MASTER'S THESIS 2023

Improvements to Planar Convex Hull Algorithms

Björn Magnusson, Erik Amirell Eklöf

ISSN 1650-2884 LU-CS-EX: 2023-35

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-35

Improvements to Planar Convex Hull Algorithms

Förbättringar av algoritmer för konvext hölje i två dimensioner

Björn Magnusson, Erik Amirell Eklöf

Improvements to Planar Convex Hull Algorithms

(through Theoretical and Empirical Analysis)

Björn Magnusson
bjorn.in.magnusson@gmail.com

Erik Amirell Eklöf
erik.amirell.eklof@gmail.com

June 27, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

The task of computing the convex hull given a set of points in the plane is a well known problem for which there are many algorithms with various performance related trade offs. In this thesis we aim to gain a better understanding of how existing algorithms perform in different circumstances.

We have concluded that for most densely distributed inputs, QuickHull is the fastest algorithm. We have also empirically evaluated different variants of QuickHull, as well as special cases where QuickHull performs poorly compared to other algorithms.

Furthermore we present a novel convex hull algorithm that combines the best aspects of Chan's algorithm and the divide and conquer algorithm by Bentley and Shamos.

Keywords: Algorithms, Computational Geometry, Convex Hull, Performance, QuickHull

Acknowledgements

We would like to thank our supervisor Jonas Skeppstedt for supporting us in this project.

Contents

Chapter 1
Introduction

The convex hull (CH) problem is the task of computing the convex hull of a set of input points. One can visualize it in the following way: Imagine a bunch of darts on a dartboard. Place a stretched rubber band around the darts. The shape of the rubber band will be the convex hull of the darts. An example of the convex hull of a set of points is found in Figure 1.1.

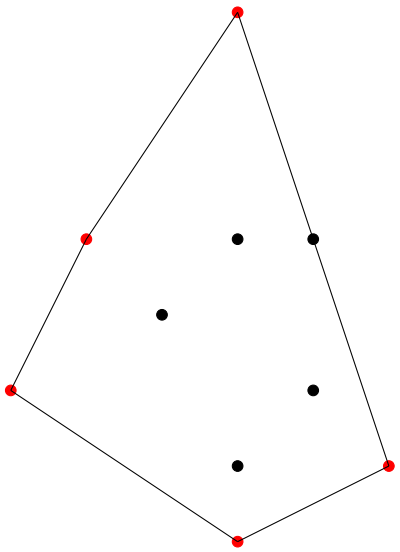


Figure 1.1: Convex hull of a set of points. The convex hull is spanned by some of the input points, marked in red.

The convex hull problem can be applied in different areas including image registration [16] [26], medicine [21] and terrain mapping [25]. Additionally, computing the convex hull can be used for other problems in computational geometry such as Delaunay triangulations and Voronoi diagrams [5], and speeding up the smallest enclosing circle problem [23].

There are several different algorithms to compute the convex hull for a set of points in the plane. We have aimed to get a better understanding of how these algorithms compare to each

other in terms of execution time. This is interesting from both a practical and a theoretical viewpoint. Having algorithms that execute fast in practice means that computation becomes cheaper, both in terms of electricity and time, and can make it attainable to solve larger problem instances. Theoretically understanding what makes an algorithm fast and the limits of the algorithms can help us identify bottlenecks and how to improve algorithms further.

In this thesis we have implemented versions of some well known convex hull algorithms. We have put special attention to the QuickHull algorithm, and tried many different variants of implementing it. We evaluate our implementations against each other and against existing implementations in the CGAL and Qhull libraries. Time measurements and other statistics were collected on different types of inputs in order to find out which algorithms perform best in different circumstances.

Furthermore we present a new convex hull algorithm that we call Refined Chan, which is based on an algorithm by Chan with a combination of refinements mentioned in the original paper [7]. This specific combination has not, to our knowledge, been studied before and our novel theoretical analysis shows that Refined Chan combines time complexities of Chan's algorithm and the divide and conquer algorithm by Bentley and Shamos [6]. Additionally, Refined Chan is simple to understand and analyze.

Finally we analyze how to generate worst case floating point inputs for the QuickHull algorithm, with respect to precision which is generally the limiting factor. The construction is based on existing ideas for how to make QuickHull run slowly, but we have made improvements and identified some pitfalls that need to be avoided to make the construction actually work as intended in practice. We also show an improved upper bound on how deep QuickHull can recurse on floating point inputs.

In our thesis we have answered the following research questions:

- Given some properties and structure of the input points, which planar convex hull algorithm is the least time consuming?
- Can we construct an easy to understand planar convex hull algorithm which has good time complexity both on average and in the worst case?
- How can we construct worst case inputs for the QuickHull algorithm, and how close are these to the theoretical limits?

In chapter 2 we cover some background material about the convex hull problem and existing algorithms to solve it. Chapter 3 introduces and analyzes the novel algorithm Refined Chan. Chapter 4 compares different variants of the QuickHull algorithm. In chapter 5 we discuss how to create worst case inputs for QuickHull. Finally, in chapter 6 we empirically evaluate our algorithms and follow up with a conclusion in chapter 7.

1.1 Contribution Statement

The authors have worked quite independently on different parts of this thesis, but naturally shared and discussed ideas with each other. Björn Magnusson has leaned towards the theoretical side, authoring and inventing the results related to Refined Chan and worst case inputs for QuickHull captured in chapters 3 and 5. Erik Amirell Eklöf has leaned towards the practical side, authoring and inventing the results related to variants of QuickHull captured in

chapter 4. Chapter 2 was authored by Björn Magnusson, with the exception of the part related to QuickHull. The evaluations in chapter 6 were a joint effort by the authors, with the exception of chapters 6.3 and 6.4 being authored by Erik and Björn respectively. Chapters 1 and 7 are a joint contribution of both authors.

Chapter 2
Background

In this chapter we introduce the convex hull problem and how it is commonly solved. We begin with a theoretical perspective in section 2.1, defining the problem and covering some of the most well known algorithms. In section 2.3 and onwards we dive deeper into some of these algorithms and how one would implement them. Our main focus will be on Chan's algorithm [7], Bentley and Shamos divide and conquer algorithm [6], and the QuickHull algorithm [11].

2.1 The Convex Hull Problem

The convex hull of a set of points $CH(P)$ is the smallest convex set that contains the set of points P [20, p. 97]. A set S is convex if for all points $p, q \in S$, the line segment between p and q is fully contained in S.

For dimensions greater than 2, $CH(P)$ will be a polytope with vertices in P, but for the 2 dimensional, planar case the CH is always a convex polygon [20, p. 97-104]. We call the points in P that are vertices of $CH(P)$ convex hull vertices.

Generally we will think of the task of computing the planar CH as receiving n 2D input points, and returning the h convex hull vertices in counter clockwise order starting with the lexicographically smallest point, as displayed in figure 2.1. We know n before execution but h is unknown until the output has been calculated. Some algorithms will have time complexities that depend on h and are usually called output-sensitive, whilst others only depend on n.

Interest in convex hull algorithms began in the early 70s with Graham and Jarvis presenting one algorithm each in 1972 and 1973 respectively [10] [14]. Graham scan runs in $O(n \log n)$ time and performs an angular sort and then a linear filtering to solve the problem. Jarvis march has time complexity $O(nh)$ and incrementally finds each vertex on the convex hull. It resembles a selection sort [5] [20].

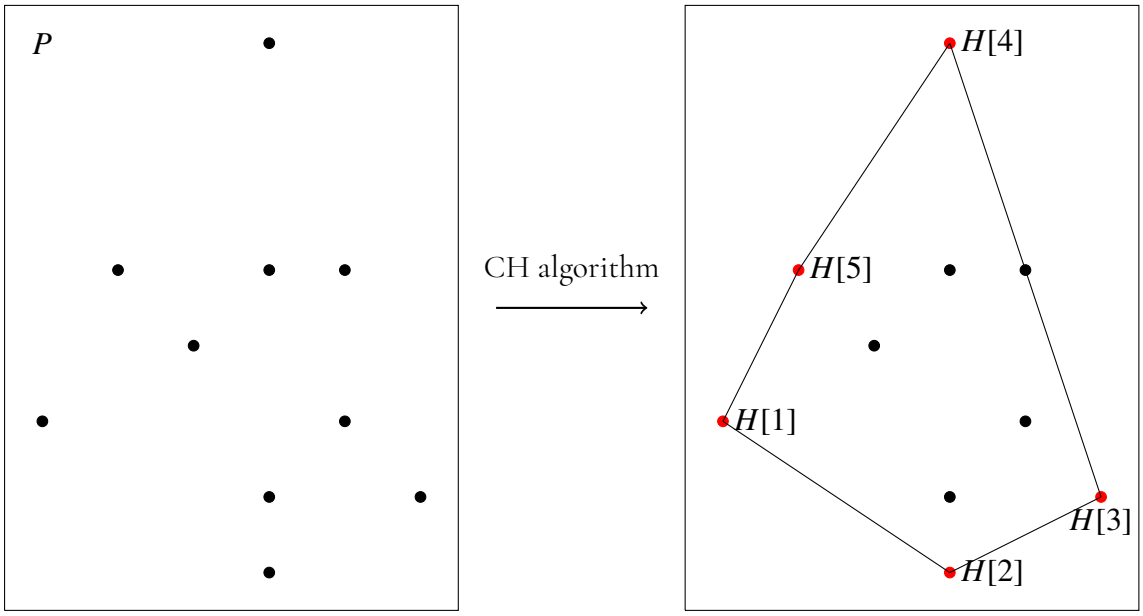


Figure 2.1: Expected behavior of a Convex Hull algorithm, getting $n = 10$ points as input in a list P, and returning $h = 5$ convex hull vertices in a list H in counter clockwise order starting with the smallest point.

Since $h \leq n$ is a tight bound on h, Graham scan is clearly better for worst case performance. One can also show that Graham scan is worst case optimal since the task of sorting n numbers can be reduced to planar convex hull , and sorting n numbers is $\Omega(n \log n)$ under most models of computation [20, p. 100] [19]. For many practical inputs however, h is much smaller than n is, which could make Jarvis march a viable option.

This gap between time complexities was closed in 1986 by Kirkpatrick and Seidel when they presented an $O(n \log h)$ algorithm which is theoretically fast both when h is small and large [15]. Their algorithm is a Divide and Conquer style algorithm with a twist they call "Marriage before conquest". Kirkpatrick and Seidel also proved that $O(n \log h)$ is worst case optimal if complexity is measured in both n and h. One problem with their algorithm is that it is tedious to implement and actual performance is quite poor [17] [9].

In 1996 Chan invented a new $O(n \log h)$ convex hull algorithm which is much simpler to implement [7]. It combines Graham scan and a variant of Jarvis march with a clever grouping trick to guess how large the output size is, aborting and trying a larger size upon failure. At the end of his paper, Chan mentions some possible refinements that might help performance in practice.

Another interesting convex hull algorithm is the divide and conquer algorithm by Bentley and Shamos from 1978 [6]. We call this algorithm MergeHull, because it resembles the MergeSort algorithm [5] [20] in the way that it splits the data into small sets and then recursively joins the solutions together until the convex hull of the complete set is obtained. MergeHull matches Graham scan's $O(n \log n)$ worst case performance. What is particularly interesting about MergeHull is that it achieves $O(n)$ expected time on many randomized inputs, which is a better bound than the $O(n \log h)$ bound would be on these inputs.

Monotone Chain is an $O(n \log n)$ algorithm of practical interest. It was invented by An-

drew in 1979 [3], and is similar to Graham Scan but has a simpler preprocessing step [9].

We have also investigated the QuickHull algorithm, which was initially proven by Scott Greenfield in 1990 [11]. QuickHull is a divide and conquer algorithm that in some ways resembles the QuickSort algorithm for sorting. QuickHull initially finds two points that are guaranteed to be on the convex hull, such as those with maximum and minimum x coordinate, and splits the remaining points into two sets depending on which side of the line between the first two hull points the point is on. The algorithm then proceeds with a divide and conquer phase. Given two points H_L and H_R that are known to be on the convex hull and a set of points P that are all oriented to the left of the line from H_L to H_R.

1. Find the point H_M in P that is furthest from the line H_L to H_R, and mark this point as being on the convex hull.
2. Divide P into three subsets using the triangle formed by the points H_L, H_M, and H_R: those inside the triangle (which are now known to not be part of the convex hull), those to the left of the line from H_L to H_M, and those to the right of the line from H_R to H_M. We will refer to these sets as P_T, P_L, and P_R respectively.
3. If P_L is not empty, recurse using P_L as P and H_L & H_M as the two hull points.
4. If P_R is not empty, recurse using P_R as P and H_M & H_R as the two hull points.

Generally one would expect QuickHull to run in $O(n \log h)$ time, if the sets P_L and P_R are equal in size. If a lot of points are removed because they are in P_T, QuickHull can have $O(n)$ behavior. In the worst case, QuickHull runs in $O(nd)$ time where d is the depth that it recurses to. Clearly $d < h$, but we discuss this in more detail in chapter 5.

2.1.1 Definition of Truly Sublinear Inputs

To formalize the properties of inputs where h is small compared to n we define truly sublinear inputs, inspired by Bentley and Shamos [6]. A truly sublinear input to the convex hull problem is an input generated by sampling n points independently from a random distribution such that the expected number of convex hull vertices is $O(n^p)$ where $p < 1$ is a real number.

A consequence of this definition is that if we select some subset of k points from a truly sublinear input without looking at the coordinate values of the input, we can model this subset as a truly sublinear input of size k.

Many natural random distributions fulfill the truly sublinear property.

- **Circle**: If points are sampled independently uniformly at random inside a circle, the expected number of hull points is $h \in O(n^{1/3})$. [6] [9]
- **Polygon**: If points are sampled independently uniformly at random inside a convex polygon with r corners, the expected number of hull points is $h \in O(r \log n)$. [6] [9]
- **Gaussian**: If points are sampled independently from a normal distribution in 2 dimensions, the expected number of hull points is $h \in O(\sqrt{\log n})$. [6] [9]
- **Independent coordinates**: If points are sampled such that the x and y coordinates are independent from each other, the expected number of hull points is $h \in O(\log n)$ [6]. For example, one could use this bound on an axis aligned square, yielding the same bound as the Polygon bound.

2.2 On notation

In our pseudo code we make use of lists. These are 1-indexed, thus if A is a list, the first element is denoted $A[1]$, and the last element is denoted $A[|A|]$, with $|A|$ denoting the number of elements in A. We will also use the notation $A[i \ldots j]$ to denote the sub-list of elements $A[i]$ through $A[j]$, with both endpoints included. We consider this sub-list to reference the same memory as the outer list, so that constructing a sub-list can be done in constant time with no copying of elements, and so that modifications of elements in the sub-list also affect elements in the outer list.

We will at times refer to points as being smaller/larger than other points. We perform this comparison lexicographically, meaning that point a is smaller than point b if either a's X coordinate is smaller than b's X coordinate, or both X coordinates are the same and a's Y coordinate is smaller than b's Y coordinate.

2.3 Geometric Helper Functions

We begin by introducing two small helper functions which perform the actual geometric computations. The side of line test checks which side of a line a point is located, and is a very common abstraction to use for convex hull algorithms.

The better hull point test takes care of cases where points are collinear (3 or more points on a line) by comparing distances. Most convex hull algorithms use some kind of variation of this test.

2.3.1 Side of Line Test

Essentially a side of line test answers the question:

> If l is a directed line from point a to point b, on what side of l does point c lie?

This basic query can be used to implement most convex hull algorithms. In Figure 2.2 the expected return value for different inputs to a side of line test is displayed. The gold standard for implementing a side of line test is by performing a cross product check. One computes the signed area spanned by $\triangle abc$ and checks the sign of the area. This technique was first proposed by AKL and Toussiant [2]. What is neat about computing the cross product is that it avoids division, as opposed to computing slopes or angles.

The pseudo code for the side of line test is found in Algorithm 1. Note that in the argument list we write $a \rightarrow b$ to help us remember that the function looks at the directed line from a to b. This is just syntactic highlighting, the function still takes 3 points as arguments.

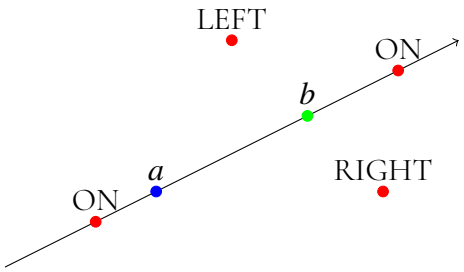


Figure 2.2: Illustration of result of SideOfLine for different choices of c (red)

Algorithm 1: Geometric helper functions

```
1:  function SideOfLine(a → b, c)  ▷ Points given in Cartesian coordinates as a = (a_x, a_y) etc.
2:      S ← (b_x − a_x)(c_y − a_y) − (b_y − a_y)(c_x − a_x)
3:      if S = 0 then
4:          return ON
5:      else if S > 0 then
6:          return LEFT
7:      else
8:          return RIGHT
9:      end if
10: end function
11:
12: function BetterHullPoint(a → b, c)
13:     return SideOfLine(a → b, c) = RIGHT OR
14:             (SideOfLine(a → b, c) = ON AND dist(a, b) < dist(a, c))
15: end function
```

2.3.2 Better Hull Point Test

In order to correctly select hull points that are collinear it is also useful to look at actual distances. BetterHullPoint() in Algorithm 1 takes care of this logic by selecting the point that is further away if they are collinear. Essentially it answers the question:

> If we are building the convex hull in counter clockwise order and a was the most recently selected vertex, should c replace b as the next candidate vertex?

Figure 2.3 shows the expected return value of a better hull point test. The function Dist(a, b) in Algorithm 1 is some distance metric, for example Manhattan or Euclidean distance.

2.4 Merge Function

The merge function merges a set of k convex hulls $H_1, \ldots H_k$ into one combined convex hull H^*. If there are in total $n = \sum_{i=1}^{k} |H_i|$ points in the input and $h = |H^*|$ points in the output it

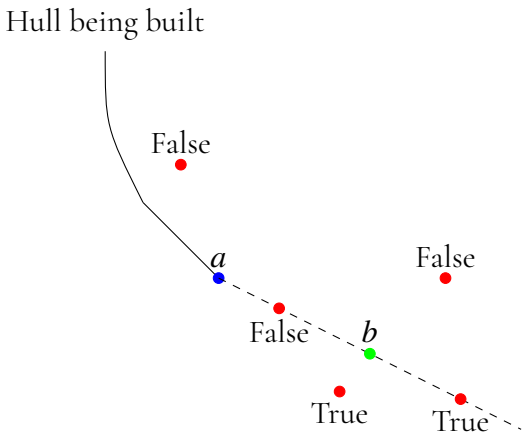


Figure 2.3: Illustration of result of BetterHullPoint(a, b, c) for different choices of c (red)

runs in $O(n + k \cdot h)$ time. Additionally the merge function takes an integer parameter H^*_{max} specifying a maximal size of H^*. If $h > H^*_{max}$ merge aborts and returns `incomplete` in $O(n+k \cdot H^*_{max})$ time. This means that the time complexity of merge is $O(n+k \cdot \min(h, H^*_{max}))$.

The idea of the algorithm is to incrementally find new points on the convex hull, similar to Jarvis march. However, instead of comparing all points and picking the best one according to the better hull point test, it is sufficient to only consider the tangents to the input hulls. This means that the cost per added hull point is k instead of n since there is one tangent to each H_i. Keeping track of the tangents to each H_i can be done with pointers that slide around the hulls, adding just an overall $O(n)$ cost to the algorithm. The idea for this implementation is taken from Chan's paper, but with the use of refinement idea 5 by E. Welzl. [7].

