.

4.5 Contributions

The algorithm builds four sets of convex layers. Each set differs from the others by the direction of curvature of the convex layers. Each set is maintained in an augmented balanced binary search tree T. The set of points must be known ahead of time, so that each point's horizontal ranking (by x-coordinate) can be precomputed. This ranking is determined by sorting the points using their x-coordinates, breaking ties by their y-coordinates. The particular order (ascending or descending) depends on the particular set. Below, we give the order used by each set.

1. North-West: $p_i \preccurlyeq_{NW} p_j$ if $x(p_i) \geq x(p_j)$ and $y(p_i) \geq y(p_j)$.
2. North-East: $p_i \preccurlyeq_{NE} p_j$ if $x(p_i) \leq x(p_j)$ and $y(p_i) \geq y(p_j)$.
3. South-West: $p_i \preccurlyeq_{SW} p_j$ if $x(p_i) \geq x(p_j)$ and $y(p_i) \leq y(p_j)$.
4. South-East: $p_i \preccurlyeq_{SE} p_j$ if $x(p_i) \leq x(p_j)$ and $y(p_i) \leq y(p_j)$.

Each of these relations $\preccurlyeq_*$ is a precedence relation on the vertices of the relevant partial hulls. For instance, the relation $\preccurlyeq_{NW}$ can be used to place a set of points in a monotone sequence as defined below. For the rest of this chapter, we shall only restrict ourselves to the $\preccurlyeq_{NW}$ relation, as the other relations can be realized by rotating the plane to the North-West orientation, applying the relation and then rotating back.

Definition 4.5. *A polygonal chain $C = (p_1, p_2, \cdots, p_n)$ is* monotone *if it satisfies the inequality:*

$$p_i \preccurlyeq_{NW} p_j \text{ whenever } i < j \text{ for all } i, j \in \{1, 2, \cdots, n\}$$

Lemma 4.1. *Suppose L and R are two monotone convex chains that lie on opposite sides of some vertical line. Let p denote the rightmost point on the chain L, and q the rightmost point on R. The bridge between the two chains is monotone if and only if $p.y \leq q.y$.*

Proof. Follows trivially. □

4.5.1 Hull Tree Data Structure

A hull tree T is either **nil** or has a *node*. A hull tree node consists of a left child hull tree T_l, a right child tree T_r, and the following additional fields:

Table 4.1: Fields of a Hull Tree Node

Hull chain, $T.hull$	A linked list of vertices flanked by two virtual sentinels, $T.hull[0]$ on the left and $T.hull[-1]$ on the right.
Left cursor, $T.b$	A cursor that scans the vertices in $T.hull$ from the left.
Right cursor, $T.c$	A cursor that scans the vertices in $T.hull$ from the right.

The hull chain linked list structure supports the following operations:

Definition 4.6. *A* bridge *between two hull trees T_l and T_r is a line segment $b_l b_r$ such that b_l is a vertex in the hull chain $T_l.hull$ and b_r is a vertex in $T_r.hull$ and the line passing though $b_l b_r$ is a tangent to both chains.*

The operations supported by the hull tree data structure are given in Table 4.3. Every one of these operations will maintain the data structure invariants given in Table 4.4 for every hull tree T.

Table 4.2: Operations supported by the Hull Chain Structure

EXTRACT(i, j)	Extracts the sub-chain with index i through j inclusively.
LISTTANGENT(p)	Returns the pair of indices (pointers) of the two tangent vertices to p in $T.hull$, provided p is not dominated by $T.hull$.
LISTINSERTAFTER(q, p)	Inserts the vertex q as a successor of vertex p.
LISTINSERTBEFORE(p, q)	Inserts the vertex p as a predecessor to vertex q.
LISTDELETE(p)	Deletes the vertex p from the list.
NEWLIST(p)	Creates a new linked list data structure and adds p to it.

To refer to these invariants, we shall use the notation $T.inv(i)$ to mean the instance T of the hull tree data structure satisfies Invariant I_i. When two or more invariants are satisfied, we shall simply list the indices of the invariants, for instance $T.inv(3, 4, 5)$ would mean that Invariants 3, 4, and 5 hold. When we mean that all the invariants are satisfied, we shall simply write $T.inv()$, rather than the rather unwieldy $T.inv(1, 2, 3, 4, 5, 6)$.

Lemma 4.2. *The space complexity of a hull-tree T that stores a set P of n points is $\Theta(n)$.*

Proof. We only need show that the following two quantities are linear in n:

1. The number of nodes in a hull tree.

2. The sum of the lengths of all the hull chains in T.

Since HEIGHT$(T) = \Theta(\log n)$ by Invariant I_5, it has no more than $2n - 1$ nodes. Each node has two subtree pointers T_l, T_r and two cursor pointers $T.b$ and $T.c$, which together sum up to a constant.

Each node also has a hull chain with a size that ranges from 0 to n. Fortunately, the sum of the sizes of hull chains over the entire tree is n, by Invariant I_2 of the hull tree data structure. This completes the proof. □

Table 4.3: Operations supported by the Hull Tree Data Structure

BUILDTREE(P)	Takes a list of points P and returns a hull tree T containing them.
INSERT(C, T)	Takes a monotone convex chain of points, updates the current hull tree with them and returns the updated hull tree.
PUSHDOWN(C, T)	Takes a monotone convex chain of points C, splits them into a left part C_l and a right part C_r. It then inserts C_l into the left subtree $T.l$ and C_r into the right subtree $T.r$.
EXTRACTHULL(T)	Makes a copy h of the convex chain $T.hull$ at the root node, removes h from T, allowing new vertices from the subtrees to bubble up and take its place as though the vertices of h were never inserted into T. EXTRACTHULL(T) returns h.
DELETE(C, T)	Deletes the monotone convex chain, C, from the hull chain $T.hull$, adjusts and returns T as though the vertices in C were never inserted into it.
GETEXTREMES(T_l, T_r, a_l, a_r)	Returns a vertex pair p, q such that $$p = \underset{p_i \in T.l.hull}{\operatorname{argmax}} \; p_i \cdot l$$ $$q = \underset{q_i \in T.r.hull}{\operatorname{argmax}} \; q_i \cdot l$$ i.e. p and q are the maxima along l, where l is a line orthogonal to $a_l a_r$.
GETBRIDGE(T_l, T_r)	Returns a vertex pair p, q such that p comes from $T.l.hull$ and q from $T.r.hull$ and the chain $T.l.hull[0:p] \cdot T.r.hull[q:-1]$ is monotone, where $\cdot$ denotes concatenation.
TANGENTS(a_l, a_r, T_l, T_r)	Given two points a_l and a_r such that $a_l.x < a_r.x$, returns a pair of vertices, one of which is the tangent to a_l and the other tangent with T_l going through a_r. This will become clearer shortly when we describe its use in Algorithm DELETE.