Note that when we say tangent we generally mean the right hand tangent, that is, the line through a given point that touches a given hull from the right side.

Merge starts with initializing the tangent pointers to the smallest point on each hull. This can be thought of as representing the tangents from an imaginary point infinitely far away in the $(\epsilon, 1)$ direction, with $\epsilon > 0$ being a sufficiently small number. We then add the smallest point P from all H_i as the first point in H^*. After this we repeatedly update the pointers to represent tangents from P and then select a new P and add it to H^* by comparing the tangent points. Figure 2.4 shows the initial steps, while Figure 2.5 shows how the Merge function finds a new hull vertex P^n by comparing the tangents from the previous hull vertex P.

The pseudo code can be found in Algorithm 2. We assume that each H_i can be accessed circularly. I.e. the lists loop around so that $H_i[|H_i| + 1] = H_i[1]$. In practice this would be handled by modulo checking the indices, but that would make the pseudo code less concise. Note that if k is a constant, Merge has time complexity $O(n)$ since $h \leq n$.

2.4.1 Correctness

Let us start with assuming that the while loop on line 7 correctly identifies the index $T[i]$ to the tangent point for each input hull H_i.

In this case, Algorithm 2 is essentially performing a Jarvis march [14] but to find the next hull point only considers the tangent points from P to each H_i. This is sufficient because in

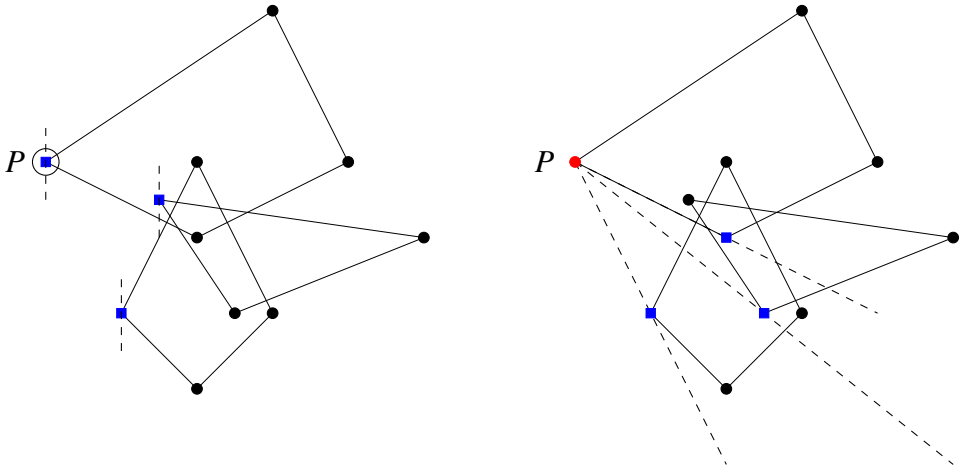


Figure 2.4: Illustration of Merge starting with vertical tangents (blue squares) and selecting leftmost point P. Tangents are then updated to the right figure by iterating forwards on each hull.

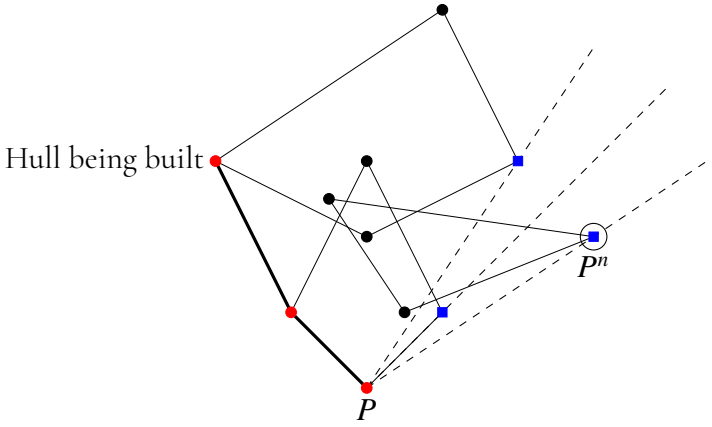


Figure 2.5: Illustration of Merge finding next hull point P^n by comparing tangents to each hull (blue squares) from P.

Jarvis march we always add the point maximizing the angle from P, breaking ties on distance, and this is exactly what we do when computing the tangent on each hull.

What remains to prove is that we correctly identify the tangent points. We will split the logic into two cases. Either P lies on the border of H_i or outside of it.

If P is outside of H_i we have a situation like in Figure 2.6. What is important is that the old value of $H_i[T[i]]$ is visible from P, because l is a right tangent, and P is not to the left (above) of l because in that case $H_i[T[i]]$ would have been picked instead of P on line 17 in Algorithm 2. This means that line 7 starts with a visible vertex T_i^c of H_i, and is thus doing right turns until finding the tangent. An edge case is if the right tangent from P touches a side of H_i, in which case the vertex that is further away from P will be selected, which is also the desired behavior in the Jarvis March.

If P is on the border of H_i it needs to be a vertex of H_i because if it was on an edge it would not be a vertex of H^*. In this case line 8 will iterate exactly one step forward finding the next vertex on H_i, or if $|H_i| = 1$ it will not do anything.

Algorithm 2: Merge function

```
1:  function MERGE([H_1, H_2, ... H_k], H*_max)
2:      T ← [1, 1, ..., 1]                ▷ Size k. H_i[T[i]] represents the current tangent to H_i
3:      P ← Smallest point of all H_i[1]   ▷ P is always the point most recently added to H*.
4:      H* ← [P]
5:      for h = 1 ... H*_max do
6:          for i = 1 ... k do
7:              while true do                                  ▷ Update the tangent to H_i
8:                  T^c_i ← H_i[T[i]]                          ▷ Current tangent point
9:                  T^n_i ← H_i[T[i] + 1]                      ▷ Next tangent point
10:                 if BETTERHULLPOINT(P → T^c_i, T^n_i) then
11:                     T[i] ← T[i] + 1
12:                 else
13:                     break
14:                 end if
15:             end while
16:         end for
17:         P^n ← Best point of all H_i[T[i]] according to BETTERHULLPOINT(P → _, _)
18:         if P^n = H*[1] then
19:             return H*
20:         else
21:             P ← P^n
22:             H* ← H* ∪ P
23:         end if
24:     end for
25:     return INCOMPLETE
26: end function
```

2.4.2 Time Complexity

We start by noting that at the end of execution $T[i] \leq 2 \cdot |H_i|$. This is motivated by the fact that as we are building H^*, the visible parts of H_i will rotate one revolution around H_i. This means that the pointer keeping track of the right tangent on H_i must have completed less than two revolutions.

This implies that the innermost while loop of line 7 in Algorithm 2 will loop at most $2 \cdot n$ times aggregated over all i and all loops of h. The loops on lines 6 and 17 introduces an extra $O(k \cdot \min(h, H^*_{max}))$ overhead, which yields the total time complexity $O(n + k \cdot \min(h, H^*_{max}))$.

2.5 MergeHull

MergeHull is straightforward to implement using the Merge function of Algorithm 2. Essentially we use the fact that we can merge two convex hulls in linear time. The version presented here is slightly different from the original, since Bentley and Shamos use a different way of combining two hulls in linear time [6].

In Algorithm 3 we present both a recursive and iterative implementation. The recursive

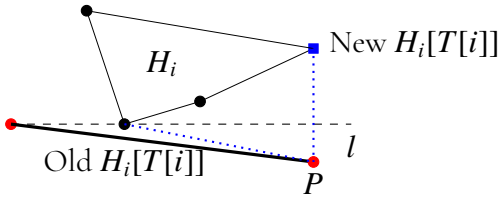


Figure 2.6: Illustration of Merge finding the correct tangent when HullPoint is outside of H_i.

implementation is more natural and straightforward, whilst the iterative implementation is presented for its similarities to the Refined Chan algorithm that is introduced in Chapter 3.

Algorithm 3: MergeHull Implementationes

```
1:  function MergeHullRec(P)                                   ▷ A list of n points
2:      if n = 1 then
3:          return P
4:      else
5:          H_1 ← MergeHullRec(P[1 ... ⌈n/2⌉])                 ▷ First half of P
6:          H_2 ← MergeHullRec(P[⌈n/2⌉ + 1 ... n])             ▷ Second half of P
7:          return Merge([H_1, H_2], ∞)
8:      end if
9:  end function
10: function MergeHullIterative(P)                             ▷ A list of n points
11:     Ĥ ← A list of n hulls, one for each point in P
12:     for i = 1 ... ⌈log n⌉ do
13:         Ĥ ← Merge hulls from Ĥ pairwise into half as many hulls   ▷ Rounded up if odd
14:     end for
15:     return Ĥ[1]                                            ▷ Ĥ only contains one element
16: end function
```

It is important that the partitions of P are computed in constant time on line 5 and 6. This can be achieved by passing pointers to the endpoints of the array segments. Both implementations clearly generate the correct answer because they just merge all input points into one hull and return it. The reason that we loop exactly $k = \lceil \log n \rceil$ times until $|\hat{H}| = 1$ on line 12 is motivated by the fact that $2^{k-1} < n \leq 2^k$ so we need to loop more than $k - 1$ times but k is sufficient.

2.5.1 Worst Case Time Complexity

The Recursive implementation of Algorithm 3 is clearly $O(n \log n)$ in the worst case because at most a linear amount of work is spent in the Merge step and the recursion depth is $\log n$. Formally this can be proven using the Master Theorem [8, p. 103].

Likewise the iterative implementation of Algorithm 3 loops $\log n$ times and runs in linear time for each iteration giving $O(n \log n)$ time complexity.

2.5.2 Average Case Time Complexity

If $T(n)$ represents the expected time to run the recursive implementation of Algorithm 3 for a truly sublinear input of size n we see that we make two calls to truly sublinear inputs of half the size taking in expectation $T(\lceil \frac{n}{2} \rceil)$ time each and to merge the results we need $O(n^p)$ time in expectation. If we assume n is a power of 2 this yields the following equation:

$$T(n) = 2T(n/2) + O(n^p); T(1) = O(1); p < 1$$

Which has the solution $T(n) = O(n)$ according to the Master Theorem [8, p. 103]. For n that is not a power of 2 it can be rounded up to a power of 2 (at most doubling it) and $O(n)$ still holds.

We exclude the proof for the iterative implementation, because it is essentially covered by the proof in section 3.2.

2.6 Chan's Algorithm

Chan's algorithm [7] uses a grouping trick where one picks a number m and guesses that $h \leq m$. If this is the case we form groups of size m and merge them, in $O(n \log m)$ time. Picking m on the format $m = 2^{2^t}$ for integers t yields an overall time complexity $O(n \log h)$. The pseudo code is found in Algorithm 4. Chan's original version uses a Merge function that binary searches to find tangents. The approach taken by us in Algorithm 2 is however mentioned as an improvement, essentially removing a $\log n$ factor from line 7.

Algorithm 4: Chan's algorithm

```
1:  function CHAN(P)                                          ▷ A list of n points
2:      for t = 1, 2 ... do
3:          m ← 2^(2^t)
4:          Partition P into subsets P_1, P_2 ..., P_⌈n/m⌉ of size ≤ m
5:          H_i ← CH(P_i) for all 1 ≤ i ≤ ⌈n/m⌉ using any O(n log n) CH algorithm.
6:          H*_max = m
7:          if MERGE([H_1, ..., H_⌈n/m⌉], H*_max) ≠ INCOMPLETE then   ▷ Check if h ≤ H*_MAX
8:              return MERGE([H_1, ..., H_⌈n/m⌉], H*_max)
9:          end if
10:     end for
11: end function
```

2.6.1 Time Complexity

We see that when $m \geq h$ line 6 in Algorithm 4 will compute the correct hull and return it. So the for loop will iterate until $t = \lceil \log \log h \rceil$. For each iteration, line 5 takes $\lceil \frac{n}{m} \rceil O(m \log m) = O(n \log m)$ time, and line 7 takes $O(n + \lceil \frac{n}{m} \rceil m) = O(n)$ time. If $m > n$ the analysis does not hold since we can't replace $\lceil \frac{n}{m} \rceil$ with $\frac{n}{m}$, but in this case have already obtained the final hull on line 5.

For some constant C, the total time spent will be bounded by:

$$\sum_{t=1}^{\lceil \log \log h \rceil} C \cdot n \log m = \sum_{t=1}^{\lceil \log \log h \rceil} C \cdot n \cdot 2^t = C \cdot n \sum_{t=1}^{\lceil \log \log h \rceil} 2^t < C \cdot n \cdot 2^{\lceil \log \log h \rceil + 1} < 4C \cdot n \cdot \log h$$

Which shows that Chan's algorithm runs in $O(n \log h)$.

2.6.2 Refinements

Chan mentions some refinement ideas at the end of his paper [7]. Relevant for our future analysis are the ideas named Idea 2,3 and 5. Idea 5 relates to replacing a binary search for sweeping pointers in the merge function, and is already incorporated in Algorithm 2.

Idea 2 is to tweak the value of H^*_{max} on line 6. Chan mentions picking $\frac{m}{\log m}$ which would reduce the cost of line 5 to $O(n)$ if one didn't already use Idea 5.

Idea 3 is to reuse the computed hulls $H_1, \ldots$ from previous steps in the iteration. One would merge the previous hulls pairwise until reaching the new desired hull size, this would essentially correspond to what Iterative MergeHull does.

Chapter 3
Refined Chan

In this section we present a novel algorithm that we will call Refined Chan's algorithm, along with theoretical limits for the time complexity.

Refined Chan combines the time complexities of Chan's algorithm and MergeHull, giving an $O(n \log h)$ worst case time complexity and an $O(n)$ expected time complexity on truly sublinear inputs.

The pseudo code is found in Algorithm 5. It clearly returns the correct solution because whenever it returns it will return the result of merging all the input points into a hull.

Algorithm 5: RefinedChan

```
1:  function RefinedChan(P)                                   ▷ A list of n points
2:      Ĥ ← A list of n hulls, one for each point in P
3:      for i = 1 ... ⌈log n⌉ do
4:          Ĥ ← Merge hulls from Ĥ pairwise into half as many hulls   ▷ Rounded up if odd
5:          if i = 2^t for some integer t > 0 then
6:              H*_max = 2^i / i
7:              if Merge(Ĥ, H*_max) ≠ INCOMPLETE then          ▷ Check if h ≤ H*_MAX
8:                  return Merge(Ĥ, H*_max)
9:              end if
10:         end if
11:     end for
12:     return Ĥ[1]                                           ▷ Ĥ only contains one element
13: end function
```

We can see that without lines 5-11 in Algorithm 5 we have exactly the iterative implementation of MergeHull in Algorithm 3. Lines 6-10 of Refined Chan correspond to the same lines in Chan's Algorithm found in Algorithm 4. Whenever $i = 2^t$, $\hat{H}$ will contain convex hulls representing at most $m = 2^i = 2^{2^t}$ input points each. This means that we are effectively reusing the computed hulls from previous steps (smaller t) which corresponds to what Chan

presents as improvement Idea 3 [7]. Furthermore we have changed H^*_{max} to $\frac{m}{\log m}$ on line 6, corresponding to Idea 2.

Even though the ideas used to create Refined Chan are not original for this thesis, to our knowledge we are the first to pick this particular combination of improvements and analyze them. Particularly the fact that Refined Chan (or any variant of Chan's algorithm) has the same expected time complexity as MergeHull is, to our knowledge, novel.

We present proofs of Refined Chan worst case time complexity $O(n \log h)$ in section 3.1 and expected time complexity $O(n)$ on truly sublinear inputs in section 3.2.

3.1 Worst Case Time Complexity

We will prove that Refined Chan has a worst case time complexity of $O(n \log h)$. We will do this by proving that each loop of i takes at most $O(n)$ time, and that i loops at most until $O(\log h)$.

Note that since $2^i < 2n$ we have that $\lceil \frac{n}{2^i} \rceil < \frac{n}{2^i} + 1 < 3\frac{n}{2^i}$. This means that for the purpose of time complexity we can replace $\lceil \frac{n}{2^i} \rceil$ with $\frac{n}{2^i}$.

On line 4 we merge $\lceil \frac{n}{2^i} \rceil$ pairs of convex hulls, each hull containing at most 2^{i-1} points. These calls take $O(2^i + 2 \cdot 2^i) = O(2^i)$ time each. In total we do $\lceil \frac{n}{2^i} \rceil$ calls so the total amount of work is $O(n)$ per loop.

On lines 5-11 we sometimes merge $\lceil \frac{n}{2^i} \rceil$ hulls containing at most n points in total. We abort if the resulting hull has more than $H^*_{max} = \frac{2^i}{i}$ vertices. This takes $O(n + \lceil \frac{n}{2^i} \rceil \cdot \frac{2^i}{i}) = O(n + \frac{n}{i}) = O(n)$ time in total.

We have shown that each loop of i takes $O(n)$ time. We now claim that i will loop at most until $4 \cdot \log h$ or a constant (for small values of h).

Since there must be an even power of 2 between $2 \log h$ and $4 \log h$, when i assumes this value, we get that $H^*_{max} = \frac{2^i}{i} \geq \frac{h^2}{i} \geq h$ since $h \geq 4 \log h \geq i$ when $h \geq 16$. This means that for this i, line 9 will return the hull H^*. For $h < 16$ we see that when $i = 8$ we have $\frac{2^8}{8} = 32$ so in these cases i will loop at most 8 times.

Thus the worst case time complexity of Refined Chan $O(n \log h)$.

3.2 Average Case Time Analysis

In the previous section we showed that each loop of i takes at most $O(n)$ time, which is not a tight bound in the average case. The idea for this section is to improve the bound to a geometrically decreasing sum which in turn sums to $O(n)$ instead of $O(n \log h)$. The analysis assumes that the input is truly sublinear.

On line 4 we have $\lceil \frac{n}{2^i} \rceil$ pairs of convex hulls that we will merge. Each hull represents at most 2^{i-1} points so its expected size is $O(2^{(i-1)p})$. Each merged hull represents at most 2^i points so its expected size is $O(2^{ip})$. The expected running time of each merge is therefore $O(2 \cdot 2^{(i-1)p} + 2 \cdot 2^{ip}) = O(2^{ip})$, and since we merge $\lceil \frac{n}{2^i} \rceil$ pairs it has an expected running time of $\lceil \frac{n}{2^i} \rceil O(2^{ip}) = O(\frac{n}{2^{i(1-p)}})$.

Summed over all loops of i this is $O(n)$ since it is a decreasing geometric sum. This analysis would cover the iterative implementation of MergeHull in Algorithm 3.

Lines 5-10 are only executed when $i = 2^t$ for integers $t = 0, 1, \ldots$. It merges $\lceil \frac{n}{2^i} \rceil$ hulls of expected size $O(2^{ip})$, additionally aborting if the output is larger than $\frac{2^i}{i}$. Thus it takes an expected time $O(\frac{n}{2^i} 2^{ip} + \frac{n}{2^i} \frac{2^i}{i})$. The first term is the same as for line 4 and must thus have at most a linear contribution summed over all i. The second term is equal to $O(\frac{n}{2^i})$, which clearly is $O(n)$ when summed over all t.

In conclusion we have shown that summed over all loops of i, the expected accumulated running time of both line 4 and lines 5-11 is expected to be linear, so the expected running time of refined Chan is $O(n)$ on truly sublinear inputs.

3.3 Simplified Variant

The presented algorithm 5 can be simplified somewhat. The key insight is that even though it is formally achieved by adding refinements to Chan's algorithm, what we obtain is the MergeHull algorithm with an extra step that checks if h is small and in that case breaks early. This step can be made simpler.

If we only care about getting worst case time complexity of $O(n \log h)$, the simplified version in algorithm 6 achieves this, because it still takes $O(n)$ time in every loop of i and i loops until at most $\lceil \log h \rceil$.