4.5.2 Tree Construction

The algorithm for building a new hull tree is the BUILDTREE routine. We shall come back to discuss after first looking into its main building block, the INSERT algorithm.

Table 4.4: Invariants for the Hull Tree Data Structure

I_1: Monotonicity	Each hull chain is monotone.
I_2: Non-Redundancy	No vertex appears more than once within a hull tree.
I_3: Dominance	The hull chain for any node in a hull tree dominates those of its subtrees.
I_4: Cursor Existence	$T.hull$ contains $T.b$ and $T.c$.
I_5: Logarithmic Height.	$\text{HEIGHT}(T) \leq \lceil \log_2(\lvert T \rvert) \rceil$.
I_6: Non-Crossing Cursors	$x(T.b) \leq x(T.c)$.

4.5.2.1 INSERT

Algorithm INSERT is a recursive algorithm. It takes as input a convex chain C of vertices and a hull tree T.

The preconditions for invoking the algorithm are given in Table 4.5.

Table 4.5: Preconditions for INSERT

Precondition 1:	$\lvert C \rvert > 0$
Precondition 2:	C is a monotone chain
Precondition 3:	$\text{TAIL}(C).y > \text{TAIL}(T.hull).y$.
Precondition 4:	$T.inv()$

The postconditions for INSERT are given in Table 4.6.

Lemma 4.3. *Given a monotone convex chain C and a hull tree T satisfying all the preconditions of Algorithm* INSERT*, Algorithm* INSERT *correctly inserts C into T.*

Proof. The algorithm breaks into two cases:

Case 3. *T is an empty tree.*

This is the base case of the recursion – all it does is to create a new node and inserts the chain C into it. T then trivially satisfies the postconditions of INSERT.

Algorithm 4.1: INSERT(C, T)

Input : C, a convex chain of points to be inserted into T,
T, a hull tree built from some point set P not including the vertices of C.
Output: T, a hull tree built from $P \cup C$.

```
1  if T = nil then
2  |  T = NODE()
3  |  T.hull = C
4  |  T.b = T.c = HEAD(T.hull)
5  else
6  |  k = T.hull. LISTTANGENT(HEAD(C))
7  |  C' = T.hull. EXTRACT(k + 1, |T.hull|)
8  |  T.hull. LISTINSERTAFTER(C, TAIL(T.hull))
9  |  T.b = HEAD(T.hull)
10 |  T.c = TAIL(T.hull)
11 |  i = 1
12 |  while C'[i] belongs in the left subtree do
13 |  |  i = i + 1
14 |  C_l, C_r = SPLIT(C', i)
15 |  T.l = INSERT(C_l, T.l)
16 |  T.r = INSERT(C_r, T.r)
17 return T
```

Table 4.6: Postconditions for INSERT

Postcondition 1:	$T.inv()$
Postcondition 2:	$T.hull$ ends with C
Postcondition 3:	$T =$ BUILDTREE$(P \cup C)$, see Section 4.5.2.2 for a definition of BUILDTREE

Case 4. *T is not empty.*

Step 6 computes the tangent scanning backwards using the right cursor $T.c$ until it finds the tangent to C. Note that during scanning, if one cursor catches up with the other, they will both move together in order to preserve Invariant I_6 (Non-Crossing Cursors).

This scan is guaranteed to find the tangent by Precondition 3, if one exists and the

returned k will point to the the tangent. However, if it does not exist, a reference to the sentinel is returned. In either case, the chain C' that has been scanned past, not including the tangent, is extracted from $T.hull$. Chain C is inserted in its place. The new hull chain preserves the monotonicity invariant I_1 and satisfies Postcondition 2.

However, Invariant I_4 may be violated since the cursors may no longer be pointing to an existing member of $T.hull$, but these are immediately restored by steps 9-10. Moreover, Postcondition 3 is still not being met. The rest of the algorithm (Steps 11 - 16) recursively restores the postcondition. This completes the proof.

□

4.5.2.2 BUILDTREE

Given a point set P, Algorithm BUILDTREE starts by sorting these points by their x-coordinate values. The zero-based index of a point p in such a sorted list is called its *rank*, denoted $\text{RANK}_P(p)$. A point's rank is used to guide its descent down the hull tree during insertion.

Algorithm 4.2: BUILDTREE(P)

Input : P, a set of points, $\{p_i \mid i = 1, \cdots, n\}$.
Output: T, a hull tree built from P.

```
1 Compute the rank of each point by x-coordinate
2 Insert each point into a min-heap H keyed on the y-coordinate
3 T = NEWHULLTREE()
4 while |H| > 0 do
5 |   p ← H.EXTRACTMIN()
6 |   INSERT(p, T)
7 return T
```

The same set of points are then inserted into a min-heap structure H, but this time the key field (or priority value) in H will be their y-coordinate value. The points are

then extracted from H in increasing order of their y-coordinate values and inserted into T. The INSERT procedure expects a hull chain as the first parameter, so the call to INSERT in BUILDTREE is a understood to be a chain of one vertex.

Once all the points have been inserted, the hull tree is returned.

Lemma 4.4. *Right after a point p is inserted into a hull tree T, the relation* TAIL$(T.hull) = p$ *holds.*

Proof. Since points are inserted into T in the order of their priority (i.e. y-coordinate value) in the min-heap H, the most recently inserted point must have the largest y coordinate value of all the points inserted so far. Thus, the statement follows. □

Lemma 4.5. *Algorithm* BUILDTREE *constructs a hull tree of a set of n points in $\mathcal{O}(n \log n)$ time.*

Proof. Note that Step 1 through 3 take linear time. The While loop is executed n times. Since EXTRACTMIN from a binary heap containing n elements costs $\Theta(\log n)$ time, it remains only to show that all the invocations of INSERT by Algorithm BUILDTREE take no more than $\mathcal{O}(n \log n)$ time overall.

Consider an arbitrary point p inserted into T by BUILDTREE into T. Initially, it goes into the $T.hull$ by Lemma 4.4. In subsequent iterations, the point either stays within its current hull chain or descends one level down owing to an eviction from its current hull chain. The cost of descending a level of a chain C is dominated by the right-to-left tangent scan. Since only points that will descend will be examined in the scan, the cost of Step 6 is simply $\mathcal{O}(|C|)$. This is equivalent to saying that the amortized cost of descending a level by p is $\mathcal{O}(1)$.

Since there are only $\mathcal{O}(\log n)$ levels in T, the cost of processing p reduces to $\mathcal{O}(\log n)$. This competes the proof. □

Lemma 4.6. *Algorithm* INSERT *completes in $\mathcal{O}(\log n)$ amortized time.*

Proof. By Lemma 4.5, the cost of all invocations of INSERT by Algorithm BUILDTREE is $\mathcal{O}(n \log n)$, which amortizes to $\mathcal{O}(\log n)$ per point. □

4.5.3 Hull Peeling

We begin the discussion of hull peeling by examining Algorithm EXTRACTHULL, which takes a valid hull tree T and extracts the root hull chain h from it and returns it. We can state this in the form of preconditions for EXTRACTHULL, given in Table 4.7.

Table 4.7: Preconditions for EXTRACTHULL

Precondition 1:	$\|T\| > 0$
Precondition 2:	$T.inv()$

Algorithm 4.3: EXTRACTHULL(T)

Input : P, a set of points, $\{p_i \mid i = 1, \cdots, n\}$.
Output: T, a hull tree built from P.

```
1 h = T.hull
2 DELETE(h, T)
3 return h
```

When EXTRACTHULL completes, the new state of T, which we shall denote as T', is as though the points on the extracted chain h had never been inserted into T. We state these postconditions in Table 4.8.

Table 4.8: Postconditions for EXTRACTHULL

Postcondition 1:	$T' =$ BUILDTREE$(P \setminus h)$, where P is the set of points in T and $h = T.hull$.
Postcondition 2:	$T'.inv()$

The correctness and cost of Algorithm EXTRACTHULL obviously depend heavily on those of DELETE. We shall state the following theorem without proof in this section and return to it in Section 4.5.3.4.

Theorem 4.1. *Given a valid hull tree T containing n vertices and a valid hull chain C of k vertices, Algorithm* DELETE *correctly deletes C from T in $\mathcal{O}(k \log n)$ amortized time.*

Algorithm DELETE itself also depends on two other procedures GETBRIDGE and TANGENTS, so let us have look at those first.

4.5.3.1 GETEXTREMES

Table 4.9: Preconditions for GETEXTREMES

Precondition 1:	$T_l.inv()$
Precondition 2:	$T_r.inv()$
Precondition 3:	$T_l \neq T_r()$
Precondition 4:	$\text{TAIL}(T_l.hull).x \leq \text{HEAD}(T_r.hull).x$

Lemma 4.7. *Given two valid hull trees T_l and T_r satisfying the preconditions of* GETEXTREMES *and the roof vertices a_l and a_r, Algorithm* GETEXTREMES *correctly computes the the extreme points closest in perpendicular distance to the segment $a_l a_r$ in linear time.*

Proof. The scan for the left extreme point in T_l closest in perpendicular distance to the segment $a_l a_r$ is done using T_l's right-to-left cursor $T_l.c$. The scan for the right extreme point in T_r closest in perpendicular distance to the segment $a_l a_r$, however, is done using T_r's left-to-right cursor $T_r.b$. On completion, the two cursors will be pointing to the extreme points, as required by GETEXTREMES's Postconditions 3 and 4. Since, no other state changes were made to the two trees, the hull tree invariants continue to hold.

Algorithm 4.4: GETEXTREMES(T_l, T_r, a_l, a_r)

Input : T_l, a left hull tree of some tree T,
T_r: a right hull tree of T,
a_l: the rightmost end in the left leftover of the roof,
a_r: the leftmost point in the right leftover of the roof

Output: c_l, b_r: The closest points to $a_l a_r$ in $T_l.hull$ and $T_r.hull$, respectively

```
1 c_l, b_r = T_l.c, T_r.b
2 if SLOPE(PRED(T_l.c), T_l.c) < SLOPE(a_l, a_r) then
3 |   T_l.c = PRED(T_l.c)
4 |   c_l, b_r = GETEXTREMES(T_l, T_r, a_l, a_r)
5 if SLOPE(T_r.b, SUCC(T_r.b)) > SLOPE(a_l, a_r) then
6 |   T_r.b = SUCC(T_r.b)
7 |   c_l, b_r = GETEXTREMES(T_l, T_r, a_l, a_r)
8 return c_l, b_r
```

Table 4.10: Postconditions for GETEXTREMES

Postcondition 1:	$T_l.inv()$
Postcondition 2:	$T_r.inv()$
Postcondition 3:	In $T_l.hull$, $T_l.c$ is closest in perpendicular distance to $a_l a_r$
Postcondition 4:	In $T_r.hull$, $T_r.b$ is closest in perpendicular distance to $a_l a_r$

Since each vertex is scanned past at most once, the runtime is $\mathcal{O}\left(|T_l.hull| + |T_r.hull|\right)$. This completes the proof. □

4.5.3.2 GETBRIDGE

Given two hull trees that satisfy the preconditions given in Table 4.11, Algorithm GETBRIDGE scans the hull chains of the given hull trees to find the bridge that connects them.

Lemma 4.8. *Given two valid hull trees T_l and T_r, satisfying the preconditions of Algorithm* GETBRIDGE, GETBRIDGE *correctly computes the the bridge connecting them in time linear in their sizes.*

Proof. The scan for the left bridge point in T_l is done using its left-to-right cursor $T_l.b$.

Table 4.11: Preconditions for GETBRIDGE

Precondition 1:	$T_l.inv()$
Precondition 2:	$T_r.inv()$
Precondition 3:	$T_l \neq T_r()$
Precondition 4:	$\text{TAIL}(T_l.hull).x \leq \text{HEAD}(T_r.hull).x$

Algorithm 4.5: GETBRIDGE(T_l, T_r)