Algorithm 6: Simplified variant of Refined Chan that achieves $O(n \log h)$ time complexity

```
1:  function SimplerRefinedChan(P)                          ▷ A list of n points
2:      Ĥ ← A list of n hulls, one for each point in P
3:      for i = 1 ... ⌈log n⌉ do
4:          Ĥ ← Merge hulls from Ĥ pairwise into half as many hulls   ▷ Rounded up if odd
5:          H*_max = 2^i
6:          if Merge(Ĥ, H*_max) ≠ INCOMPLETE then           ▷ Check if h ≤ H*_MAX
7:              return Merge(Ĥ, H*_max)
8:          end if
9:      end for
10:     return Ĥ[1]                                         ▷ Ĥ only contains one element
11: end function
```

If we also want $O(n)$ expected time complexity on truly sublinear inputs, we can reduce H^*_{MAX} to 2^{ip} for some $p < 1$ and make a similar argument as is section 3.2. In this case i will loop until at most $\lceil \frac{\log h}{p} \rceil$.

Chapter 4
QuickHull

In this chapter we will discuss the different implementation strategies for QuickHull that we have tested, and potential advantages and disadvantages of these.

For all implementations of QuickHull, we chose to represent sets of points using arrays of contiguously allocated memory.

4.1 Partitioning Strategies

There are several occasions in the QuickHull algorithm where a set of points need to be divided into two or three disjoint subsets. In order to implement QuickHull with $O(1)$ extra memory, we implement this division into disjoint subsets by partitioning the points such that each subset occupies a contiguous range of the original memory. Partitioning refers to the operation of moving points so that all points satisfying a predicate are moved to the start of the array.

By partitioning the points such that those in P_R appear before the point H_M, which in turn appears before the points in P_L, the points that are on the convex hull will at the end of the algorithm be ordered counter clockwise along the hull.

We have tested four different approaches for partitioning points:

i First partition the points to extract those to the left of the line from $H_M \rightarrow H_R$, then partition the remaining points to extract those to the left of the line H_L to H_M.

ii First partition the points by comparing their X-coordinate with that of H_M. After this, partition the points to the left of H_M to extract those to the left of the line $H_L \rightarrow H_M$, and likewise for the points to the right of H_M with the line $H_R \rightarrow H_M$.

iii Partition the points in a single scan by swapping points that should be in P_R to the beginning and points that should be in P_L to the end.

iv During the initial phase, sort the points above and below the initial line (separately) by X-coordinate, with the points above being sorted in the opposite order to those below. This way, points will always be partitioned according to whether their X-coordinate is less than that of H_M and approach ii can be used without the first partitioning step.

See algorithms 7 through 10 for pseudo code for the different partitioning strategies. The function Partition(A, P) partitions the list A according to the predicate P (modifying A). The function returns a sub-list of A containing the elements where the predicate was true (which references the same memory as A so that modifications of elements in the returned list affect the elements in A).

Algorithm 7: Partition Strategy i

function PartitionPoints(P, H_L, H_R, H_M)
  Swap($P[|P|]$, P[IndexOf(P, H_M)]) ▷ Temporarily moves H_M to the end.
  $P_R \leftarrow$ Partition($P[1 \ldots |P|-1],\ p \mapsto$ LeftOfLine($H_M \rightarrow H_R, p$))
  Swap($P[|P|]$, $P[|P_R|+1]$) ▷ Moves H_M so that it's in between P_R and P_L
  $P_L \leftarrow$ Partition($P[|P_R|+2 \ldots |P|],\ p \mapsto$ LeftOfLine($H_L \rightarrow H_M, p$))
  return $P_L,\ P_R$
end function

Algorithm 8: Partition Strategy ii

function PartitionPoints(P, H_L, H_R, H_M)
  Swap($P[|P|]$, P[IndexOf(P, H_M)]) ▷ Temporarily moves H_M to the end.
  if $H_L < H_R$ **then**
    $P_{RX} \leftarrow$ Partition($P[1 \ldots |P|-1], p \mapsto p.x > H_M.x$) ▷ Upper hull
  else
    $P_{RX} \leftarrow$ Partition($P[1 \ldots |P|-1], p \mapsto p.x < H_M.x$) ▷ Lower hull
  end if
  $P_R \leftarrow$ Partition($P_{RX}, p \mapsto$ LeftOfLine($H_M \rightarrow H_R,\ p$))
  Swap($P[|P|]$, $P[|P_{RX}|+1]$) ▷ Moves H_M so that it's in between P_R and P_L
  $P_L \leftarrow$ Partition($P[|P_{RX}|+2 \ldots |P|],\ p \mapsto$ LeftOfLine($H_L \rightarrow H_M, p$))
  return $P_L,\ P_R$
end function

Comparing options i and ii, we can see that for both options, the first of partitioning involves all $|P|$ points, whereas the remaining partition operations involve $|P| - |P_L|$ points for option i and $|P|$ points for option ii. This means that option i may perform better due to not having to read as many points during its second partition operation. Option i, however, potentially requires more side of line comparisons than option ii. For option i the number of side of line comparisons is $|P_L|+2\cdot(|P_R|+|P_T|)$ compared to $|P_L|+|P_R|+|P_T|$ for option ii. Since side of line comparisons require two multiplications these could be more computationally expensive than the X-coordinate comparisons performed during the first partition of option ii. Option iii could perform better than i and ii because of only needing to traverse the list of points once, at the cost of more complicated branching logic and swapping both towards the start and end of the points array, instead of only towards the start.

Algorithm 9: Partition Strategy iii

```
1:  function PartitionPoints(P, H_L, H_R, H_M)
2:      Swap(P[|P|], P[IndexOf(P, H_M)])          ▷ Temporarily moves H_M to the end.
3:      R_Next ← 1
4:      L_Next ← |P| − 2
5:      i ← 1
6:      while i ≤ L_Next do
7:          if LeftOfLine(H_M → H_R, P[i]) then
8:              Swap(P[R_Next], P[i])
9:              R_Next ← R_Next + 1
10:             i ← i + 1
11:         else if LeftOfLine(H_L → H_M, P[i]) then
12:             Swap(P[L_Next], P[i])
13:             L_Next ← L_Next − 1
14:         else
15:             i ← i + 1
16:         end if
17:     end while
18:     Swap(P[|P|], P[L_Next + 1])          ▷ Moves H_M so that it's in between P_R and P_L
19:     P_L ← P[L_Next + 2 . . . |P|]
20:     P_R ← P[1 . . . R_Next − 1]
21:     return P_L, P_R
22: end function
```

Algorithm 10: Partition Strategy iv

```
1:  function PartitionPoints(P, H_L, H_R, H_M)    ▷ P has already been sorted by X-coordinate
2:      N_RX ← IndexOf(P, H_M)
3:      P_R ← Partition(P[1 . . . N_RX − 1], p ↦ LeftOfLine(H_M → H_R, p))
4:      P_L ← Partition(P[N_RX + 1 . . . |P|], p ↦ LeftOfLine(H_L → H_M, p))
5:      return P_L, P_R
6:  end function
```

Option iv could potentially perform better than the other two due to only having to iterate through the points once, at the cost of initially having to sort all points. On data sets where points are somewhat evenly distributed, such as on our *square* and *disk* data sets, a large amount of points are likely to be removed during the first few levels of QuickHull. On data sets such as these, sorting all points initially may not give any performance benefit because many of the points that are sorted may be removed very quickly and thus not contribute to the reduced number of memory accesses during later partitioning operations. Because of this we have also tested a hybrid approach where partitioning strategy i is used until a certain depth is reached, at which point the remaining points for that sub problem are sorted and deeper levels of recursion use partitioning strategy iv.

4.2 Removal of Points

In the pseudo code points that are determined to not be part of the convex hull are added to the set called $NotOnHull$ and finally removed at the end of the algorithm. In order to achieve $O(1)$ memory complexity, our implementations do not store this set and instead assign points a special value in order to represent that they are not part of the convex hull. For floating point implementations we use NaN and for integer implementations we use the smallest possible representable value.

Another approach would be to instead mark points that are determined to be on the convex hull, for example by appending these to a list that is passed as a parameter to the recursive function (see the pseudo code in algorithm 11). Most of our implementations use the previous approach (marking points that are not on the convex hull). Note that this approach does not allow for $O(1)$ memory complexity because of the list labeled $Result$ in algorithm 11. This approach is also more difficult to parallelize by running the two recursive calls to $QuickHullDAC$ in parallel. This is because both invocations of $QuickHullDAC$ need to append points to the list of results, and in order for these points to be ordered by their position in the convex hull all hull points found by the first recursive call need to be appended before points from the second recursive call.

Algorithm 11: Depth First QuickHull (marking points on the convex hull)

```
1:  function DepthFirstQuickHull(P_in)
2:      P_Hull ← []
3:      H_L ← the lexicographically smallest point in P_in
4:      H_R ← the lexicographically largest point in P_in
5:      P_in ← P_in \ {H_L, H_R}
6:      P_below ← Partition(P_in, p ↦ RightOfLine(H_L → H_R, p))
7:      Append H_L to the end of P_Hull
8:      DivideAndConquer(P_below, H_R, H_L, P_Hull)
9:      Append H_R to the end of P_Hull
10:     DivideAndConquer(P_in \ P_below, H_L, H_R, P_Hull)
11:     return P_Hull
12: end function
13: function DivideAndConquer(P, H_L, H_R, P_Hull)
14:     if P ≠ ∅ then
15:         H_M ← the point in P furthest from the line H_L → H_R
16:         P_L, P_R ← PartitionPoints(P, H_L, H_R, H_M)
17:         DivideAndConquer(P_L, H_L, H_M, P_Hull)
18:         Append H_M to the end of P_Hull
19:         DivideAndConquer(P_R, H_M, H_R, P_Hull)
20:     end if
21: end function
```

Algorithm 12: Depth First QuickHull

function DepthFirstQuickHull(P_{in})
  $H_L \leftarrow$ the lexicographically smallest point in P_{in}
  $H_R \leftarrow$ the lexicographically largest point in P_{in}
  Swap($P_{in}[1]$, $P_{in}[\text{IndexOf}(P_{in}, H_L)]$) ▷ Moves H_L to its correct location at the start of P
  $P_{below} \leftarrow \text{Partition}(P_{in}[2 \dots |P_{in}|], p \mapsto \text{RightOfLine}(H_L \to H_R, p))$
  Swap($P_{in}[|P_{below}| + 2]$, $P_{in}[\text{IndexOf}(P_{in}, H_R)]$) ▷ Moves H_R to its correct location at the first index among points above the line
  $P_{above} \leftarrow P_{in}[|P_{below}| + 3 \dots |P_{in}|]$
  DivideAndConquer(P_{below}, H_R, H_L)
  DivideAndConquer(P_{above}, H_L, H_R)
  Remove points from P that have been marked as not on the convex hull
  return P
end function
function DivideAndConquer(P, H_L, H_R)
  if $P \neq \emptyset$ **then**
    $H_M \leftarrow$ the point in P furthest from the line $H_L \to H_R$
    $P_L, P_R \leftarrow \text{PartitionPoints}(P, H_L, H_R, H_M)$
    Mark all points in $P \setminus (P_L \cup P_R \cup \{H_M\})$ as not on the convex hull
    DivideAndConquer(P_L, H_L, H_M)
    DivideAndConquer(P_R, H_M, H_R)
  end if
end function

4.3 Merged Distance & Orientation Tests

In the article "Quicker than QuickHull" [13] the authors present an approach for reducing the amount of computation needed by merging the orientation calculations performed in step 2 with the furthest point calculations performed at the next recursion depth in step 1. This is done by, when partitioning points into subsets P_L and P_R, keeping track of the point with the largest cross product among those partitioned into P_L and P_R respectively. These two points are the new hull points (H_M) at the next recursion depth, and can then be passed down instead of being computed in step 1.

4.4 Breadth First QuickHull

Our implementations can be largely divided into two groups depending on whether they process recursion in a depth-first manner or a breadth-first manner. The implementations discussed up until this point are depth-first implementations since they process subsets of points by immediately recursing to these subsets. Breadth-first implementations maintain two lists of intervals at the current and next depth respectively, with each interval encoding the parameters H_L, H_R and P described above. These implementations process all intervals at the current depth using the steps outlined above, but append subdivided intervals to the

list for the next depth in steps 3 and 4 instead of immediately processing these recursively. See the pseudo code below for more details.

Algorithm 13: Breadth First QuickHull

1: **function** BreadthFirstQuickHull(P_{in})
2: $H_L \leftarrow$ the lexicographically smallest point in P_{in}
3: $H_R \leftarrow$ the lexicographically largest point in P_{in}
4: Swap($P_{in}[1]$, $P_{in}[\text{IndexOf}(P_{in}, H_L)]$) ▷ Moves H_L to its correct location at the start of P
5: $P_{below} \leftarrow \text{Partition}(P_{in}[2 \ldots |P_{in}|], p \mapsto \text{RightOfLine}(H_L \rightarrow H_R, p))$
6: Swap($P_{in}[|P_{below}| + 2]$, $P_{in}[\text{IndexOf}(P_{in}, H_R)]$) ▷ Moves H_R to its correct location at the first index among points above the line
7: $P_{above} \leftarrow P_{in}[|P_{below}| + 3 \ldots |P_{in}|]$
8: $I_{Current} \leftarrow \emptyset$
9: **if** $P_{below} \neq \emptyset$ **then**
10: Add (P_{below}, H_R, H_L) to $I_{Current}$
11: **end if**
12: **if** $P_{above} \neq \emptyset$ **then**
13: Add (P_{above}, H_L, H_R) to $I_{Current}$
14: **end if**
15: **while** $I_{Current} \neq \emptyset$ **do**
16: $I_{Next} \leftarrow \emptyset$
17: **for all** (P, H_L, H_R) in $I_{Current}$ **do**
18: $M \leftarrow$ the point in P furthest from the line $L \rightarrow R$
19: $P_L, P_R \leftarrow \text{PartitionPoints}(P, H_L, H_R, H_M)$
20: Mark all points in $P \setminus (P_L \cup P_R \cup \{H_M\})$ as not on the convex hull
21: **if** $P_L \neq \emptyset$ **then**
22: Add (P_L, H_L, H_M) to I_{Next}
23: **end if**
24: **if** $P_R \neq \emptyset$ **then**
25: Add (P_R, H_M, H_R) to I_{Next}
26: **end if**
27: **end for**
28: $I_{Current} \leftarrow I_{Next}$
29: Optionally, remove points from P_{in} that have been marked as not on the convex hull and update bounds in $I_{Current}$
30: **end while**
31: Remove points from P_{in} that have been marked as not on the convex hull
32: **return** P_{in}
33: **end function**

4.4.1 Periodic Removal of Points

In our depth first implementations, points that are not on the hull are removed at the end by partitioning points that have not been set to the special value. In our breadth first implementations we have also tested whether it is beneficial to periodically remove points determined

to not be on the hull before advancing to the next depth (effectively at line 27 in algorithm 13). We have tested performing this removal of points at every iteration, when a certain number of iterations have passed since the last removal, or when the total amount of points that could be removed exceeds a certain threshold.

The supposed performance benefit of removing points periodically would come from better cache utilization during later iterations, since points that are marked as not being on the hull would otherwise fragment the memory regions that later iterations need to work on and potentially pollute cache lines at the edges of intervals of points that are still being considered.

4.5 Parallelization

We have investigated two approaches to parallelization of QuickHull. One for parallelizing depth first QuickHull, and one for parallelizing breadth first QuickHull.

The parallelization of depth first QuickHull works by simply running the two recursions in the divide and conquer phase (steps 3 and 4) in parallel.

This method of parallelization is very simple to implement, but may suffer from poor load balancing in certain situations, especially at low recursion depths. This is because at low recursion depths the input points will not have been divided into enough separate sub problems for many threads to be able to work in parallel. Unevenly distributed points could also cause poor load balancing if there are significantly more points on one side of a new hull point than the other, since this could mean that threads handling points on one side of the new hull point get significantly more work than threads on the other side.

The parallelization of breadth first QuickHull is more complicated, and loosely based on the method presented in the article *"Finding Convex Hulls Using QuickHull on the GPU"* by *Stanley Tzeng* and *John D. Owens* [24]. The algorithm makes use of established parallel algorithms for sorting and computing the prefix sum and prefix maximum of an array. Sorting can for example be performed in parallel using the "Bitonic Sort" algorithm originally introduced by Ken Batcher in 1968 [4]. Prefix sum and prefix maximum can be performed in parallel using an algorithm introduced by W. Daniel Hillis and Guy L. Steele in 1986 [12]. Our implementation uses the *Thread Building Blocks* library from Intel for these algorithms.

The initial phase of the algorithm is similar to the sequential breadth first QuickHull, but utilizing parallel algorithms for finding the lexicographically smallest and largest points as well as partitioning the points depending on whether they are above or below the initial line. During the initial stage, we also sort the points by X-coordinate using a parallel sorting algorithm. The points are sorted in ascending order below and descending order above, in the same way as described in partitioning strategy iv (on page 28).

Following the initial phase, the handling of intervals differs from the sequential algorithm by inverting the mapping between intervals to the points they contain. I.e. for each point, we track which interval contains that point, instead of for each interval tracking which set of points that interval contains. We will use $\boldsymbol{P}[i]$ to refer to the point at index i and $\boldsymbol{I}_{lo}[i]$, $\boldsymbol{I}_{hi}[i]$ to refer to the index of the first point and last point respectively in the interval that contains point i. As in the sequential algorithm, points within the same interval are stored sequentially in memory which means that $\boldsymbol{I}_{lo}$ and $\boldsymbol{I}_{hi}$ will be monotonic increasing and constant for points in the same interval.

Iterating through the list of active intervals (performed by the loop on line 17 in the sequential algorithm on page 32) is broken up into several stages, each of which is performed in parallel and completed fully before the next stage starts.

1. For each point, compute the distance to the line between the two hull points from the interval containing that point. Let $D[i]$ be the distance computed for point i.

2. Compute the prefix maximum of the tuple $(I_{lo}[i], D[i], i)$:

$$M[i] = \max\{(I_{lo}[1], D[1], 1), \ \ldots, \ (I_{lo}[i], D[i], i)\}$$

   Tuples are compared lexicographically which, due to the monotonic behavior of I_{lo}, means that the index of the point furthest from the line of the interval containing point i is given by the third element of $M[I_{hi}[i]]$. We will refer to this index as $P_{hull}[i]$.

3. For each point:

   (a) Compare the point's index i to $P_{hull}[i]$ to determine if the point is to the left or right of the new hull point for its interval. Since the points are initially sorted by their X-coordinate all points on the same side of the new hull point will also have indices on the same side of $P_{hull}[i]$.

   (b) Determine the point's orientation relative to the line between the new hull point and the left or the right hull point of the interval containing the point (depending on the point's side of the new hull point).

   (c) If the point is outside the line, either set $I_{hi}[i] \leftarrow P_{hull}[i] - 1$ or $I_{lo}[i] \leftarrow P_{hull}[i] + 1$ depending on the point's side of the new hull point.

   (d) If the point is not outside the line, set it to a special value marking it as not part of the convex hull.

4. (optionally) Remove points marked as not being on the convex hull.

Removal of points marked as not on the convex hull is performed at the end of the algorithm using a parallel stable partition operation. This can also be done periodically, in which case the indices in I_{lo} and I_{hi} also need to be updated. For I_{lo} the new values would be $I'_{lo}[i] = I_{lo}[i] - \sum_{j=1}^{i-1} remove(j)$ where $remove(j)$ is 1 if the point with index j will be removed and 0 otherwise (and similarly for I_{hi}). This update can be performed in parallel by computing the prefix sum $S[i] = \sum_{j=1}^{i-1} remove(j)$ in parallel.