Input : T_l, a left hull tree of some tree T,
T_r: a right hull tree of T
Output: b_l, c_r: The left and right bridge points connecting the trees T_l and T_r

```
1  if T_l = nil then b = (−∞, −∞)
2  else b = T_l.b
3  if T_r = nil or TAIL(T_l.hull).y > TAIL(T_r.hull).y then c = (+∞, −∞)
4  else c = T_r.c
5  if COUNTERCLOCKWISE(T_l.b, T_r.c, SUCC(T_l.b)) then
6  |   T_l.b = SUCC(T_l.b)
7  |   b, c = GETBRIDGE(T_l, T_r)
8  if COUNTERCLOCKWISE(T_l.b, T_r.c, PRED(T_l.b)) then
9  |   T_l.b = PRED(T_l.b)
10 |   if T_l.b.x > T_l.c.x then T_l.c = T_l.b
11 |   b, c = GETBRIDGE(T_l, T_r)
12 if COUNTERCLOCKWISE(T_l.b, T_r.c, SUCC(T_r.c)) then
13 |   T_r.c = SUCC(T_r.c)
14 |   b, c = GETBRIDGE(T_l, T_r)
15 if COUNTERCLOCKWISE(T_l.b, T_r.c, PRED(T_r.c)) then
16 |   T_r.c = PRED(T_r.c)
17 |   if T_r.b.x > T_r.c.x then T_r.b = T_r.c
18 |   b, c = GETBRIDGE(T_l, T_r)
19 return b, c
```

The scan for the right bridge point in T_r, however, is done using T_r's right-to-left cursor $T_r.c$. On completion, the two cursors will be pointing to the bridge points, as required by GETBRIDGE's Postcondition 3. Since, no other state changes were made to the two trees, the hull tree invariants continue to hold.

Table 4.12: Postconditions for GETBRIDGE

Postcondition 1:	$T_l.inv()$
Postcondition 2:	$T_r.inv()$
Postcondition 3:	$(T_l.b, T_r.c)$ is the bridge connecting the chains $T_l.hull$ and $T.r.hull$

Since each vertex is scanned past at most once, the runtime is $\mathcal{O}\left(|T_l.hull| + |T_r.hull|\right)$. This completes the proof. □

4.5.3.3 TANGENTS

Given two hull trees and a pair of roof points that meet the preconditions given in Table 4.13, Algorithm TANGENTS returns a left tangent point and a right tangent point as seen from the given roof points.

Table 4.13: Preconditions for TANGENTS

Precondition 1:	$T_l.inv()$
Precondition 2:	$T_r.inv()$
Precondition 3:	(a_l, a_r) is monotone

4.5.3.4 DELETE

Algorithm DELETE is a recursive algorithm that breaks into four cases. We shall employ the analogy of a roof caving in from a heavy snow pile. Restricting our analogy to a vertical plane cutting through the home, the remaining roof has a left portion ending with point labeled a_l and a right portion starting with the point a_r.

Algorithm 4.6: TANGENTS(a_l, a_r, T_l, T_r)

Input : a_l, the rightmost end of the left leftover from the roof,
a_r, the leftmost point in the right leftover from the roof,
T_l, a child hull tree of some tree T,
T_r: a child hull tree of T
Output: b, c: tangent to a_l and a_r respectively

```
1  if T_l.b ≠ TAIL(T_l.hull) and CLOCKWISE(a_l, SUCC(T_l.b), T_l.b) then
2      T_l.b = SUCC(T_l.b)
3      if T_l.b.x > T_l.c.x then T_l.c = T_l.b
4      return TANGENTS(a_l, a_r, T_l, T_r)
5  if T_l.b ≠ HEAD(T_l.hull) and COUNTERCLOCKWISE(a_l, T_l.b, PRED(T_l.b)) then
6      T_l.b = PRED(T_l.b)
7      return TANGENTS(a_l, a_r, T_l, T_r)
8  if T_r.c ≠ HEAD(T_r.hull) and CLOCKWISE(T_r.c, a_r, PRED(T_r.c)) then
9      T_r.c = PRED(T_r.c)
10     if T_r.b.x > T_r.c.x then T_r.b = T_r.c
11     return TANGENTS(a_l, a_r, T_l, T_r)
12 if T_r.c ≠ TAIL(T_r.hull) and COUNTERCLOCKWISE(T_r.c, a_r, SUCC(T_r.c)) then
13     T_r.c = SUCC(T_r.c)
14     return TANGENTS(a_l, a_r, T_l, T_r)
15 return T_l.b, T_r.c
```

Table 4.14: Postconditions for TANGENTS

Postcondition 1:	$T_l.inv()$
Postcondition 2:	$T_r.inv()$
Postcondition 3:	$(T_l.b, T_r.c)$ is monotone

Table 4.15: Preconditions for DELETE

Precondition 1:	$\|C\| > 0$
Precondition 2:	C is monotone
Precondition 3:	C is a subchain of $T.hull$
Precondition 4:	$T.inv()$

Algorithm 4.7: DELETE(C, T)

Input : C, a chain of points to be deleted from T,
T: a hull tree
Output: T: The updated hull tree with the chain C deleted from it

1 $j = T.hull.$ LISTDELETE(C)
2 $c_l, b_r =$ GETEXTREMES$(T.l, T.r, T.hull[j-1], T.hull[j])$
▷ Case 1: Neither subtree needed to rebuild the roof ◁
3 **case** $\neg$ COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], c_l)$ *and*
$\neg$ COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], b_r)$
4 | Do nothing
▷ Case 2: Only the right subtree needed to rebuild the roof ◁
5 **case** $\neg$ COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], c_l)$
and COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], b_r)$
6 | $i, k =$ TANGENTS$(T.hull[j-1], T.hull[j], T.r, T.r)$
7 | $T.hull = T.hull[0 : j-1] \cdot T.r.hull[i : k] \cdot T.hull[j : |T.hull|]$
8 | DELETE$(T.r.hull[i : k], T.r)$
▷ Case 3: Only the left subtree needed to rebuild the roof ◁
9 **case** COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], c_l)$
and $\neg$ COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], b_r)$
10 | $i, k =$ TANGENTS$(T.hull[j-1], T.hull[j], T.l, T.l)$
11 | $T.hull = T.hull[0 : j-1] \cdot T.l.hull[i : k] \cdot T.hull[j : |T.hull|]$
12 | DELETE$(T.l.hull[i : k], T.l)$
▷ Case 4: Both subtrees needed to rebuild the roof ◁
13 **case** COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], c_l)$
and COUNTERCLOCKWISE$(T.hull[j-1], T.hull[j], b_r)$
14 | $i, k =$ TANGENTS$(T.hull[j-1], T.hull[j], T.l, T.r)$
15 | $b_l, c_r =$ GETBRIDGE$(T.l, T.r)$
| $T.hull = T.hull[0 : j-1] \cdot T.l.hull[i : b_l] \cdot T.r.hull[c_r : k] \cdot T.hull[j : |T.hull|]$
16 | DELETE$(T.l.hull[i : b_l], T.l)$
17 | DELETE$(T.r.hull[c_r : k], T.r)$
18 **return** T

Table 4.16: Postconditions for DELETE

Postcondition 1:	$T' =$ BUILDTREE$(P \setminus C)$, where P is the set of points in T.
Postcondition 2:	$T'.inv()$

Case 1. *Neither subtree is needed to rebuild the roof.*

This case, depicted in Figure 4.1, results when the deletion of subchain C from $T.hull$ does not result in the violation of the Invariant I_3 (Dominance Invariant). Since no invariant is violated, there is nothing left to do. This case includes the special case when both subtrees are empty.

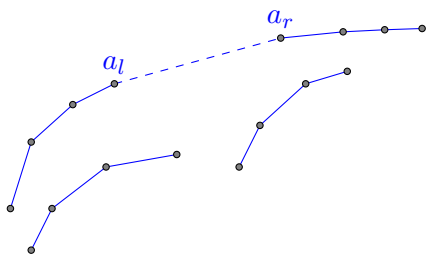

Figure 4.1: Case 1: Invariant I_3 not violated.

Case 2. *Only the right subtree is needed to rebuild the roof.*

This case leads to a violation of Invariant I_3 because now $T.hull$ no longer dominates the hull chain in the right subtree $T.r.hull$, as shown in Figure 4.2. To maintain Invariant I_3, a subchain of $T.r.hull$ will have to be extracted and moved up to become part of $T.hull$.

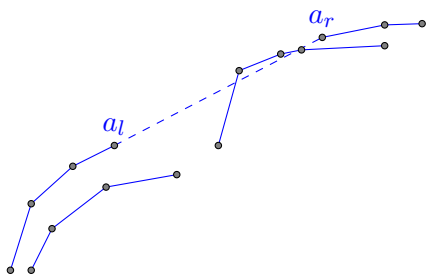

Figure 4.2: Case 2: Invariant I_3 violated only by right child.

Lemma 4.9. *In Case 2, only the vertices of $T_r.hull$ that will be moved up to join the roof are scanned twice.*

Proof. After the call to GETEXTREMES in line 2 of Algorithm DELETE, T_r's left-to-right cursor $T_r.b$ is positioned on the right extreme point relative to $a_l a_r$, by Postcondition 4 of Algorithm GETEXTREMES, but its left-to-right cursor $T_r.b$ is still pointing to HEAD$(T_r.hull)$, not having done any scan so far.

The scan for the left tangent point, visible to a_l and above the segment $a_l a_r$, is done by having $T_r.b$ walk up the chain, until a_l can see no further, at which point the left tangent point has been found. Note that in this walk, all the points that were scanned were seen for the first time.

Similarly, the scan for the right tangent point visible to a_r is done by walking forward or backward. The decision of which walk to take is done in constant time. If the walk backward toward $T_r.b$ is selected, then all the points encountered in this walk will be encountered for the first time. However, if the scan is forward toward the tail of $T_r.hull$, then any point encountered is a point that will be moved up to join the roof. □

Case 3. *Only the left subtree is needed to rebuild the roof.*

This case, depicted in Figure 4.3, is the converse of case 2 – Invariant I_3 is violated with respect to only the left subtree. So, we only need compute the subchain of $T.l.hull$ that needs to move up to repair the roof and restore Invariant I_3.

Lemma 4.10. *In Case 3, only the vertices of $T_l.hull$ that will be moved up to join the roof are scanned twice.*

Proof. The argument is symmetric to that of Case 2. □

Case 4. *Both subtrees are needed to rebuild the roof.*

In this case, Invariant I_3 is violated by both subtrees, as shown in Figure 4.4. So, we need to compute two subchains, on from $T.l.hull$ and the other from $T.r.hull$, which are then moved up to fix the roof and restore Invariant I_3.

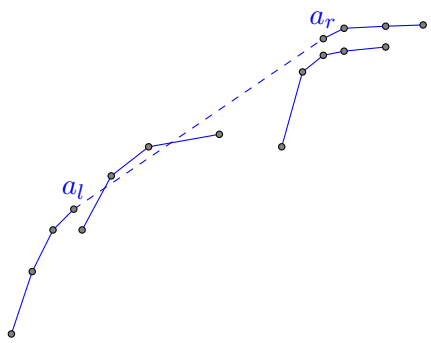

Figure 4.3: Case 3: Invariant I_3 violated only by left child.

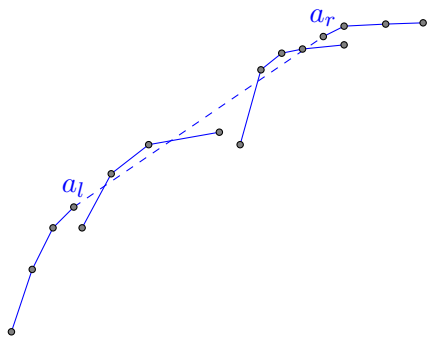

Figure 4.4: Case 4: Invariant I_3 violated by both.

Lemma 4.11. *In Case 4, only the vertices of $T_l.hull$ and $T_r.hull$ that will be moved up to join the roof are scanned twice.*

Proof. After the call to GETBRIDGE in line 15 of Algorithm DELETE, the two cursors $T_l.b$ and $T_r.c$ are already pointing to the left and right bridge points, by Postcondition 3 of Algorithm GETBRIDGE.

The scan for the left tangent point visible to a_l and above the segment $a_l a_r$ is done by walking $T_l.b$ forward or backward. The decision of which direction to walk can be done in constant time. If the walk forward toward $T_l.c$ is chosen, then all the points encountered will be encountered for the first time. However, if the scan is backward toward the head of $T_l.hull$, then any point encountered is one that will be moved up to join the roof.

Symmetrically, the scan for the right tangent point visible to a_r and above the segment $a_l a_r$, is done by walking forward or backward. The decision of which walk to take is done in constant time. If the walk backward toward $T_r.b$ is selected, then all the points encountered in this walk will be encountered for the first time. However, if the scan is forward toward the tail of $T_r.hull$, then any point encountered is a point that will be moved up to join the roof. □

We are now ready to prove the theorem stated earlier about the correctness and runtime of Algorithm DELETE.

Theorem 4.2. *Given a valid hull tree T containing n vertices and a valid hull chain C of k vertices, Algorithm* DELETE *correctly deletes C from T in $\mathcal{O}(k \log n)$ amortized time.*

Proof. We proceed in two steps. First, we shall consider the correctness argument in Part 1 and then the runtime argument in Part 2.