Points that are determined to be on the convex hull stay in the points array, and have $I_{hi}[i] = I_{lo}[i] = i$ to mark that they are a hull point. These points, along with points marked as not being on the hull, we set $D[i] = -\inf$ in step 1 and don't perform any processing in step 3. Because of this, periodically removing points marked as not on the hull has the advantage of reducing the amount of wasted calculations in step 2 and parallelism in step 3, in addition to the potentially reduced memory fragmentation like in the sequential version.

Chapter 5

Worst case inputs for QuickHull

In this section we explore how to generate worst case inputs for QuickHull. The idea is to make QuickHull recurse as deep as possible, yielding a time complexity of $\Omega(nd)$ where d is the depth of the recursion, and we place $n - d$ points where the recursion ends.

One way to achieve deep recursion is explained in a stack overflow thread on the issue, `https://stackoverflow.com/a/14457421`, as of May 18, 2023. The presented approach by the user "John", posted January 22, 2013, is to put points on a circle and in every step one would bisect the current arc and place a point there. Formally one would generate a sequence of points $P = [P_0, P_1, \ldots]$ where $P_i = (1, \theta_i), \theta_i = \frac{\pi}{2^i}$ in polar coordinate form. For simplicity of analysis we will also add the point $P_{-1} = (1, 0)$ to P.

As visualized in Figure 5.1, QuickHull would for this input always have P_{-1} as H_R and at recursion depth i have P_i as H_L. In every recursive step P_{i+1} would maximize the distance to the line, and all remaining points would end up in the same recursive call.

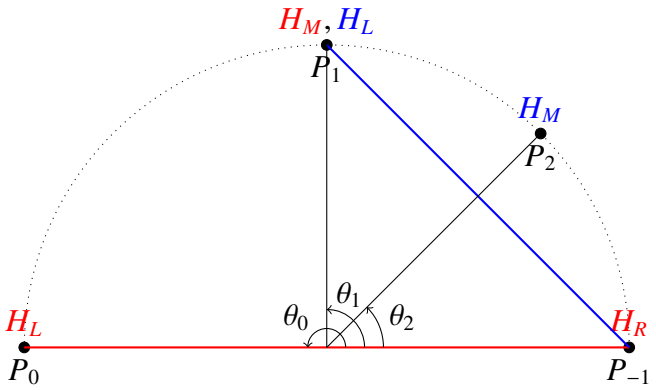


Figure 5.1: Illustration of worst case input for QuickHull as described by a stack overflow thread. The first recursive step is shown in red and the next in blue.

Assuming computers have perfect floating point arithmetic this approach will be sufficient to achieve any desired depth, thus meaning that QuickHull would run in $\Omega(n^2)$ time. Practically however, numbers on computers are represented with a maximal size and amount of precision, which means that there is a bound on how small coordinates we can represent.

We will assume that QuickHull is implemented using 64-bit double precision numbers, using 1 sign bit, 11 exponent bits and 52 precision bits according to the IEEE 754 standard [1, §3.6]. We also assume that the points are represented with Cartesian coordinates, and not polar coordinates. To our knowledge there is no CH algorithm doing computations exclusively in polar coordinates.

We will start by getting an upper bound on the depth of the recursion. After that we will improve the approach from the stack overflow thread. Finally we will address some problems by presenting a more numerically stable approach. Our analysis in this chapter easily extends to floating point numbers with lower or higher size and precision.

5.1 Upper Bound on Recursion Depth

Overmars and van Leeuven showed that the area spanned by a recursive call C in QuickHull is at least 2 times the sum of the areas spanned by the two calls that C recurses into. They used this to prove that QuickHull has $O(n)$ expected time complexity on inputs with points selected uniformly at random from a convex region. [18]

The argument that areas are divided by 2 was later used by Gamby and Katjainen to put an upper bound on the recursion depth of QuickHull. This was done for integer coordinate inputs. [9]

We begin by improving Overmars and van Leeuvens bound to a factor of 4 instead of 2, and will then use this to show an upper bound on the recursion depth of QuickHull for floating point inputs.

Note that whenever QuickHull identifies a vertex v of the convex hull, it is because v was the maximal point in some direction, like maximal/minimal x coordinate or maximum distance to some line. Consider figure 5.2. It shows a recursive call on QuickHull for points H_L and H_R. We will also associate an imaginary point Q with the call, which is placed so that H_L was maximal in the direction perpendicular to H_LQ and H_R was maximal in the direction perpendicular to H_RQ. This means that all points P sent to the recursive call are inside the triangle $T = \triangle H_LH_RQ$. QuickHull finds the point $H_M \in P$ with maximal distance to H_LH_R, and recurses into T_L and T_R as seen in the figure.

If the height from Q to Q_LQ_R is some proportion x of the total height h from Q to H_LH_R we see that

$$\text{Area}(T_L) + \text{Area}(T_R) = \frac{|Q_LQ_R|(1-x)\cdot h}{2} = \frac{x|H_LH_R|(1-x)\cdot h}{2} = x(1-x)\cdot\text{Area}(T)$$

Since $x(1-x)$ has maximal value $\frac{1}{4}$ when $x = \frac{1}{2}$, we can conclude that the area of the two recursive calls is at most $\frac{1}{4}$ of the area of the original call. This also means that the largest area of the two recursive calls is at most $\frac{1}{4}$ of the original call.

We also have that $\text{Area}(T) \geq \text{Area}(T_M) \geq \text{Area}(T_L) + \text{Area}(T_R)$, which means that we can bound the areas of T_M similarly. If $A_d = \text{Area}(T_M)$ in a recursive call at depth d, we have that $A_0 \geq 4^{d-1}A_d$

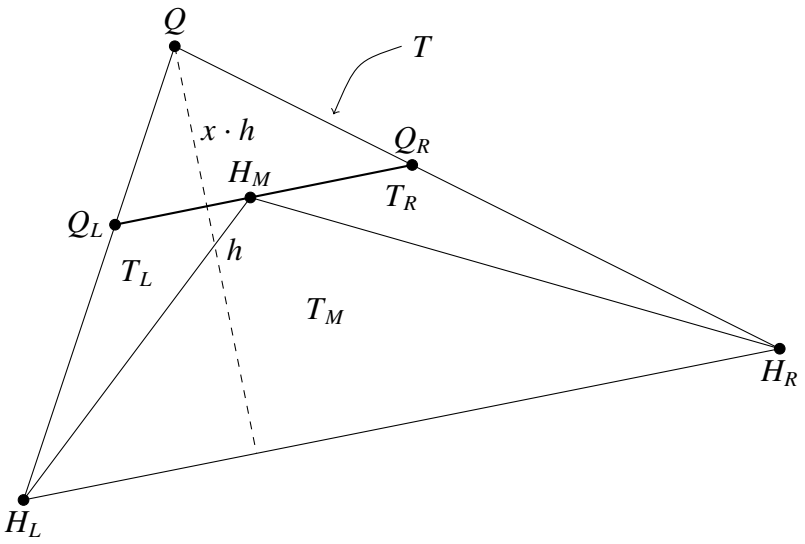


Figure 5.2: Illustration of QuickHull recursion step. The call for H_L, H_R contains points in $T = \triangle H_L H_R Q$. QuickHull finds the points H_M with maximal distance to $H_L H_R$, and recurses into T_L and T_R. The height from Q to $H_L H_R$ is h, and the height from Q to $Q_L Q_R$ is $x \cdot h$

Since both A_0 and A_d are areas that are computed by QuickHull when a point H_M is identified both of these numbers need to be representable floating points numbers, we see that $4^{d-1} \leq \frac{\text{MAX DOUBLE}}{\text{MIN DOUBLE}} = 2^{2^{11}} = 4^{2^{10}}$ meaning $d \leq 2^{10} + 1 = 1025$. Generally, if the floating point architecture has b exponent bits, our upper bound is $2^{b-1} + 1$.

5.2 Improving Achievable Depth

The approach from Stack Overflow clearly cannot achieve a very deep recursion because the points will be of the form $(\cos(\theta), \sin(\theta))$ in cartesian coordinates for very small θ. This converges towards $(1, 0)$. At around depth 27 when $\theta_{27} = 2^{-27}$, the x coordinate $\cos(\theta)) \approx 1 - \frac{\theta^2}{2}$ will be badly affected by precision and indistinguishable from 1 since doubles have 53 bits of precision.

We will start by fixing this issue, and then improve the depth of the recursion by scaling the input and splitting the arc using a golden ratio trick instead of in 2 equal parts.

5.2.1 Shifting and Scaling

The issue that the x-coordinate of P_i as affected by precision issues can be solved by shifting the circle left so that it is centered around $(-1, 0)$ and thus the points generated will converge towards $(0, 0)$. This means that as the points get closer to each other their absolute values also decrease, meaning that doubles can still differentiate between them.

If the program generating the input also has bounds on precision, one would not be able to accurately calculate the x-coordinate naively on the format $\cos(\theta) - 1$. The x-coordinate

can instead, for small values of θ, be replaced with the second term in its Taylor series giving the expression $1 - \frac{\theta^2}{2} - 1 = -\frac{\theta^2}{2}$.

We can now scale everything with a radius $r = \sqrt{\text{MAX DOUBLE}}$ of the circle so that the area of the initial triangle is close to MAX DOUBLE, and generate points until the area is close to MIN DOUBLE. This would yield an input that causes QuickHull to recurse to a depth of $d = 678$.

One way to put this approach into context with the upper bound of 2^{10} is that the area of $\triangle P_{-1}P_{i-1}P_i \approx \theta_i^3$, so the area is reduced by a factor of 8 for each recursion, which means that we are achieving $\log_8 4 = \frac{2}{3}$ of the upper bound on depth.

5.2.2 Golden Ratio Trick

We can improve the depth by noting that if we were to put P_{i+1} slightly off center, we can recurse into the larger part as long as we avoid placing any points in the area that would be further from P_iP_{-1} than P_{i+1} is. By doing this we do not split the arc in half each time, because we always use a little bit more than half the arc length for subsequent calls.

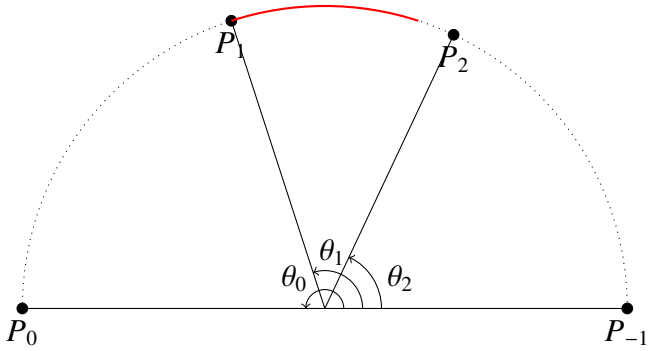


Figure 5.3: Illustration of not splitting the arc evenly. The red curve marks where we don't allow any points to be placed since P_1 should be the furthest point from P_0P_{-1}. Importantly $\theta_2 < \theta_0 - \theta_1$ means that P_2 is outside the red region.

We can find the limit of this strategy by assuming that $\theta_i = \pi x^i$ for some factor x that we want to maximize. We now see that $\theta_{i+1} = x\theta_i$ and $\theta_{i+2} = x^2\theta_i$. P_{i+1} will be further from P_iP_{-1} than P_{i+2} is if $\theta_{i+2} < \theta_i - \theta_{i+1} \implies x^2 < 1 - x \implies x < \frac{1}{\phi} \approx 0.61803$ if we only consider positive x. This means that in the limit of this construction we are dividing the angle with the golden ratio in every step.

Rounding down to $x = 0.618$ we have a little bit of margin which means that precision will not be an issue when comparing the distances of P_{i+1} and P_{i+2} to P_iP_{-1}. Using this factor we were able to generate an input with $d = 976$. We also see that since $x^3 \approx 0.236$ we are quite close to dividing the area by 4 in every step. More exactly we are achieving $-\log_{x^3} 4 \approx 0.96$ of the upper bound on depth.

5.2.3 Numerical Issues

The two described approaches yielding depth 670 and 978 does have some numerical issues. To illustrate, if we have an input without scaling or the Golden ratio trick, QuickHull will begin by identifying $P_0 = (-2, 0)$ and $P_{-1} = (0, 0)$ as the leftmost and rightmost points. It will then find $P_1 = (-1, 1)$ as the point maximizing the distance to P_0P_{-1}. For the next recursive call QuickHull needs to know which side of the line P_1P_{-1} each remaining point is on. If $P_i = (-\theta_i^2, \theta_i)$ then SideOfLine(P_{-1}, P_1, P_i) will compute $S = -1 \cdot \theta_i - 1 \cdot (-\theta_i^2) = \theta_i^2 - \theta_i$, whilst SideOfLine$(P_1, P_i, P_{-1})$ will compute $S = (1 - \theta_i^2) \cdot -1 - (\theta_i - 1) \cdot 1 = \theta_i^2 - \theta_i$. For a small θ_i, the first calculation still works fine whilst the second one will not due to $1 - \theta_i^2$ being very close to 1. This means that the answer that QuickHull generates and how deep it recurses depends on internal implementation details and rounding errors. Specifically it matters which permutation of points that QuickHull calls Side Of Line with.

Generally we can think of this issue as being caused by the fact that the line from P_1 to P_{-1} is very close to some point P_i compared to the distance between P_1 and P_{-1}. Avoiding this situation for any three points in the input is not possible because we will need to have some points very close to each other and others much further away, but if we are careful we can construct the input so that QuickHull doesn't look at these combinations of points.

5.3 Numerically Stable Approach

To create an input that does not have the issues mentioned in section 5.2.3 we will instead of always recursing into the right arc, alternatively recurse into the left and right arcs. This way we do not run into this issue because as seen in figure 5.4 each line P_iP_{i+1} is relatively far from the origin where the points are converging to.

We still want the points to converge towards the origin, so to do this we will imagine a unit circle with center in $(0, -1)$ and angles measured from the y-axis. We will now generate points on the following format: $P = [P_0, P_1, \ldots]$ where P_i is on the circle with angle $\theta_i = \frac{\pi}{2}(-x)^i$ measured from the y-axis. This way the points will converge towards the top of the circle.

Clearly $x = 0.5$ works, because every P_{i+2} will bisect the arc of P_iP_{i+1} since $\theta_{i+2} = \frac{\theta_i + \theta_{i+1}}{2}$.

However, as for the golden ratio trick we would like x to be as large as possible while not changing how QuickHull recurses. We can start by noting that P_i will alternatively be on the left and right side of the origin. If we assume P_i is to the left (meaning $\theta_i > 0$) we see that P_{i+2} needs to be further from P_iP_{i+1} than all remaining points. Moving clockwise from P_{i+2} the next point will be P_{i+4} giving the sufficient condition that P_{i+2} is further from P_iP_{i+1} than P_{i+4} is. Algebraically this becomes $\theta_i - \theta_{i+2} > \theta_{i+4} - \theta_{i+1} \implies 1 - x^2 > x^4 + x \implies x < 0.56984\ldots$ if we only consider positive x.

We will round down to $x = 0.5698$ giving a little bit of margin for precision in the computations. Using this factor (and again scaling the circle to radius $\sqrt{\text{MAX DOUBLE}}$ we were able to generate an input with $d = 834$. We also see that $x^3 \approx 0.185$ so the approach is not as close to dividing the area by 4 in each step as the golden ratio trick is. More exactly it achieves $-\log_{x^3} 4 \approx 0.82$ of the upper bound on depth.

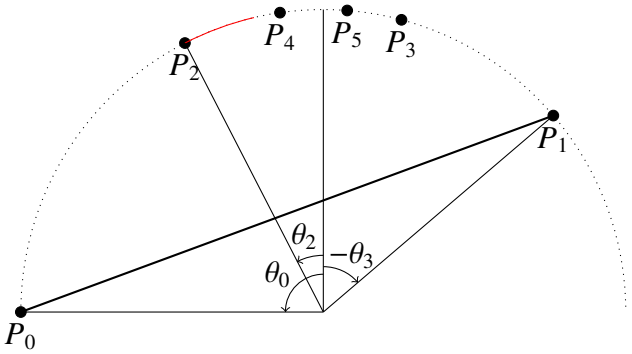


Figure 5.4: Illustration of numerically stable approach to worst case QuickHull inputs. The line P_0P_1 that QuickHull will start by considering is not very close to any points. The red curve marks where we don't allow any points to be placed since P_2 should be the furthest point from P_0P_1. Importantly $\theta_0 - \theta_2 > \theta_4 - \theta_1$ means that P_4 is outside the red region.

Chapter 6
Evaluation

In this chapter we will empirically evaluate our findings from chapters 3, 4, and 5. We cover how the experiments have been performed in section 6.1, and the different kinds of inputs we are using in section 6.2. In section 6.3 we compare different variants of the QuickHull algorithm, and in section 6.4 we consider how Refined Chan compares to Chan's algorithm and MergeHull. Finally in section 6.5 we compare our best implementations with library code and answer the question of which convex hull algorithm is the fastest in different circumstances.

6.1 Experimental Setup

In order to evaluate the performance of planar convex hull algorithms we created our own framework for testing and evaluating. All algorithms were implemented in C++ and compiled with the GCC compiler version **11.2.0** and **12.2.1**. The implementations were thoroughly tested to spot potential bugs.

All implementations use 64-bit double-precision floating-point numbers to represent coordinates, and points are stored in Array-of-Structure layout, i.e. with X and Y coordinates interleaved.

Our framework passes data by reference to the algorithms, meaning that they could be implemented in-place by modifying the input array to return the output. In general all algorithms were implemented with as little copying as possible, making temporary arrays in the cases that the algorithms couldn't be implemented completely in-place.

Performance was measured on two different systems, the first of which has an Intel Core i7-1065G7 processor, and the second of which is a POWER8 TN71-BP012 computer with 10 cores and 80 hardware threads [22].

Our framework uses std::chrono::high_resolution_clock to accurately measure time. Before running an algorithm we read the input into memory as a continuous array, and we define computation time as the time it takes for the algorithm to return the result after running it. The loading time is excluded to remove noise from measurements. All of our im-

plementations, as well as CGAL use this framework for measuring time. For Qhull, we used the measurements that are output by the qhull program labeled as "CPU seconds to compute hull (after input)".

In addition to measuring time we also measured memory usage and instruction counts. On the Intel computer this was done with Intel's "Performance Counter Monitor" (PCM) library, using the following counters:

- **Instructions Retired**: This counter measures the total number of instructions completed.
- **M.C. Read** and **M.C. Written**: These counters measure memory traffic between the L3 cache and DRAM. This means that a high number for this measurement could be because the implementation performs many memory accesses, or because it has poor cache performance.
- **L3 Hits/Misses**: This counter measures the total number of L3 cache hits and misses.
- **Bad Speculation**: This measures the fraction of pipeline slots wasted due to incorrect speculations, which may be caused by a high amount of mispredicted branches.

The source code for our implementations can be found in our Github repository: `https://github.com/Eae02/convex-hull-exjobb/tree/main/src/implementations`

6.2 Data Sets

The algorithms were evaluated on different types of inputs. We believe that these inputs should capture both the variety of inputs that one might want to use a planar convex hull algorithm for, and also capture how the performance of convex hull algorithms vary with different types of inputs.

6.2.1 Randomly Generated Data Sets

We have four sets of randomly generated data. The data sets were generated with 1 million input points for all tables presented in this chapter. These data sets have also been used to plot graphs, where they vary in size from 100 to 10 million input points.

Circle: This data set consists of points placed uniformly at random on the circumference of a circle. For n input points we also expect n output points, perhaps with the loss of a few points that are excluded due to precision limitations. This input could translate to a user running CH on an input that already is or almost is a convex hull.

Square: This data set consists of points placed uniformly at random within a randomly rotated square. The reason that we rotate the square is so that algorithms that use points with maximal/minimal x/y coordinates do not gain an unfair advantage/disadvantage. For n input points we expect $h \approx \frac{8}{3} \log_e n$ [6] output points. This input could translate to a use case where the number of hull points is quite low. An example is seen in figure 6.1.

Disk: This data set consists of points placed uniformly at random within a circle. For n input points we expect $h \in O(n^{\frac{1}{3}})$ [6] output points.

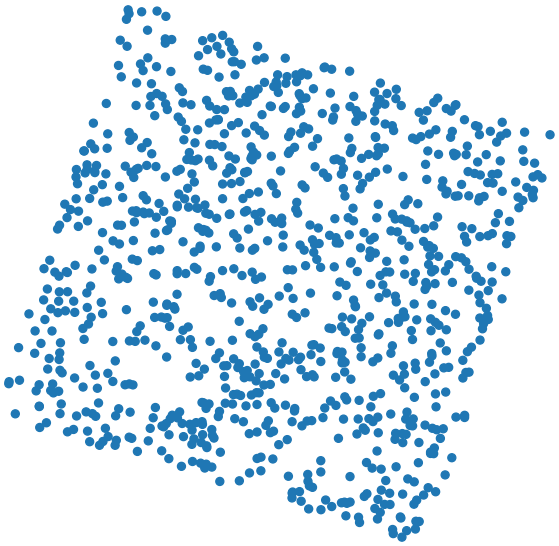


Figure 6.1: A square data set with 1000 random points.

MergeHull killer: This data set consists of the circle data set surrounded by a triangle. This data set is handcrafted to penalize the MergeHull algorithm, because MergeHull will run in roughly $O(n \log n)$ whilst h is very small. An example is seen in figure 6.2.

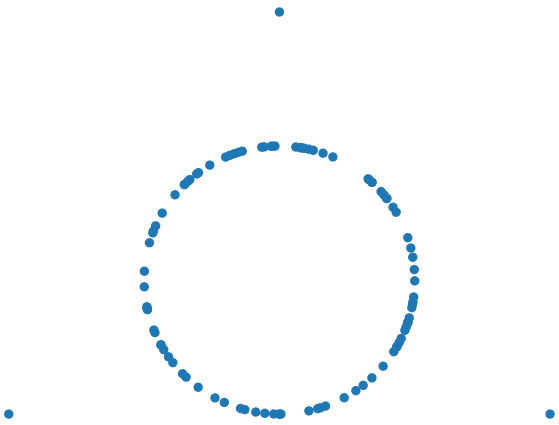


Figure 6.2: A MergeHull killer input with 100 points, consisting of a circle surrounded by a triangle.

6.2.2 Medical Data Set

This data set is based on a medical research article [21] where convex hull computations were used to evaluate the similarity of different patient data sets.

Our data set was generated by reproducing the steps taken in the study and saving the convex hull computations that they perform. Each input point represents a combination of two different medical measurements in an ICU patient. An example is seen in figure 6.3, but

other inputs contain other combinations of measurements. The inputs are generated from two different clinical data sets of different sizes, which leads to us having two medical data sets. The first data set consists of 1000 inputs having $n = 11839$ input points each and an average of $h \approx 9.73$ hull points per input. The second data set consists of 1000 inputs having $n = 13037$ input points each and an average of $h \approx 9.94$ hull points per input. We call these data sets Medical A and Medical B respectively.

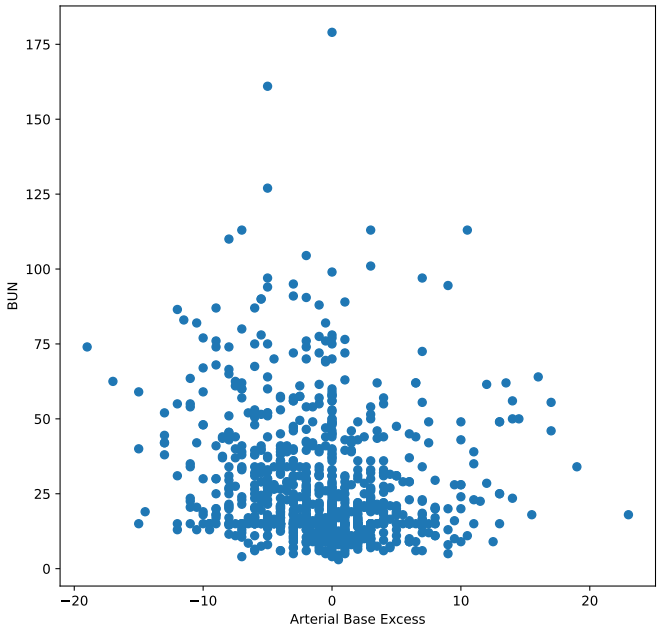


Figure 6.3: One of the inputs in the medical data set, showing only 1000 of 11839 points.

Due to patient data protection rules we are not allowed to share the actual data sets, but we have recorded the steps necessary to reproduce them in our GitHub repo.

6.2.3 QuickHull Killer

This data set contains one input, generated from the construction in chapter 5. It contains 1 million input points and has 836 hull points. It causes QuickHull to recurse 834 times, a visualization of the input is seen in figure 6.4.

6.3 QuickHull

In order to better understand how QuickHull performs on different test cases, we have analyzed how many points remain at different recursion depths, as well as what percentage of the execution time is spent at different depths. Execution time at different recursion depths was measured with the Linux `perf` command, with a sample rate of 59999 Hz. For both measurements QuickHull with partitioning strategy iii was run on the 10 large test cases in each of the randomized data sets.

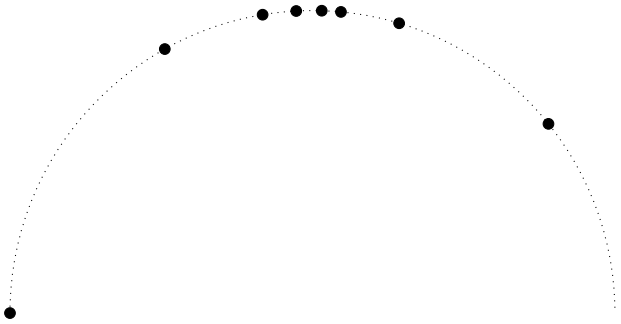


Figure 6.4: The first 8 points generated in QuickHull killer input. The points converge to the origin at the top of the circle, alternating between the left and right halves of the arc.

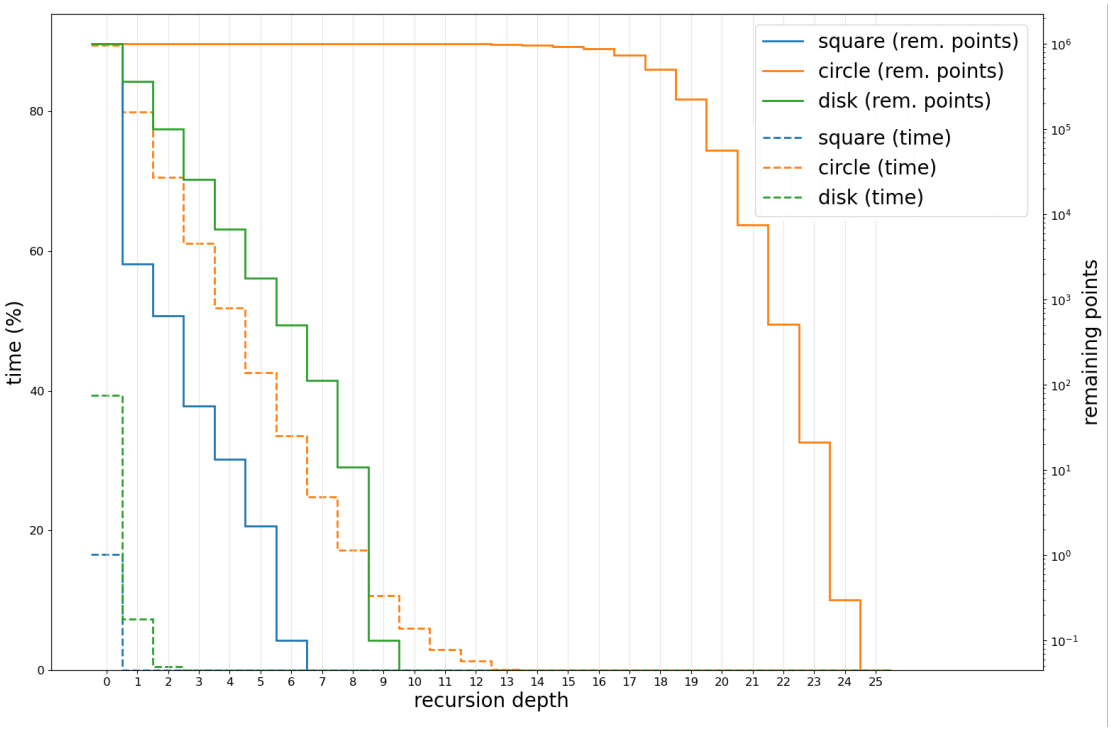


Figure 6.5: Points remaining and percentage of execution time at different recursion depths.

Figure 6.5 shows the results of these measurements. Solid lines show the number of points remaining, and dashed lines show the percentage of the program's execution time that is spent at that depth or greater. Depth 0 refers to the first recursion depth in the divide and conquer phase of the algorithm, and does not include time spent finding the first two hull points and partitioning points according to the line between them.

From the figure we can see that the number of points remaining, as well as the time percentage, falls very quickly as recursion depth increases on the square and disk data sets. For our data, there were no cases where QuickHull reached a depth greater than **6** on square and **9** on disk. On circle, however, the number of points remaining does not fall by more than **10%** until depth **16**, after which it falls more rapidly. Given that a QuickHull which always splits points in half during the divide and conquer stage would reach roughly depth

$\log_2 10^6 \approx 20$, it makes sense that the major decrease in points remaining happens at depths 16 - 24.

Since the recursion is much deeper on circle cases compared to the disk and square cases, the divide and conquer phase is expected to take more time in relation to the initial phase. Indeed, on the circle cases depth 0 (the entire divide and conquer phase) takes up 89.50% of total execution time, whereas the same measurement for square and disk is 16.59% and 39.36% respectively. Because of this, prepossessing which aims to improve performance during the divide and conquer phase by spending more time in the initial phase may be more effective on the circle cases.

6.3.1 Partition Strategies

These results were collected by running our implementation of depth first QuickHull on the Intel computer, using the different partitioning strategies listed in section 4.1 on the 10 large test cases in each group. For each of the 10 test cases, the implementation was run 5 times with the process being completely restarted in between runs. The values presented are the average of the 50 runs for that data set. For the hybrid strategy, points were sorted at depth 3. "Q.T.Q." refers to the optimization presented in the paper "Quicker than QuickHull" that finds the furthest point for the next recursion depth at the same time as partitioning points.

Table 6.1: Results on square and disk data sets.

Data	Partitioning Strategy	Compute Time (ms)	Instructions Retired	M.C. Bytes Read	M.C. Bytes Written
Square	i (no X-partition)	14.55	1.06×10^8	9.77×10^7	3.02×10^7
	ii (partition by X)	16.78	1.06×10^8	1.22×10^8	4.85×10^7
	iii (single scan)	13.54	1.01×10^8	8.66×10^7	2.87×10^7
	iv (sort initially)	88.18	2.59×10^8	1.73×10^8	1.15×10^8
	Hybrid i and iv	14.56	1.08×10^8	7.88×10^7	3.02×10^7
	Q.T.Q.	12.65	1.01×10^8	5.70×10^7	2.99×10^7
Disk	i (no X-partition)	21.12	1.32×10^8	8.72×10^7	4.22×10^7
	ii (partition by X)	26.53	1.30×10^8	8.81×10^7	4.61×10^7
	iii (single scan)	20.99	1.30×10^8	7.47×10^7	3.34×10^7
	iv (sort initially)	94.68	2.74×10^8	1.64×10^8	1.06×10^8
	Hybrid i and iv	22.87	1.40×10^8	1.02×10^8	5.01×10^7
	Q.T.Q.	20.51	1.17×10^8	6.31×10^7	3.40×10^7

On the square and disk data sets we can see that the version with the "Quicker than QuickHull"-optimization performs the best, while versions without the optimization using partitioning strategies i and iii are only somewhat slower. The implementation labeled "Q.T.Q." uses a partitioning strategy similar to that of iii, and could be implemented with other partitioning strategies as well. We chose to evaluate this optimization with strategy iii since this performed best in cases without the optimization.

In section 4.1 we discussed the difference in theoretical number of memory accesses needed by approaches i, ii, and iii. The memory traffic statistics appear to align with this, since ii shows heavier memory traffic than i while iii is slightly lighter. This may explain the

difference in performance between options i and ii. The number of instructions retired is also slightly higher for option ii than i, which also seems reasonable since option ii performs more iterations in order to potentially reduce the amount of more expensive instructions performed by the line orientation tests, but the instructions retired statistic does not account for different instructions being more expensive than others.

As for option iv (initially sorting all points), we can see that the number of instructions retired and the memory traffic is significantly higher than any of the other options. As discussed in section 4.1, on data sets where many points can be removed at the first few levels it could be the case that sorting all points initially requires much more work than what is saved by not needing to rearrange points across the new hull point during the divide and conquer stage. This also seems to align with our experimental results, since it appears that option iv is performing significantly more work than the other options. Also note that the hybrid approach performs similarly to strategies i, ii, and iii, which also suggests that the large amount of points removed during early stages is related to sorting initially performing significantly worse.

Table 6.2: Results on circle data set.

Partitioning Strategy	Compute Time (ms)	Instructions Retired	M.C. Read (MiB)	M.C. Written (MiB)
i (no X-partition)	180.48	9.26×10^8	1.61×10^8	8.63×10^7
ii (partition by X)	157.48	1.01×10^9	1.84×10^8	1.01×10^8
iii (single scan)	198.86	1.02×10^9	1.66×10^8	9.26×10^7
iv (sort initially)	139.43	8.83×10^8	1.65×10^8	7.89×10^7
Hybrid i and iv	148.53	8.97×10^8	1.35×10^8	7.26×10^7
Q.T.Q.	210.66	8.71×10^8	1.47×10^8	9.32×10^7

From table 6.2 we can see that the behavior on the circle data set differs significantly from that of the square and disk data sets. Here, initially sorting all points performs better than any other option. This could be because, as described in the previous section, the greater recursion depth means that preprocessing which reduces time spent in the divide and conquer phase is more worthwhile on circle cases.

The reason why sorting initially did not perform well on square and disk cases could be because a large number of points are filtered out at low recursion depths, meaning that a significant amount of work would be spent sorting points that are quickly removed. The idea behind the hybrid approach that sorts at a certain recursion depth was to improve the performance of the sorting-based approach on disk and square cases, by sorting points after a significant number of them have been removed. While this hybrid approach performs significantly better than sorting all points initially, it did not perform as well as not sorting at all. Figure 6.6 shows execution time as a function of which depth to perform sorting at. From this figure we can see that on circle cases it is best to sort as early as possible, whereas on disk and square sorting should be done late, although the only significant improvement in performance appears between depths 0 to 3.

Another interesting result for the circle cases is that partitioning strategy ii (partitioning by X) performs better than i and iii while the opposite was true for disk and square. We can also see that the "Quicker than QuickHull" optimization makes QuickHull perform worse on circle cases. Strategy ii performing better than i and iii could mean that reducing the

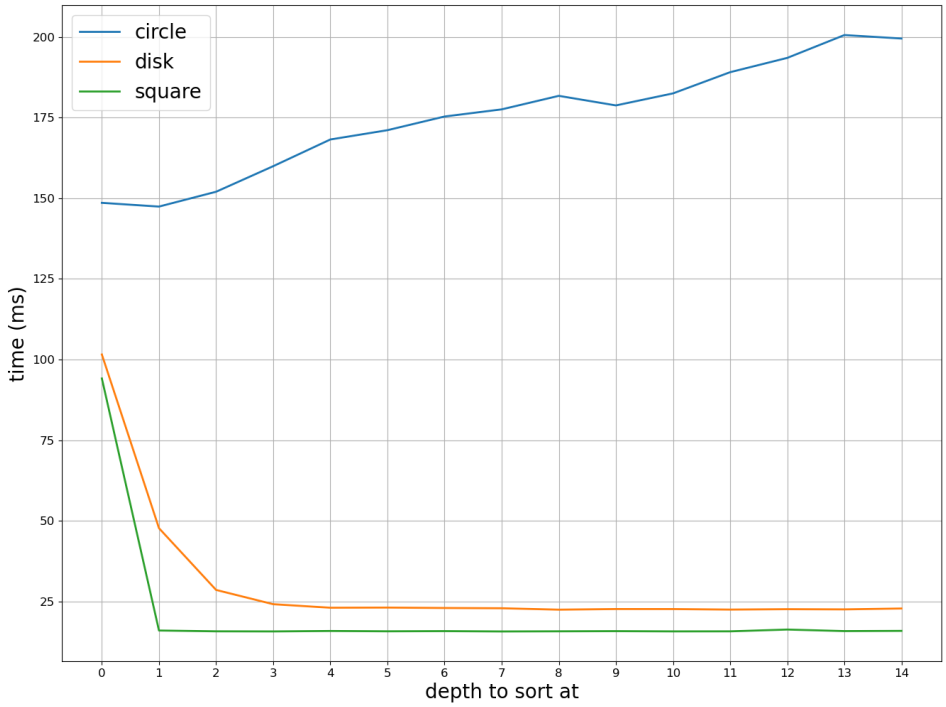


Figure 6.6: Performance of hybrid between strategies i and iv as a function of the depth at which to sort.

number of side-of-line tests at the cost of performing more iterations through the points is worthwhile on circle cases but not on disk and square.

Some of the performance behavior of these implementations may be related to differences in how branches are laid out. For Q.T.Q. and partitioning strategy iii, the combination of multiple passes through the points into one comes at the cost of more complicated and nested branching, whereas strategy ii performs more passes through the data but with less complicated branching. Table 6.3 shows the fraction of pipeline slots wasted due to incorrect speculation for different data sets and implementations of QuickHull. We can see from the table that the Q.T.Q. optimization and partitioning strategy iii have a generally higher rate of wasted pipeline slots on all data sets compared to partitioning strategy i. Partitioning strategy ii (partitioning by X) has a relatively high rate of wasted pipeline slots on square and disk while having a relatively low rate on circle, which matches the performance results.

Also note that the average fraction of wasted pipeline slots increases from square to disk to circle, especially when sorting based implementations (iv and hybrid) are not included in the average. This would suggest that poor speculation increases as the recursion depth increases and the divide and conquer phase consumes a larger percentage of the total execution time. The results for sorting based implementations could then be explained by these causing improved pipeline speculation in the divide and conquer phase at the cost of more work, and poor speculation in the initial phase. Furthermore, the poor performance of the Q.T.Q. optimization and partitioning strategy iii on circle cases could in part be because of

bad speculation being high enough to affect execution time only on circle cases.

Table 6.3: Fraction of pipeline slots wasted due to incorrect speculation (measured on Intel computer).

Implementation	Square	Disk	Circle
i (no X-partition)	0.259	0.386	0.485
ii (partition by X)	0.315	0.457	0.402
iii (single scan)	0.276	0.424	0.536
iv (sort initially)	0.460	0.479	0.355
Hybrid i and iv	0.263	0.380	0.384
Q.T.Q.	0.354	0.452	0.495
Average	0.321	0.430	0.443
Average (excl. sort)	0.301	0.430	0.479

We have also performed similar performance measurements on the POWER computer, the results of which are presented in table 6.4. The relative performance of different partitioning strategies is mostly the same on POWER as on the Intel computer, with two main differences. The first being that the Q.T.Q. optimization appears to not perform very well on POWER, and the second difference being that partition strategy ii performs worse than i on the circle cases, which is opposite from on the Intel computer.