Part 1. *By Lemma 4.7, the points closest to the segment $a_l a_r$ are returned correctly from Algorithm* GETEXTREMES. *This is used to select the correct case. The correctness of each case is already shown in Lemmas 4.9 to 4.11.*

Part 2. *Since points are only ever scanned twice when they will be moved up a level as shown in Lemmas 4.9 to 4.11, and a point is moved up at most once per level and there are no more than $\log n$ levels in T by Invariant I_5, we have that any set of k points C satisfying the preconditions of Algorithm* DELETE *can be deleted from T in $\mathcal{O}(k \log n)$ amortized time.* □

4.5.4 Merge

The merge routine takes as input the four hull trees T_{NW}, T_{NE}, T_{SE}, and T_{SW} with the orientations of NW, NE, SE, and SW and then iteratively performs the following actions for each layer i:

- Extract the root hull chain from each of the hull trees.

- Add their unmarked vertices into a new chain l in clockwise order.
- Mark the vertices in l by adding them to R.
- Delete marked vertices that now appear in the root hull chain of each of the hull trees T_{NW}, T_{NE}, T_{SE}, and T_{SW}.

This process stops when all vertices have been marked.

Table 4.17: Preconditions for MERGE

Precondition 1:	$\min(\|T_{NW}\|, \|T_{NE}\|, \|T_{SE}\|, \|T_{SW}\|) > 0$
Precondition 2:	$T_{NW} \cap R = \varnothing$
Precondition 3:	$T_{NE} \cap R = \varnothing$
Precondition 4:	$T_{SE} \cap R = \varnothing$
Precondition 5:	$T_{SW} \cap R = \varnothing$
Precondition 6:	$T_{NW}.inv()$
Precondition 7:	$T_{NE}.inv()$
Precondition 8:	$T_{SE}.inv()$
Precondition 9:	$T_{SW}.inv()$

Table 4.18: Postconditions for MERGE

Postcondition 1:	$R_{pre} \subseteq R_{post}$ where $R_{post}(R_{pre})$ is the set of marked vertices before (after) the call to MERGE.
Postcondition 2:	$L_{pre} \subseteq L_{post}$
Postcondition 3:	$\text{Int}\,(L_i) \cap \text{Int}\,(L_{i+1}) = \text{Int}\,(L_i)$ for all $i = 1, 2, \cdots \|L\| - 1$.
Postcondition 4:	$T_{NW}.inv()$
Postcondition 5:	$T_{NE}.inv()$
Postcondition 6:	$T_{SE}.inv()$
Postcondition 7:	$T_{SW}.inv()$

Algorithm 4.8: MERGE$(T_{NW}, T_{NE}, T_{SE}, T_{SW}, R)$

Input : T_{NW}, a hull tree with the NW orientation,
T_{NE}, a hull tree with the NE orientation,
T_{SE}, a hull tree with the SE orientation,
T_{SW}, a hull tree with the SW orientation,
R, the set of vertices already extracted from one of the hull trees.

Output: L, A list of merged convex layers.

$L = \varnothing$
$l_{NW} =$ EXTRACTHULL(T_{NW})
$l_{NE} =$ EXTRACTHULL(T_{NE})
$l_{SE} =$ EXTRACTHULL(T_{SE})
$l_{SW} =$ EXTRACTHULL(T_{SW})
foreach $p \in l_{NW} \setminus R$ **do**
    $l = l \cdot p$
    $R = R \cup \{p\}$
foreach $p \in l_{NE} \setminus R$ **do**
    $l = l \cdot p$
    $R = R \cup \{p\}$
foreach $p \in l_{SE} \setminus R$ **do**
    $l = l \cdot p$
    $R = R \cup \{p\}$
foreach $p \in l_{SW} \setminus R$ **do**
    $l = l \cdot p$
    $R = R \cup \{p\}$
$L = L \cdot l$
foreach $p \in R \cap T_{NW}.hull$ **do**
    DELETE(p, T_{NW})
foreach $p \in R \cap T_{NE}.hull$ **do**
    DELETE(p, T_{NE})
foreach $p \in R \cap T_{SE}.hull$ **do**
    DELETE(p, T_{SE})
foreach $p \in R \cap T_{SW}.hull$ **do**
    DELETE(p, T_{SW})
if $\min(|T_{NW}|, |T_{NE}|, |T_{SE}|, |T_{SW}|) > 0$ **then**
    $L = L \cdot$ MERGE$(T_{NW}, T_{NE}, T_{SE}, T_{SW}, R)$
return L

Lemma 4.12. *Given a set S of n points and the four hull trees of S with the four orientations of $NW, NE, SE,$ and SE, the* MERGE *procedure correctly returns the convex layers of S.*

Proof. Denote the output of the MERGE procedure by the sequence $L = (L_1, L_2, \cdots, L_m)$ of m convex polygons L_i, such that:

$$|L_{i+1}| \subseteq |L_i|, \text{ for all } i = 1, 2, \cdots m-1 \tag{4.5.1}$$

We proceed by induction on i. For case $i = 1$, the set of marked vertices (i.e. vertices in R) is initially empty, thus the Preconditions 2-5 are trivially true. The algorithm extracts the four monotone hull l_{NW}, l_{NE}, l_{SE} and l_{SW} and then adds their yet unmarked vertices into l in clockwise order, and also marks them in the process. Thus, at the end of Line 17, l contains a clockwise sequence of the vertices from all the four hull chains, as does R – the set of marked vertices.

At this point, it is possible that one of the preconditions 2-5 might be violated, so the algorithm restores these preconditions before making a recursive call by deleting marked points from the new hull chains.

For an arbitrary case $i > 1$, after Line 17, the following invariant always holds:

$$R = \bigcup_{i=1}^{|L|} L_i \tag{4.5.2}$$

□

Lemma 4.13. *Given a set S of n points and the four hull trees of S with the four orientations of $NW, NE, SE,$ and SE, the* MERGE *procedure executes in $\mathcal{O}(n \log n)$ time.*

Proof. Lines 1 to 5 of Algorithm MERGE take $\mathcal{O}(\log n)$ amortized time per point. Lines 6 to 18 take amortized constant time per point. Lines 19 to 26 also take $\mathcal{O}(\log n)$ amortized

time per point. Thus, the non-recursive part, Lines 1 to 26, is dominated by $\mathcal{O}(\log n)$ amortized time per point. So, the entire algorithm follows the recurrence relation:

$$T(n, m_k) = T(n - m_k, m_{k-1}) + \mathcal{O}(m_k \log n) \tag{4.5.3}$$

where k is the number of layers in L, and the $m_{k-i+1} = |L_i|$, the size of the i-th layer. Expanding this recurrence relation gives:

$$T(n) = \sum_{i=1}^{k} \mathcal{O}(m_i \log n) = \mathcal{O}(\log n) \sum_{i=1}^{k} m_i = \mathcal{O}(n \log n) \tag{4.5.4}$$

since $\sum_{i=1}^{k} m_i = n$. □

4.6 Conclusion

We have given a simple optimal algorithm for the convex layers problem. The pseudocode might appear detailed but that is only because the approach is simple enough that we can deal with all cases explicitly. However, by using four sets of hulls, we only need to work with monotone chains which simplifies our case analyses and make the correctness argument straightforward.

It should be noted that while Chazelle [15] used a balanced tree approach as well, the information stored in our tree corresponds to a different set of polygonal chains.

Chapter 5: Conclusion

This thesis presented a framework for describing approximation algorithms for the convex hull problem. This framework is then applied to a number algorithms found in the literature as well as new algorithms proposed in this thesis. The framework fills a need for practising engineers who need guidance in choosing an appropriate algorithm for their problem. The framework can also serve to analyze future algorithms so that they can be better evaluated and compared to existing algorithms. Many more problems will benefit from a similar framework to help potential implementers select the algorithm that is most appropriate to their problem and context.

A new streaming algorithm for the convex hull is also presented. Its runtime and area error bounds are analyzed. Empirical area and distance error results are also presented. Future work will address analytical distance error analysis. Hershberger and Suri [28, 30, 29] only provided a distance error bound for their algorithm. One future research direction will be to derive an area bound of their algorithm. An empirical comparison of the two algorithms would also be an interesting direction to pursue.

This thesis has studied the problem of maintaining a k-gon within the true convex hull of a stream of incoming points. One might be interested instead in a k-gon that circumscribes the true convex hull. While there are several results [1, 7, 21, 39, 40, 43, 56] for circumscribing k-gons for an offline point set, we are not aware of any published results on circumscribing k-gons for a data stream.

Finally, this thesis also gave a new simple optimal algorithm for the convex layers problem. Detailed pseudocode, space and time complexity results and error bounds of the algorithm are also given. There are other related problems that will benefit from simpler but optimal algorithms. One example is the dynamic convex hull problem, where

the convex hull is to be maintained under arbitrary sequence of insert, delete and query requests.

To date the most practical algorithm for this problem remains that of Overmars and van Leeuwen [45], which runs in $\mathcal{O}\left(\log^2 n\right)$ time per update and $\mathcal{O}\left(\log n\right)$ per query request. While the work of Brodal and Jacob [8] did resolve the long-standing open problem in 2002 by achieving the optimal $\mathcal{O}\left(\log n\right)$ amortized time for update and query requests while using optimal $\mathcal{O}\left(n\right)$ space, their solution depends on data structures that are too intricate and complex to be practical. One line of future work would be to explore simpler and more practical solutions that are nonetheless optimal, either in an amortized sense or even in the worst-case.

Appendix A: Link to Source Code Repository

All the source code used in this thesis can be downloaded from:

```
https://github.com/rrufai/jcg.git
```

Bibliography

[1] AGGARWAL, A., CHANG, J., AND YAP, C. Minimum area circumscribing polygons. *The Visual Computer 1*, 2 (1985), 112–117.

[2] ATALLAH, M., CALLAHAN, P., AND GOODRICH, M. P-complete geometric problems. In *SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 1990), ACM, pp. 317–326.

[3] BENTLEY, J., FAUST, M., AND PREPARATA, F. Approximation algorithms for convex hulls. *Communications of the ACM 25*, 1 (01 1982), 64–68.

[4] BENTLEY, J. L., CLARKSON, K. L., AND LEVINE, D. B. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica 9*, 2 (1993), 168–183.

[5] BENTLEY, J. L., AND SHAMOS, M. I. Divide and conquer for linear expected time. *Inf. Process. Lett. 7*, 2 (1978), 87–91.

[6] BLUNCK, H., AND VAHRENHOLD, J. In-place algorithms for computing (layers of) maxima. *Algorithmica 57*, 1 (2010), 1–21.

[7] BOYCE, J. E., DOBKIN, D. P., DRYSDALE III, R. L., AND GUIBAS, L. J. Finding extremal polygons. *SIAM Journal on Computing 14*, 1 (1985), 134–147.

[8] BRODAL, G. S., AND JACOB, R. Dynamic planar convex hull. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on* (2002), IEEE, pp. 617–626.

[9] BUCHSBAUM, A. L., AND GOODRICH, M. T. Three-dimensional layers of maxima. *Algorithmica 39*, 4 (2004), 275–286.

[10] BYKAT, A. Convex hull of a finite set of points in two dimensions. *Information Processing Letters 7* (1978), 296–298.

[11] CHAN, T. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry 16*, 4 (04 1996), 361–368.

[12] CHAN, T. Output-sensitive results on convex hulls, extreme points, and related problems. *Discrete & Computational Geometry 16*, 4 (04 1996), 369–387.

[13] CHAN, T. M. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry 16*, 4 (04 1996), 361–368.

[14] CHAND, D. R., AND KAPUR, S. S. An algorithm for convex polytopes. *J. ACM 17*, 1 (Jan. 1970), 78–86.

[15] CHAZELLE, B. On the convex layers of a planar set. *IEEE Trans. Information Theory 31* (1985), 509–517.

[16] CHAZELLE, B. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry 10* (1993), 377–409.

[17] CHAZELLE, B., GUIBAS, L. J., AND LEE, D. T. The power of geometric duality. *BIT 25*, 1 (1985), 76–90.

[18] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[19] DALAL, K. Counting the onion. *Random Struct. Algorithms 24*, 2 (2004), 155–165.

[20] DESSMARK, A., LINGAS, A., AND MAHESHWARI, A. Multilist layering: complexity and applications. *Theor. Comput. Sci. 141*, 1-2 (1995), 337–350.

[21] DORI, D., AND BEN-BASSAT, M. Circumscribing a convex polygon by a polygon of fewer sides with minimal area addition. *Computer Vision, Graphics, and Image Processing 24*, 2 (1983), 131 – 159.

[22] EDDY, W. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software 3* (1977), 393–403.

[23] GRAHAM, R. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters 1* (1972), 132–133.

[24] GRAHAM, R., AND YAO, F. Finding the convex hull of a simple polygon. *Algorithms 4* (1983), 324–331.