Table 6.4: Compute times for depth first QuickHull on POWER

Implementation	Medical data A (ms)	Medical data B (ms)	Square (ms)	Disk (ms)	Circle (ms)
i (no X-partition)	12.29	12.54	33.93	48.79	417.00
ii (partition by X)	12.35	12.61	36.55	62.85	430.04
iii (single scan)	12.29	12.53	33.01	49.08	454.20
iv (sort initially)	13.23	13.57	192.50	206.84	348.89
Hybrid i and iv	12.31	12.56	35.05	52.48	366.04
Q.T.Q.	—	—	73.73	83.04	466.85

6.3.2 Breadth First QuickHull

Table 6.5 shows results for different partitioning strategies used with breadth first QuickHull. These measurements were made on the Intel computer in a similar way to the measurements in the previous section. For all measurements in this table a breadth first QuickHull which compacts points by removing those determined to not be on the convex hull was used. The results are generally similar to that of depth first QuickHull in terms of differences in memory traffic and compute times.

Table 6.6 shows results for different ways of compacting points by removing those marked as not on the convex hull (as described in section 4.4.1). The compaction strategy “Always” means that at the end of every iteration (before the algorithm proceeds to the next depth) points that are determined not to be on the hull are removed. “At end” means that points

Table 6.5: Partitioning results for breadth first QuickHull

Data	Partitioning Strategy	Compute Time (ms)	Instructions Retired	M.C. Read (MiB)	M.C. Written (MiB)
Square	i (no X-partition)	15.03	8.74×10^7	75.97	18.51
	ii (partition by X)	16.86	8.58×10^7	82.02	30.59
	iii (single scan)	14.58	8.77×10^7	70.38	19.12
Disk	i (no X-partition)	22.36	1.08×10^8	92.07	42.56
	ii (partition by X)	27.95	1.05×10^8	93.43	50.30
	iii (single scan)	22.99	1.09×10^8	83.44	38.05
Circle	i (no X-partition)	219.42	1.05×10^9	478.33	349.94
	ii (partition by X)	190.54	1.09×10^9	479.25	339.93
	iii (single scan)	245.28	1.13×10^9	471.71	364.84

Table 6.6: Results for different compaction strategies for breadth first QuickHull

	Compaction Strategy	Compute Time (ms)	Instructions Retired	M.C. Read (MiB)	M.C. Written (MiB)	L3 Hits	L3 Misses
Square	Always	15.27	8.66×10^7	82.90	21.54	1.40×10^4	1.12×10^4
	At end	16.71	9.62×10^7	92.46	31.78	1.24×10^4	2.15×10^4
	Depth > 1	16.76	9.93×10^7	96.17	32.46	1.38×10^4	2.86×10^4
	Marked > 10	16.80	1.11×10^8	101.09	32.40	3.63×10^4	3.74×10^4
Disk	Always	22.47	1.07×10^8	92.62	41.80	1.49×10^4	1.27×10^4
	At end	23.88	1.19×10^8	108.17	50.82	1.68×10^4	2.72×10^4
	Depth > 1	23.37	1.13×10^8	99.58	46.08	1.39×10^4	2.07×10^4
	Marked > 10	24.02	1.18×10^8	107.35	51.25	1.76×10^4	2.30×10^4
Circle	Always	223.04	1.05×10^9	524.71	379.33	1.61×10^5	1.87×10^5
	At end	218.34	1.02×10^9	538.79	362.92	1.52×10^5	1.86×10^5
	Depth > 1	224.56	1.05×10^9	514.19	369.88	1.62×10^5	1.87×10^5
	Marked > 10	221.19	1.04×10^9	557.83	374.33	1.68×10^5	2.15×10^5

determined not to be on the convex hull stay in memory until the end of the algorithm where they are filtered out in the same way as in the depth first implementations. "Depth > 1" and "Marked > 10" mean that points are filtered out if the respective condition is satisfied.

The results show that on the data sets square and disk, where many points can be removed, it is beneficial to remove points at every iteration. Comparing the memory traffic and L3 hit/miss measurements between "Always" and "At end" on these two data sets, the L3 hits are lower while memory traffic and L3 misses are higher for the "At end" strategy. This would suggest that always removing points that are determined to not be on the hull leads to better cache utilization, possibly because ranges of memory being worked on in later iterations then becomes less fragmented.

On the circle data set it will never be possible to remove any points. Therefore the processing that is needed when attempting to remove points, such as for updating interval indices, is entirely wasted.

From testing various thresholds for removal of points based on depth and number of

points marked, it appears that removing points periodically depending on these conditions is not beneficial on any of the test cases we have used. Instead it is better to remove points at every iteration, or only remove points at the very end. Figure 6.7 shows calculation time for different depth thresholds (on the left) and different thresholds for the number of points marked as not on the hull (on the right). For the square and disk data sets the performance does not change depending on the threshold, except for when the threshold is 0 which corresponds to removing points at every iteration. For circle cases, however, the performance generally becomes worse as the threshold for removing points is increased.

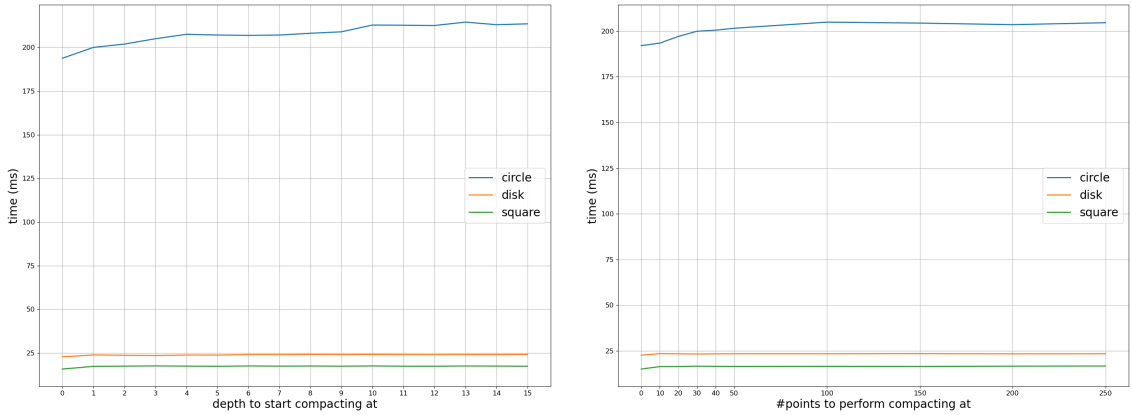


Figure 6.7: Calculation time for breadth first QuickHull as a function of thresholds at which to start compacting points.

The performance of different partitioning strategies and compaction was also measured on POWER, the results of which can be seen in table 6.4, with similar results as on the Intel computer.

Table 6.7: Compute times for breadth first QuickHull on POWER

Compaction	Implementation	Square (ms)	Disk (ms)	Circle (ms)
Always Compact	i (no X-partition)	31.52	46.77	471.28
	ii (partition by X)	34.32	60.96	466.52
	iii (single scan)	30.47	47.38	508.30
Compact at End	i (no X-partition)	35.12	50.07	443.31
	ii (partition by X)	38.14	64.11	455.89
	iii (single scan)	33.58	49.67	472.03

6.3.3 Fastest QuickHull

Table 6.8 and figure 6.8 show the best performing implementations for sequential depth first and breadth first QuickHull on different data sets. On the Intel computer, we can see that the best implementations of depth first QuickHull perform better than breadth first QuickHull on all data sets. On POWER however, breadth first QuickHull performs better on the square and disk data sets. Breadth first QuickHull not performing better on the circle data set on any of the machines could indicate that the primary advantage of breadth first QuickHull is

Table 6.8: Best overall results for QuickHull

		Square	Disk	Circle
Depth First (Intel)	Best Time	12.65 ms	20.51 ms	139.43 ms
	Best Version	Q.T.Q. & Partitioning iii (single scan)	Q.T.Q. & Partitioning iii (single scan)	Sort Initially
Breadth First (Intel)	Best Time	14.58 ms	22.36 ms	186.74 ms
	Best Version	Always Compact & Partitioning iii (single scan)	Always Compact & Partitioning i (no X-partition)	Compact at End & Partitioning ii (partition by X)
Depth First (Power)	Best Time	33.01 ms	48.79 ms	348.89 ms
	Best Version	Partitioning iii (single scan)	Partitioning i (no X-partition)	Sort Initially
Breadth First (Power)	Best Time	30.47 ms	46.77 ms	443.31 ms
	Best Version	Always Compact & Partitioning iii (single scan)	Always Compact & Partitioning i (no X-partition)	Compact at End & Partitioning i (no X-partition)

the ability to compact the points array periodically by removing points determined to not be on the convex hull. Since the POWER computer has 128 byte cache lines whereas the Intel computer has 64 byte cache lines, the advantage gained from the reduced fragmentation of active points may be greater on POWER, which could explain why breadth first QuickHull performs better than depth first on POWER but not on the Intel computer.

6.3.4 Parallel QuickHull

In order to evaluate the degree of parallelism in our parallel implementations of QuickHull, we have run these implementations on the POWER computer with a varying number of threads. The measurements were performed on the randomly generated data sets with $n = 1000000$. 5 runs were performed for each test case and thread count, and the values presented are the average of all 50 runs for that data set. The number of threads was limited by running the process with the command `taskset -cpu-list 0-n`.

Figure 6.9 shows the performance of parallel depth first QuickHull, with partition strategy i (no X-partition). As discussed in section 4.5 (page 33) this implementation starts a new thread for one of the recursive calls in the divide and conquer phase. From the figure we can see that this method of parallelization does not seem very effective on the square and disk data sets, where the only significant performance improvement appears when increasing the thread count from 1 to 2 (for the disk data set) and from 8 to 9. However, parallelization on the circle data set appears to work well. This result could be explained by most of the work on the square and disk data sets being performed at low recursion depths, where work has not yet been split up into enough threads to allow for a large amount of parallelization.

The figure also shows the performance of some sequential implementations on the circle data set. One interesting thing to note here is that while sequential QuickHull generally performs worse than Monotone Chain on the circle cases, this parallel implementation of QuickHull performs better in our measurements with 4 or more threads. Since Monotone

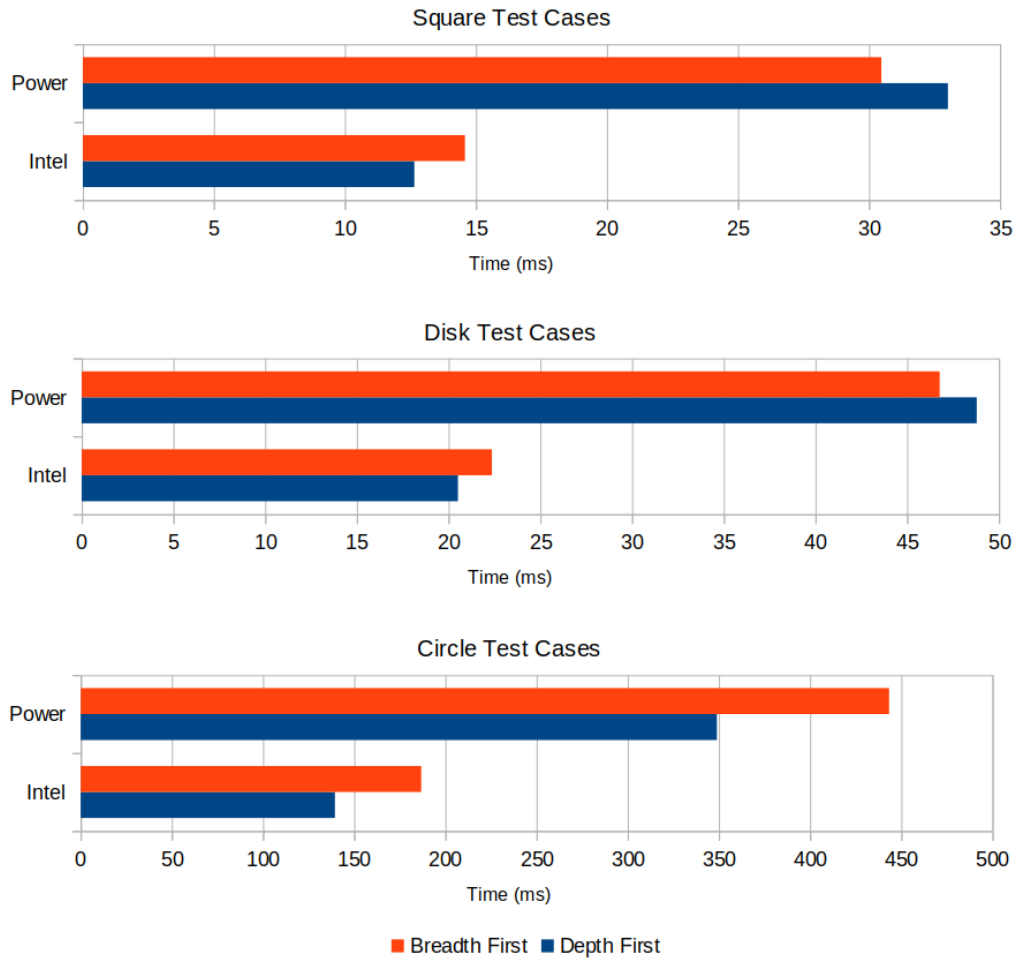


Figure 6.8: Bar charts showing the best overall results for QuickHull

Chain does not provide a similar simple approach to parallelization, the potential for parallelization could be an advantage of using QuickHull even on data sets where sequential QuickHull performs poorly.

Figure 6.10 shows the performance of parallel breadth first QuickHull. There are several interesting differences between these results and those for depth first QuickHull. Firstly, this parallel implementation performs much worse than depth first on all test data. This is especially true on circle cases, where performance generally degrades as the number of threads increases. Note that there are clear spikes in running time when the number of threads is increased past a multiple of 8, but that performance generally improves between these spikes. Since the POWER computer has 8 threads per core, this would suggest that the worsening performance is due to poor utilization of multiple cores.

In contrast to the behavior on circle cases, the performance for disk and square cases generally improves as the number of threads increases. Also for these cases there are slight spikes in performance when the thread count is increased past a multiple of 8, however the performance gained by the extra parallelism appears to be greater than the overhead of using another core, once all parallelism on the core is available. The reason for the better utilization of multiple cores on disk and square cases is likely that the initial phase of QuickHull consumes a larger percentage of the total execution time, and this phase makes better use

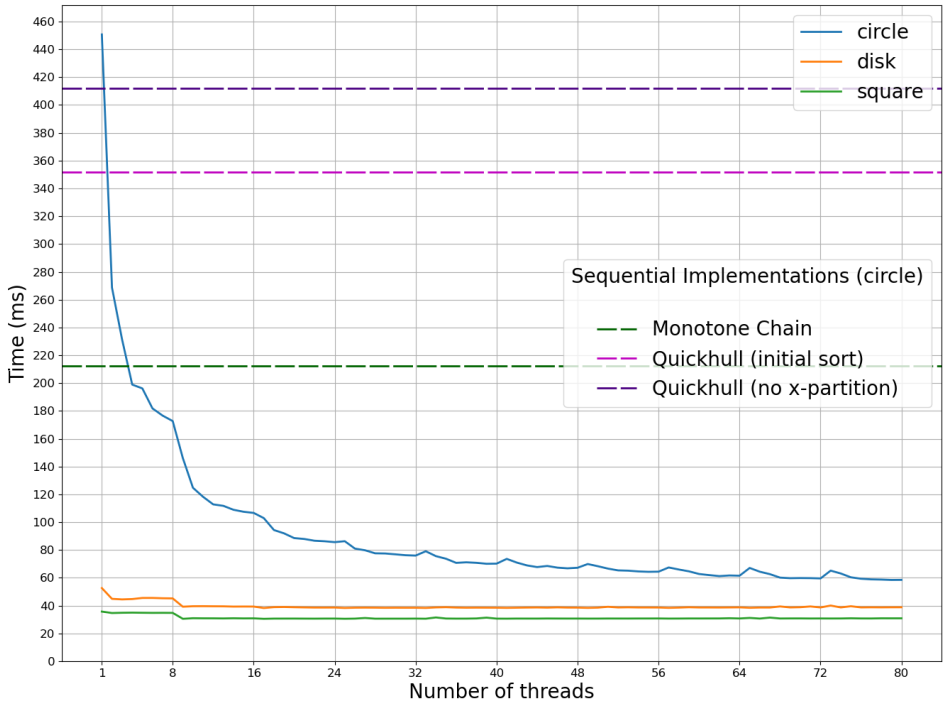


Figure 6.9: Running time of parallel depth first QuickHull on POWER as a function of thread count

of multiple cores. The time for the initial phase is shown in figure 6.11, where we can see that the initial phase does not have the same spiky behavior as the total running time on circle cases, and that the initial phase represents a large part of the running time for disk and square. These measurements were captured from circle inputs, but the time for the initial phase on other inputs is very similar.

While the performance on disk and square cases never reaches that of the sequential or depth first parallel implementations, the running time does improve as the amount of parallelism is increased. This would suggest that the breadth first implementation does not have the same problem with load balancing that the depth first implementation had in these cases.

Table 6.9 shows the timings from parallel implementations of QuickHull at specific thread counts, as well as timings from the best performing sequential implementations. These measurements are from the same set of measurements shown in figures 6.9 and 6.10 and previous tables for sequential implementations.

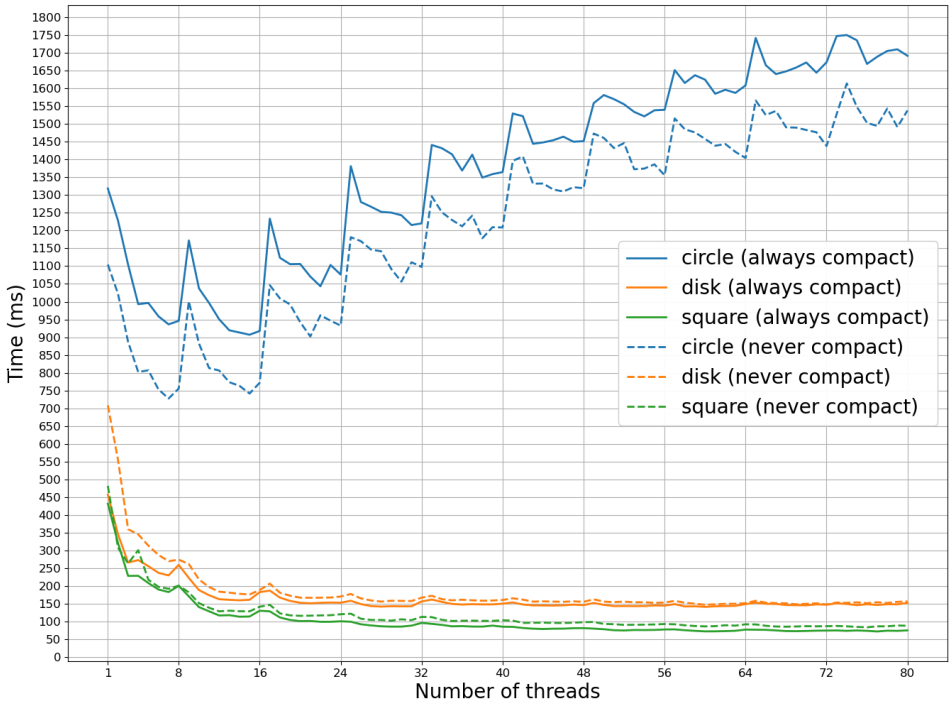


Figure 6.10: Running time of parallel breadth first QuickHull on POWER as a function of thread count

Table 6.9: Running time of various QuickHull implementations on POWER

			Square (ms)	Disk (ms)	Circle (ms)
Parallel QuickHull	Depth first	1 thread	35.74	52.63	450.68
		4 threads	34.97	44.74	198.93
		10 threads	30.96	39.58	124.74
		80 threads	30.89	38.83	58.53
	Breadth first	1 thread	431.54	456.79	1318.46
		4 threads	228.49	272.68	992.95
		10 threads	140.42	188.26	1037.40
		80 threads	74.86	152.17	1691.96
Sequential		Monotone Chain	233.00	233.16	212.15
		QuickHull (single-scan)	33.05	49.07	454.87
		QuickHull (sort by X)	192.48	205.76	351.74

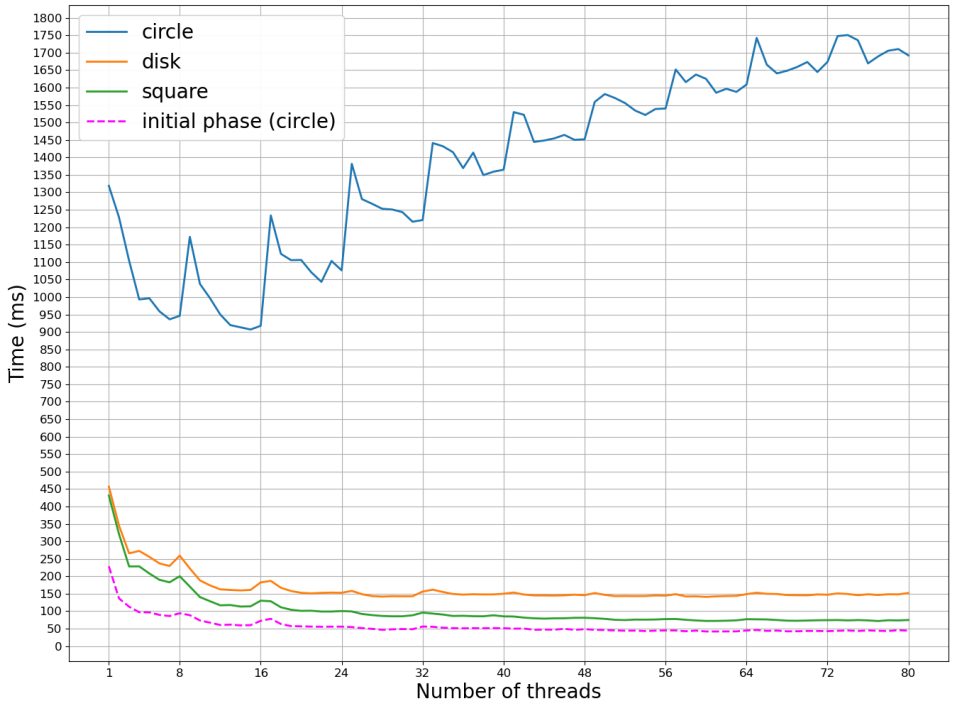


Figure 6.11: Running time of parallel breadth first QuickHull, with time for the initial phase.

6.4 Chan, Refined Chan and MergeHull

In chapters 2 and 3 of this thesis we introduced MergeHull in algorithm 3, Chan's algorithm in algorithm 4 and Refined Chan in algorithm 5. Refined Chan combines the worst case time complexity $O(n \log h)$ from Chan's algorithm with the average case time complexity $O(n)$ on truly sublinear inputs from MergeHull.

In figures 6.12, 6.13, and 6.14 we can see how our implementations of these algorithms compare to each other in terms of execution time per input point on different data sets.

In figure 6.12 we see that Refined Chan performs similarly to MergeHull on both the square and disk data set with $O(n)$ performance as is expected. On the disk input, Chan's algorithm can be seen having distinct increments in compute time which can be explained by the fact that when t loops one more step on line 2 in algorithm 4 lots of additional computations are introduced.

A similar pattern is seen in figure 6.13 but since the circle input is not truly sublinear and since $h = n$, the algorithms will run in $\theta(n \log n)$ time. Chan's algorithm has a distinct increment in compute time around $n = 65536 = 2^{2^4}$ which is explained by the fact that this is the point where t has to loop until $t = 5$ instead of $t = 4$.

So far Refined Chan just seems like a slower version of MergeHull. In figure 6.14 we see the difference between $O(n \log n)$ and $O(n \log h)$ behavior. MergeHull runs in $\Theta(n \log n)$ on the MergeHull killer inputs, because almost all partial hulls that it merges with will be of the

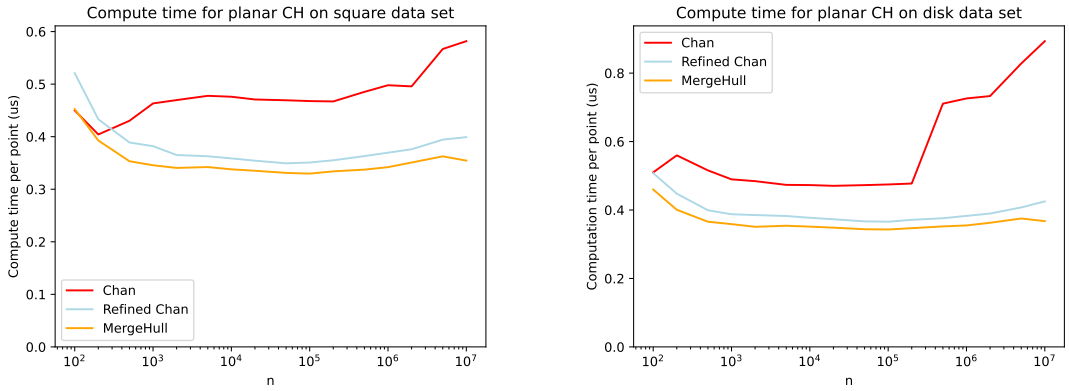


Figure 6.12: Computation time for Chan, MergeHull and Refined Chan on Square and Disk data sets of varying size.

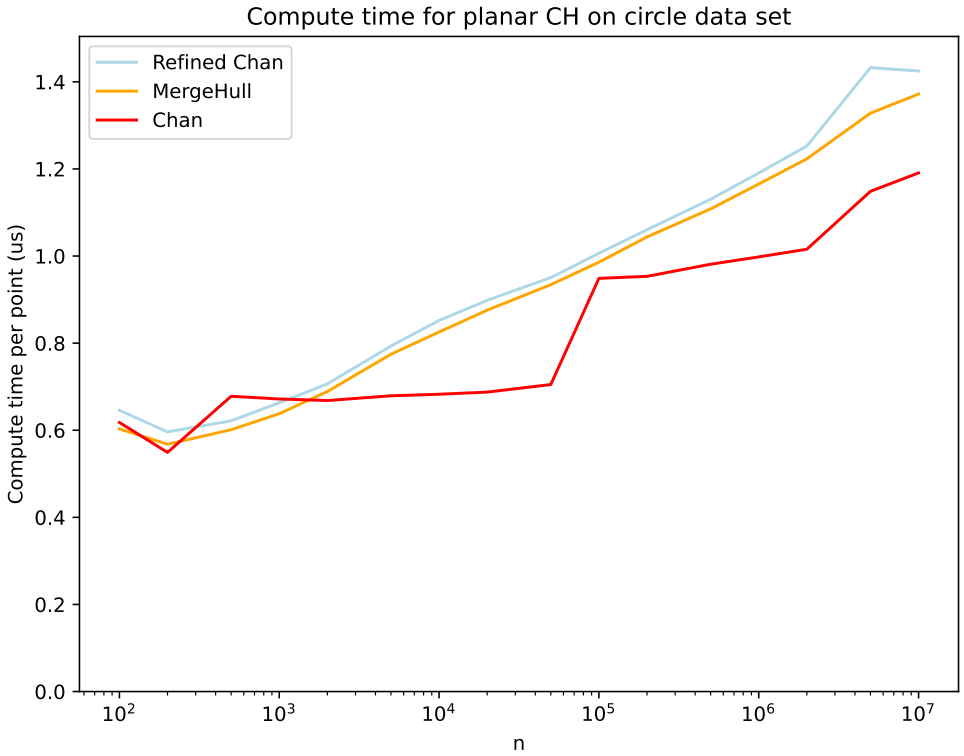


Figure 6.13: Computation time for Chan, MergeHull and Refined Chan on Circle data set of varying size.

same size as the number of points they represent. The other two algorithms run in $O(n \log h)$ which is just $O(n)$ since $h = 3$.

In order to speed up these algorithms, we added some intuitive optimizations. Firstly, instead of starting with hulls of size 1 and merging into larger hulls, we start with hulls of size 256 that we compute with the Monotone Chain algorithm, and start running the algorithm from there. This would, for a recursive algorithm, correspond to ending recursion

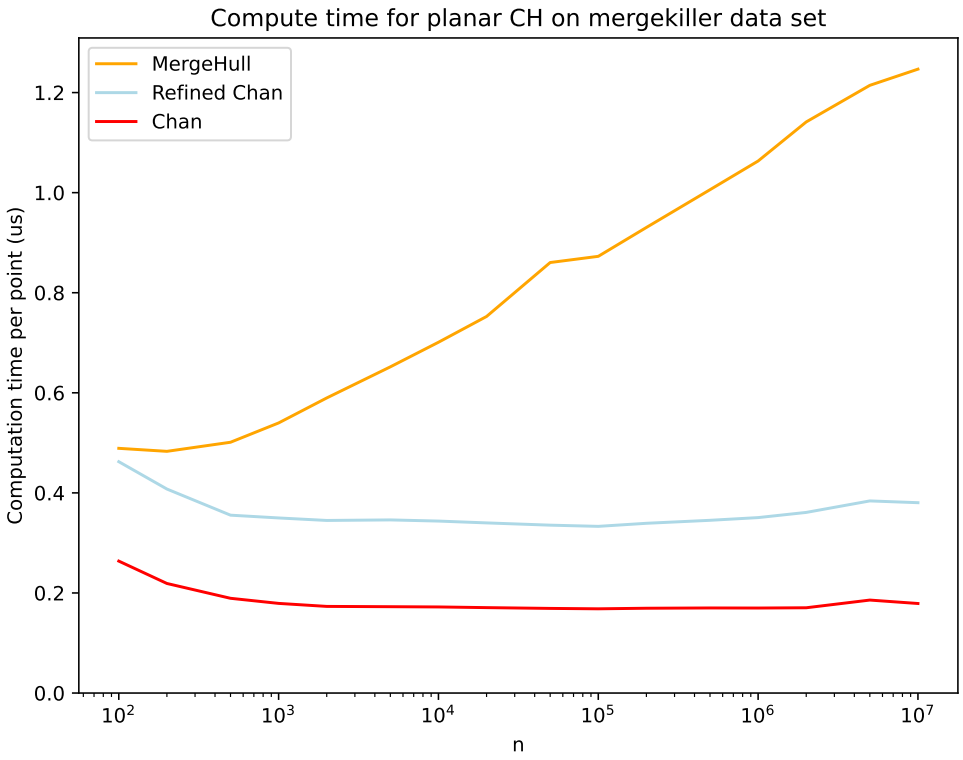


Figure 6.14: Computation time for Chan, MergeHull and Refined Chan on MergeHull killer data set of varying size.

early and solving the base case with another algorithm. Secondly, instead of merging hulls pairwise, we merge 3 hulls at a time. This would, again for a recursive algorithm, correspond to splitting the problem in 3 equally sized instances instead of 2. This way the merge step is more expensive but recursion is not as deep, or the number of iterations is lower for an iterative algorithm.

Combining these two optimizations to Refined Chan made it roughly 2 times faster on the square, disk, and circle data sets. Graphs showing this are found in Appendix A. We will use these optimizations for Refined Chan in the comparisons in section 6.5, since we are comparing it to other algorithms that have been tuned. However, as it will turn out Refined Chan is not a contender for the overall fastest algorithm, so spending a lot of time to carefully tune it was deemed unnecessary.

6.5 Overall Comparisons

In order to gain an understanding of which convex hull algorithm is the fastest in different scenarios, we compare QuickHull with Refined Chan and Monotone Chain. Furthermore in order to put our results into context of the industry standard, we include implementations from the libraries CGAL and Qhull.

From CGAL we are using their implementations of Graham Scan [10], and Akl and Tous-

saints's algorithm [2], since initial testing showed that these were generally the best performing implementations in the library. The latter algorithm is roughly equivalent to executing QuickHull to a depth of 2, and then running Graham Scan. We were able to import the CGAL source code directly into our project and run it under the same conditions as our own code, with the exception that CGAL writes output to a new array in memory whilst our algorithms are allowed to modify the input array.

Qhull was installed as a binary program and run separately. It reported back the computation time excluding input handling, so it should be measured on the same terms as the other implementations.

Since the difference between some of our QuickHull variants is quite small (mainly those that do not do any initial sorting), we have selected a subset of them as representatives. Depth first QuickHull with partitioning strategy iv (initial sort) represents QuickHull variants that perform a sorting step, and is therefore having $O(n \log n)$ behavior. Depth first QuickHull with partitioning strategy iii (single scan) represents QuickHull variants that do not perform a sorting step and most closely follows the pseudo code as introduced by Greenfield [11].

Algorithms that have been evaluated, but are not included in this analysis to remove clutter, are the following. The divide and conquer algorithm by Preparata and Hong [19], because it was generally slow. Chan's algorithm [7] and MergeHull [6], because their performance was comparable to Refined Chan. Jarvis march [14] because it is too slow as soon as h is not extremely small.

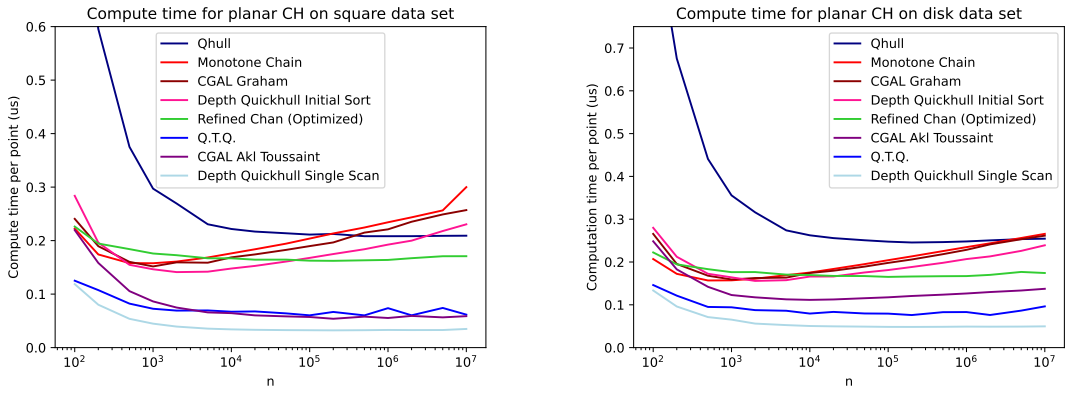


Figure 6.15: Computation time for various algorithms on Square and Disk inputs on the POWER computer.

In figure 6.15 we see how these algorithms compare to each other on the Square and Disk data sets on the POWER computer. It is clear that the single scan QuickHull variant performs very well on these inputs with relatively few hull points. Like before Quicker Than QuickHull is similar to single scan, but slower. The $\Theta(n \log n)$ algorithms Monotone Chain, Graham Scan and QuickHull with initial sort do not perform as well. Qhull has the same behavior as single scan QuickHull, but the constant factor is very bad. Akl Touissant also does well, especially on the square data set. This can be explained by the fact that it is pruning away points in a quadrilateral which is a good fit in this case. Refined Chan outperforms the $\Theta(n \log n)$ algorithms but is not close to the performance of single scan QuickHull.

In figure 6.16 we see how these algorithms compare to each other on the Circle data set. In this case the $\Theta(n \log n)$ algorithms perform well, with Monotone chain a clear winner. Akl

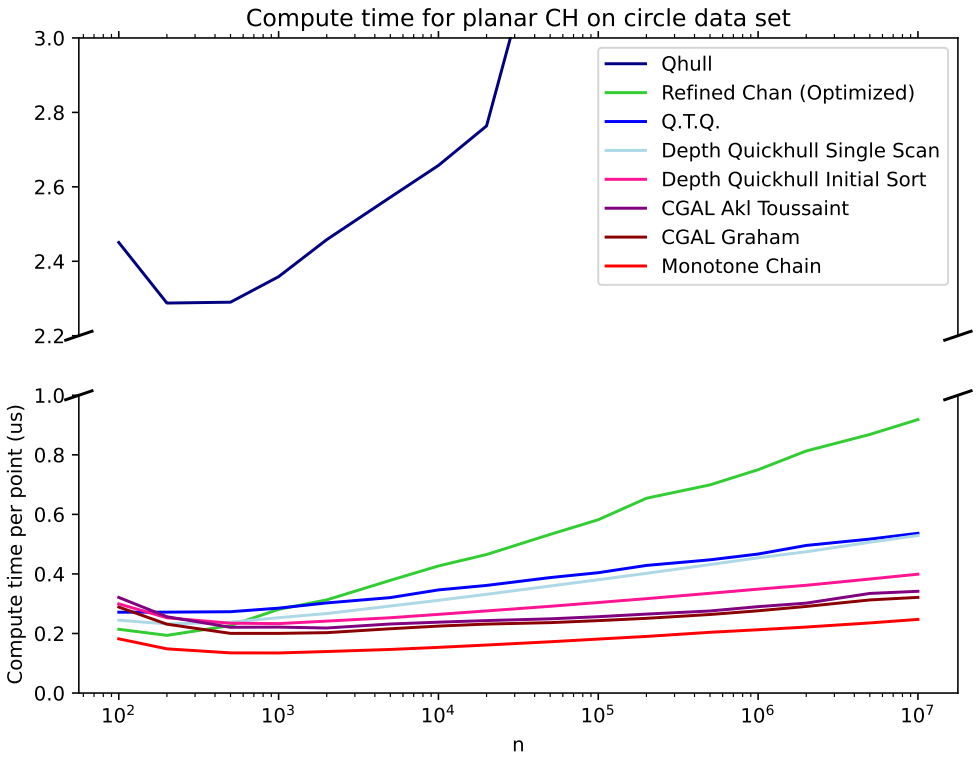


Figure 6.16: Computation time for various algorithms on Circle input on the POWER computer.

Toussaint still performs well, being on par with Graham scan.

Table 6.10: Computation times (µs per point) on POWER, overall comparison.

Implementation	Medical A	Medical B	Square	Disk	Circle	QuickHull killer
CGAL Akl Toussaint	0.069	0.066	0.055	0.126	0.290	60.3
CGAL Graham	0.171	0.174	0.221	0.228	0.276	0.214
Q.T.Q. iii (single scan)	0.068	0.068	0.074	0.083	0.467	6.62
Depth QuickHull iv (sort initially)	0.110	0.110	0.192	0.207	0.349	5.53
Depth QuickHull iii (single scan)	0.030	0.030	0.033	0.049	0.454	7.71
Refined Chan (Optimized)	0.169	0.171	0.164	0.167	0.750	0.071
Monotone Chain	0.186	0.189	0.234	0.235	0.213	0.105
Qhull	0.223	0.219	0.208	0.248	—	—

Table 6.11: Computation times (µs per point) on the Intel computer, overall comparison.