[25] GREEN, P., AND SILVERMAN, B. Constructing the convex hull of a set of points in the plane. *Computer Journal 22* (1979), 262–266.

[26] HERSHBERGER, J. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett. 33*, 4 (1989), 169–174.

[27] HERSHBERGER, J. Upper envelope onion peeling. *Comput. Geom. Theory Appl. 2*, 2 (1992), 93–110.

[28] HERSHBERGER, J., AND SURI, S. Convex hulls and related problems in data streams. In *Proc. of the ACM/DIMACS Workshop on Management and Processing of Data Streams* (2003).

[29] HERSHBERGER, J., AND SURI, S. Adaptive sampling for geometric problems over data streams. *Computational Geometry 39*, 3 (2008), 191–208.

[30] HERSHBERGER, J., AND SURI, S. Simplified planar coresets for data streams. In *Algorithm Theory–SWAT 2008*. Springer, 2008, pp. 5–16.

[31] HUBER, P. J. The 1972 wald lecture robust statistics: A review. *The Annals of Mathematical Statistics 43*, 4 (1972), 1041–1067.

[32] JARVIS, R. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters 2* (1973), 18–21.

[33] KALLAY, M. The complexity of incremental convex hull algorithms in $\mathbf{R}^d$. *Information Processing Letters 19*, 4 (11 1984), 197–197.

[34] KAPOOR, S. Dynamic maintenance of maxima of 2-d point sets. *SIAM J. Comput. 29*, 6 (2000), 1858–1877.

[35] KAVAN, L., KOLINGEROVA, I., AND ZARA, J. Fast approximation of convex hull. In *ACST'06: Proceedings of the 2nd IASTED international conference on Advances in computer science and technology* (Anaheim, CA, USA, 2006), ACTA Press, pp. 101–104.

[36] KIM, C. E., AND STOJMENOVIC, I. Sequential and parallel approximate convex hull algorithms. *Computers and Artificial Intelligence 14*, 6 (1995).

[37] KIRKPATRICK, D. G., AND SEIDEL, R. The ultimate planar convex hull algorithm. *SIAM J. Comput. 15*, 1 (1986), 287–299.

[38] KLETTE, R. On the approximation of convex hulls of finite grid point sets. *Pattern Recognition Letters 2*, 1 (1983), 19–22.

[39] LOPEZ, M. A., AND REISNER, S. Efficient approximation of convex polygons. *International Journal of Computational Geometry & Applications 10*, 05 (2000), 445–452.

[40] LOPEZ, M. A., AND REISNER, S. Hausdorff approximation of convex polygons. *Computational Geometry 32*, 2 (2005), 139 – 158.

[41] NÄHER, S., AND SCHMITT, D. A framework for multi-core implementations of divide and conquer algorithms and its application to the convex hull problem.

[42] NIELSEN, F. Output-sensitive peeling of convex and maximal layers. *Inf. Process. Lett. 59*, 5 (1996), 255–259.

[43] O'ROURKE, J. Counterexamples to a minimal circumscription algorithm. *Computer Vision, Graphics, and Image Processing 30*, 3 (1985), 364 – 366.

[44] O'ROURKE, J. *Computational geometry in C (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1998.

[45] OVERMARS, M. H., AND VAN LEEUWEN, J. Maintenance of configurations in the plane. *Journal of Computer and System Sciences 23*, 2 (1981), 166 – 204.

[46] PREPARATA, F. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM 22* (1979), 402–405.

[47] PREPARATA, F., AND HONG, S. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM 20*, 1 (01 1977), 87–93.

[48] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: An Introduction*, 3rd ed. Springer-Verlag, 10 1990.

[49] RÉNYI, A., AND SULANKE, R. über die konvexe hülle von n zufällig gewählten punkten. *Probability Theory and Related Fields 2* (1963), 75–84. 10.1007/BF00535300.

[50] SEDGEWICK, R. Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures* (2008), p. 17.

[51] SEDGEWICK, R., AND WAYNE, K. *Algorithms*, 4th ed. ed. Addison-Wesley, Boston, 2011.

[52] SOISALON SOININEN, E. On computing approximate convex hulls. *Information Processing Letters 16* (1983), 121–126.

[53] STOJMENOVIC, I., AND SOISALON-SOININEN, E. A note on approximate convex hulls. *Inf. Process. Lett. 22*, 2 (1986), 55–56.

[54] SUK, T., AND FLUSSER, J. Convex layers: A new tool for recognition of projectively deformed point sets. In *Computer Analysis of Images and Patterns* (1999), pp. 454–461.

[55] ŽUNIĆ, J. Approximate convex hull algorithm—efficiency evaluations. *J. Inf. Process. Cybern. 26*, 3 (1990), 137–148.

[56] WOOD, T. C., AND LEE, H.-C. On the time complexity for circumscribing a convex polygon. *Computer Vision, Graphics, and Image Processing 30*, 3 (1985), 362 – 363.

[57] XU, Z.-B., ZHANG, J.-S., AND LEUNG, Y.-W. An approximate algorithm for computing multidimensional convex hulls. *Applied Mathematics and Computation 94*, 2-3 (1998), 193 – 226.

[58] YAO, A. A lower bound to finding convex hulls. *Journal of the ACM 28* (1981), 780–787.

Curriculum Vitae

Raimi A. Rufai

rrufai@gmu.edu

Education

2003 MS in Computer Science, King Fahd University of Petroleum, KSA

1998 BS in Computer Science, University of Ilorin, Nigeria

Work Experience

2010- ? *Senior Software Developer*, SAP Labs, Inc., Montreal, Canada

2009-2010 *Consultant*, Straj Solutions, Inc., Toronto, Canada

2008-2010 *Graduate Teaching Assistant*, Computer Science Dept., George Mason Univ., Fairfax VA

2007-2008 *Software Engineer*, Sonex Enterprises, Inc, Fairfax VA

2006-2007 *Graduate Research Assistant*, Center for Air Transportation Systems Research (CATSR), Fairfax VA

2005-2006 *Graduate Teaching Assistant*, Computer Science Dept., George Mason Univ., Fairfax VA

2003-2005 *Research Software Engineer/Lecturer*, Info. & Comp. Sci. Dept., University of Petroleum & Minerals, KSA

2000-2002 *Graduate Research Assistant*, Info. & Comp. Sci. Dept., University of Petroleum & Minerals, KSA

1998-2000 *Programmer/Analyst*, SoftWorks Ltd., Lagos, Nigeria