Implementation	Medical A	Medical B	Square	Disk	Circle	QuickHull killer
CGAL Akl Toussaint	0.030	0.028	0.026	0.065	0.136	0.710
CGAL Graham	0.084	0.085	0.108	0.104	0.131	0.051
Q.T.Q. iii (single scan)	0.012	0.012	0.013	0.022	0.218	2.390
Depth QuickHull iv (sort initially)	0.049	0.047	0.094	0.105	0.145	2.366
Depth QuickHull iii (single scan)	0.013	0.013	0.015	0.024	0.208	2.778
Refined Chan (Optimized)	0.070	0.070	0.073	0.073	0.323	0.016
Monotone Chain	0.087	0.086	0.120	0.118	0.105	0.030
Qhull	0.071	0.073	0.088	0.113	2.685	—

In Table 6.10 we see how the algorithms compare on the POWER computer also on the Medical and QuickHull Killer inputs. We see that the behavior of the algorithms on the Medical data sets is quite similar to the behavior on the square and disk data sets. The QuickHull killer input shows how the QuickHull variants are exposed to this type of input. They are almost 100 times slower than the other algorithms. Qhull is not able to run on this input, stating that there are numerical issues. One notable outlier is the AKL Toussaint algorithm from CGAL which is running very slow on the QuickHull killer input, almost 1000 times slower than other algorithms. We will discuss this further on.

In Table 6.11 we see the same results but on the Intel laptop. Some notable differences to the POWER measurements is that Quicker Than QuickHull is now faster than Depth first QuickHull on all data sets except circle, and that AKL Toussaint is less of an outlier on the QuickHull killer input, even though it is still quite slow.

We can see that generally the single scan QuickHull variant is the fastest, except on circular inputs. There is some discrepancy between the Intel and POWER results for how the variants of QuickHull compare to each other, but generally the Depth first QuickHull variant will perform well.

For the Circle data set we see how Monotone Chain clearly is the winner. One explanation is that for the circle input the algorithms need to sort all the points to find the output. Algorithms like QuickHull, Akl Toussaint, and Refined Chan that filter out points that are not on the hull are penalized by this because they are not able to remove any points in these steps.

The QuickHull killer input shows how even if QuickHull is generally a good choice and a fast algorithm, it is possible to force it to be very slow. As for the AKL Toussaint algorithm, it runs extremely slow on the POWER computer, but also slower than expected on the Intel computer. We do not have a clear explanation for this, as AKL Toussaints algorithm should run in $O(n \log n)$ time and not be much slower than Graham scan since it only performs an additional filtering step. We have identified that the filtering step is what is taking a majority of the execution time, specifically performing a linear number of side of line tests.

So for some reason some of the side of line tests become very slow in CGALs AKL Toussaint algorithm, but we are uncertain of exactly why.

Chapter 7

Conclusion

In this thesis we have analyzed the convex hull problem in the plane from both theoretical and practical viewpoints, with the aim of gaining a better understanding of which algorithms perform well in different circumstances.

We had three main research questions connected to our thesis, and believe we have been able to answer all of them to a satisfactory extent.

Experimental results show that in real world applications QuickHull is probably the fastest convex hull algorithm, but for some extreme applications one might want to use a sorting based algorithm like Monotone Chain. Among the QuickHull variants we have tested the single scan partitioning strategy appears to be generally fastest. Although there are cases where other partitioning strategies are faster, the difference between QuickHull variants that do not perform initial sorting is quite small, especially in comparison to other algorithms tested (with the exception of very special inputs).

In chapter 5 we have presented an upper bound on how deep QuickHull can recurse on floating point inputs, along with two constructions that achieve 0.96 and 0.82 of this depth. The former has some numerical issues whilst the other is what we call the "QuickHull killer" because it causes QuickHull to run very slowly. A way to avoid this would be to use some kind of hybrid algorithm, that falls back to a safer algorithm at some recursion depth, similar to the use of QuickSort in the Introsort algorithm. This would be a good idea if it is crucial that the algorithm can't run for too long.

In chapter 3 we showed how one can make a refined variant of Chan's algorithm that runs in $O(n)$ expected time on truly sublinear inputs. This could be validated with experimental results. Even though a similar theoretical running time as Refined Chan can be achieved by just implementing both Chan and MergeHull and running them at the same time, we believe that having a contained algorithm like Refined Chan to do this is nice. We also presented a simplified version of Refined Chan, which we believe is one of the shortest and easiest to understand convex hull algorithm with time complexity $O(n \log h)$.

7.1 Future Work

Throughout this project we have identified several areas of convex hull algorithms that fell outside the scope of this thesis, but that would be interesting to investigate and compare with the results found here.

One such area is convex hull in greater than two dimensions. In this thesis we have only examined planar convex hull algorithms, but various algorithms that we have discussed (such as QuickHull) can be expanded to more than two dimensions. It would be interesting to evaluate how these algorithms, and other algorithms, perform in more than two dimensions. Considering convex hull in three dimensions would also present more practical applications, such as computing the convex hull of 3D-meshes used in 3D-graphics applications and computing two dimensional Voronoi diagrams by reduction to three dimensional convex hull.

Regarding QuickHull, it would be interesting to further investigate why the results at times differ on POWER, especially for the "Quicker than QuickHull"-implementation.

Regarding QuickHull worst case inputs we proved an upper bound on the depth that QuickHull can recurse to for floating point inputs. We then showed a construction that assumes roughly 0.82 of this bound. It remains unknown to us if it is possible to make a construction that is closer to the bound than this.

In this thesis we have only investigated implementations where coordinates are represented using 64-bit double-precision floating point numbers. It would also be interesting to evaluate other formats, such as integers and 32-bit floating point. For 32-bit floating point it would be interesting to compare how much the performance improves in relation to precision losses. For integer formats it would be interesting to investigate how the performance changes between different number of bits per coordinate, and also investigate to what degree low bit width integer formats could be used in applications such as image processing.

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Selim G. Akl and Godfried T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219–222, 1978.

[3] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[4] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[5] Valentina Bayer. Survey of algorithms for the convex hull problem. 04 1999.

[6] Jon Louis Bentley and Michael Ian Shamos. Divide and conquer for linear expected time. *Information Processing Letters*, 7(2):87–91, 1978.

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

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022.

[9] Ask Neve Gamby and Jyrki Katajainen. Convex-hull algorithms: Implementation, testing, and experimentation. *Algorithms*, 11(12), 2018.

[10] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1:132–133, 1972.

[11] Jonathan S. Greenfield. A proof for a quickhull algorithm. 1990.

[12] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, dec 1986.

[13] Nam-Dũng Hoang and Nguyen Linh. Quicker than quickhull. *Vietnam Journal of Mathematics*, 43, 01 2014.

[14] R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[15] David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[16] Litao Ma, Runhua Zhao, and Dan Yang. Image registration based on convex hull ransac-like. 12 2009.

[17] Mary M McQueen and Godfried T Toussaint. On the ultimate convex hull algorithm in practice. *Pattern Recognition Letters*, 3(1):29–34, 1985.

[18] Mark H. Overmars and Jan van Leeuwen. Further comments on bykat's convex hull algorithm. *Information Processing Letters*, 10(4):209–212, 1980.

[19] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, feb 1977.

[20] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Monographs in Computer Science. Springer New York, 2012.

[21] Konstantin Sharafutdinov, Jayesh Bhat, Sebastian Fritsch, Kateryna Nikulina, Moein Samadi, Richard Polzin, Hannah Mayer, Gernot Marx, Johannes Bickenbach, and Andreas Schuppert. Application of convex hull analysis for the evaluation of data heterogeneity between patient populations of different origin and implications of hospital bias in downstream machine-learning-based data processing: A comparison of 4 critical-care patient datasets. *Frontiers in Big Data*, 5:603429, 10 2022.

[22] Jonas Skeppstedt and Christian Söderberg. *Writing efficient C code: a thorough introduction*. Independently published, 2020.

[23] Michal Smolik and Vaclav Skala. Efficient speed-up of the smallest enclosing circle algorithm. *Informatica*, 33(3):623–633, 2022.

[24] Stanley Tzeng and John D. Owens. Finding convex hulls using quickhull on the gpu, 2012.

[25] Junhui Wang, Bin Tian, Yachen Zhu, Tingting Yao, Ziyu Pan, and Long Chen. Terrain mapping for autonomous trucks in surface mine. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 4369–4374, 2022.

[26] Zhengwei Yang and F.S. Cohen. Image registration and object recognition using affine invariants and convex hulls. *IEEE Transactions on Image Processing*, 8(7):934–946, 1999.

Appendices

Appendix A

Performance of Refined Chan with and without optimizations

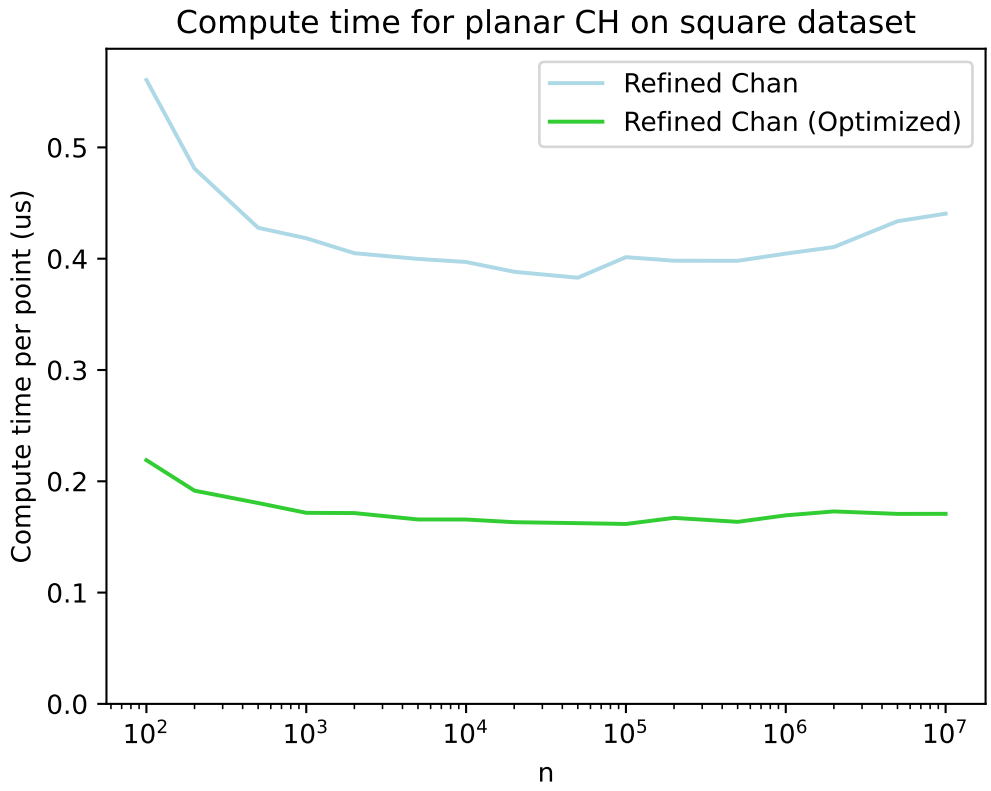


Figure A.1: Computation time for Refined Chan with and without basic optimizations on Square data set of varying size.

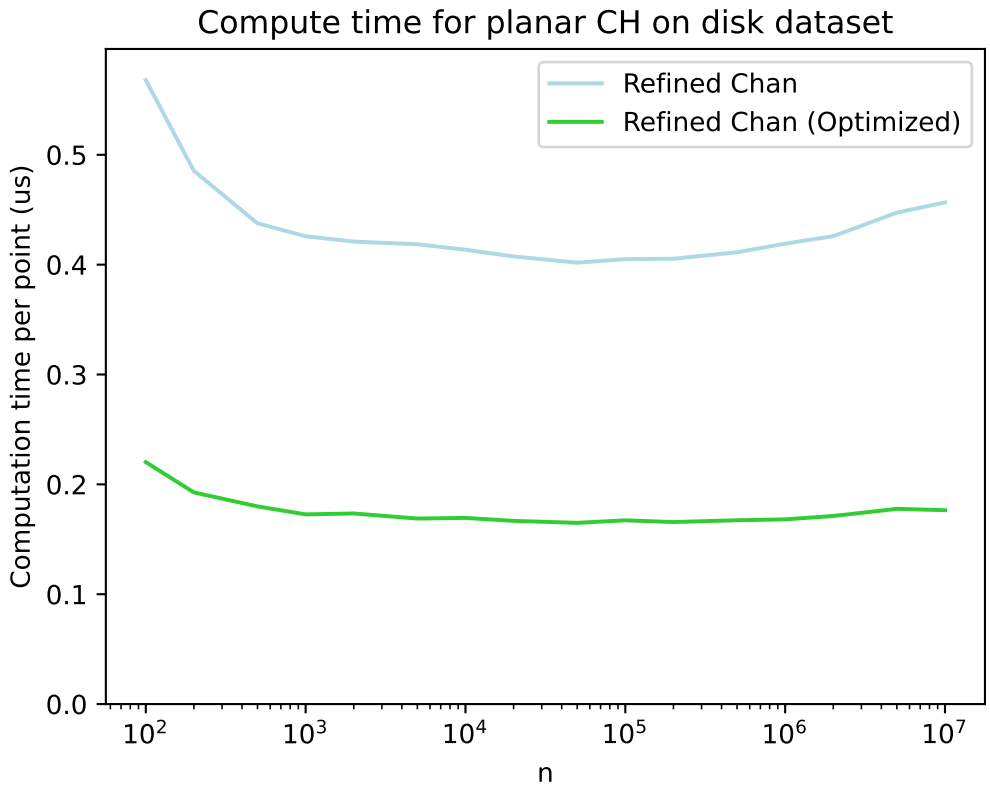


Figure A.2: Computation time for Refined Chan with and without basic optimizations on Disk data set of varying size.

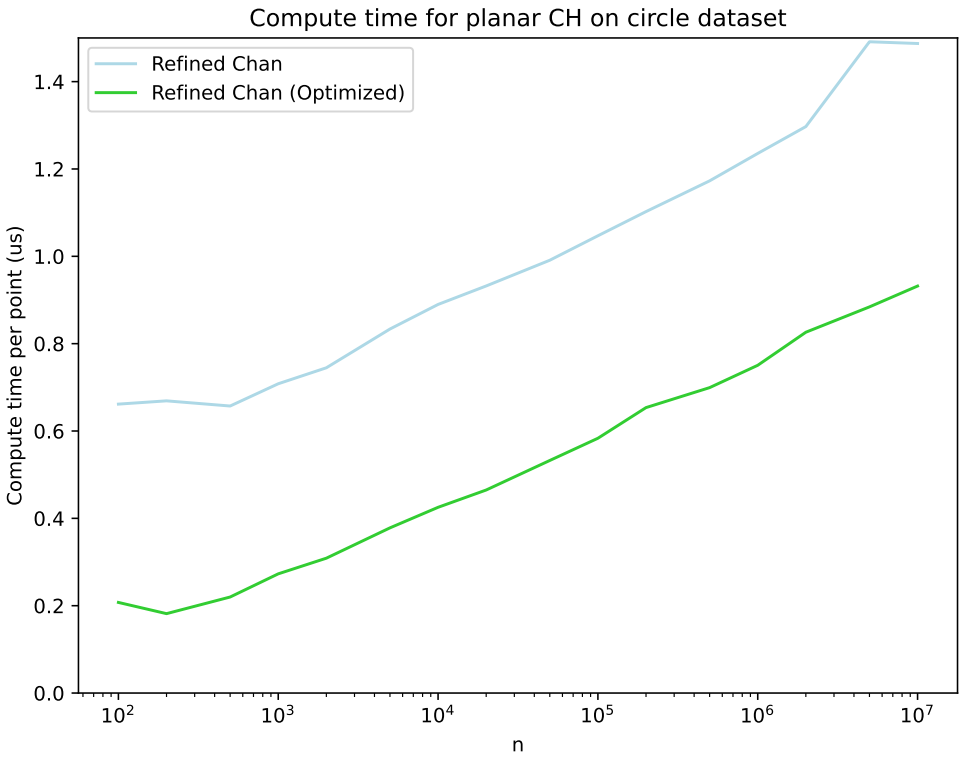


Figure A.3: Computation time for Refined Chan with and without basic optimizations on Circle data set of varying size.

EXAMENSARBETE
Improvements to Planar Convex Hull Algorithms through Theoretical and Empirical Analysis
STUDENTER Björn Magnusson, Erik Amirell Eklöf
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Michael Doggett (LTH)

Bygg en hage så snabbt som möjligt

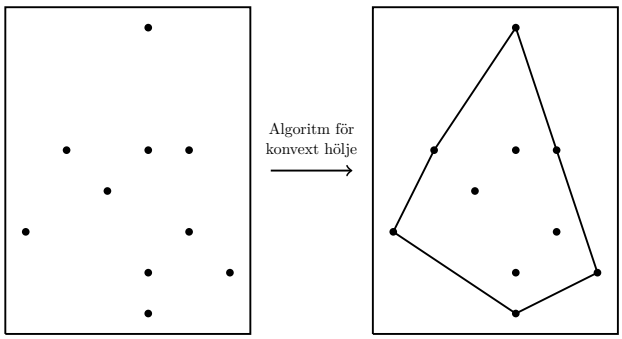
POPULÄRVETENSKAPLIG SAMMANFATTNING **Björn Magnusson, Erik Amirell Eklöf**

Att räkna ut ett konvext hölje kan användas för att jämföra hur mycket två mängder mätdata överlappar eller matcha ihop två 3D-bilder på en hjärna. I vårt arbete har vi undersökt hur man ska göra algoritmerna som räknar ut konvext hölje så snabba som möjligt.

Vi har undersökt algoritmer som räknar ut konvext hölje. Man kan tänka på det som att man vill bygga en så liten hage som möjligt runt ett antal blommor. Kanske vill man skydda dem från att bli uppätna?

Konvexa höljen kan till exempel användas för att uppskatta hur stort område en ny sjukdom har spridits till eller hur lika två grupper patienter är.

Bilden nedan illustrerar beräkningen av det konvexa höljet av ett antal punkter. I fallet med blommor skulle punkterna i den vänstra bilden representera blommor sedda ovanifrån. Dessa punkter matas sedan in i en algoritm för konvext hölje som genererar hagen i den högra bilden.



Det finns många algoritmer för att beräkna konvext hölje, som fungerar olika bra i olika sammanhang. Till exempel beroende på hur punkterna är fördelade och vilken sorts dator som används.

För att kunna räkna ut konvexa höljen snabbare och effektivare har vi undersökt vilken algoritm som blir snabbast och i vilka sammanhang. På så sätt kan konvexa höljen av stora mängder punkter beräknas snabbare och med mindre elförbrukning.

Vi har kommit fram till att algoritmen QuickHull nästan alltid är snabbast. QuickHull hittar en punkt som är på det konvexa höljet, och delar sedan problemet i två halvor med punkterna som ska vara till höger och vänster om punkten. Detta gör även att QuickHull kan parallelliseras och bli snabbare på en dator med många kärnor.

QuickHull har dock svagheter. Om punkterna är fördelade så att det konvexa höljet måste bestå av väldigt många punkter kan den ibland bli långsam. Detta kan till exempel inträffa om punkterna ligger på en cirkel. I detta fall har vi identifierat en annan algoritm som presterar bättre än QuickHull. Denna algoritm börjar med att sortera alla punkter. Oftast tar detta onödigt lång tid, men det visar sig att sorteringen lönar sig för punkter på en cirkel.

Vi har också kommit på en ny algoritm som parar ihop punkterna till större och större höljen (hagar), och ibland prövar att sätta ihop alla höljen till ett stort. Den är inte lika snabb som QuickHull, men är lätt att förstå sig på och har inte svagheter på samma sätt.