Computer-Aided Design 70 (2016) 153–160

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

A Total Order Heuristic-Based Convex Hull Algorithm for Points in the Plane☆

Abel J.P. Gomes

Instituto de Telecomunicações, Universidade da Beira Interior, Portugal

HIGHLIGHTS

- We propose a 2D convex hull algorithm based on comparison operators.
- We propose a 2D convex hull algorithm that outperforms Quickhull.
- We propose a 2D non-convex hull algorithm.

ARTICLE INFO

Keywords:
Convex hull
Geometric algorithms
Computational geometry

ABSTRACT

Computing the convex hull of a set of points is a fundamental operation in many research fields, including geometric computing, computer graphics, computer vision, robotics, and so forth. This problem is particularly challenging when the number of points goes beyond some millions. In this article, we describe a very fast algorithm that copes with millions of points in a short period of time without using any kind of parallel computing. This has been made possible because the algorithm reduces to a sorting problem of the input point set, what dramatically minimizes the geometric computations (e.g., angles, distances, and so forth) that are typical in other algorithms. When compared with popular convex hull algorithms (namely, Graham's scan, Andrew's monotone chain, Jarvis' gift wrapping, Chan's, and Quickhull), our algorithm is capable of generating the convex hull of a point set in the plane much faster than those five algorithms without penalties in memory space.

1. Introduction

The *convex hull* $\mathbb{H}(\mathcal{P})$ of a planar point set $\mathcal{P}$ can be defined as the smallest convex polygon that encloses $\mathcal{P}$. Every point of $\mathcal{P}$ belonging to the boundary of $\mathbb{H}(\mathcal{P})$ is called an *extreme vertex*. The notion of convex hull is considered by many as one of the most fundamental geometric structures we find in computational geometry, computer graphics, robotics, etc. [1]. In fact, important problems in computational geometry like Delaunay triangulation, Voronoi diagrams, halfsapce intersection, etc. can be reduced to the problem of computing the convex hull of a set of points [2].

Besides, the problem of finding the convex hull crosses many research domains and applies to an endless number of problems and situations. For example, convex hulls play an important role in computer vision [3], pattern recognition [4,5], visual pattern matching [6], operations research [7], path planning and obstacle avoidance in robotics [8,9], astronomy [10,11], and biology and genetics [12], just to mention a few of them.

The remainder of the paper is organized as follows. Section 2 overviews the prior work on convex hull algorithms. Section 3 describes the TORCH (Total Order-Based Convex Hull) algorithm step by step. Section 4 carries out the complexity analysis of the TORCH algorithm. Section 5 compares the TORCH algorithm to other well-known convex hull algorithm with reference to both arbitrary and definite sets of points in the plane. Section 6 concludes the paper, with some hints for future work.

2. Related work

According to Avis et al. [13], the convex hull algorithms fall into two categories: *graph traversal* and *incremental*. Graham scan [14], Jarvis march [15] and monotone chain [16] are representatives of graph traversal algorithms. In the graph traversal algorithms, the input points work as vertices of a graph whose edges are formed temporarily to check whether two connected edges are convex or not. For example, Graham scan uses the angle between two connected edges to decide about the convexity on the shared vertex.

☆ This paper has been recommended for acceptance by Scott Schaefer and Charlie C.L. Wang.

E-mail address: agomes@di.ubi.pt.

http://dx.doi.org/10.1016/j.cad.2015.07.013
0010-4485/

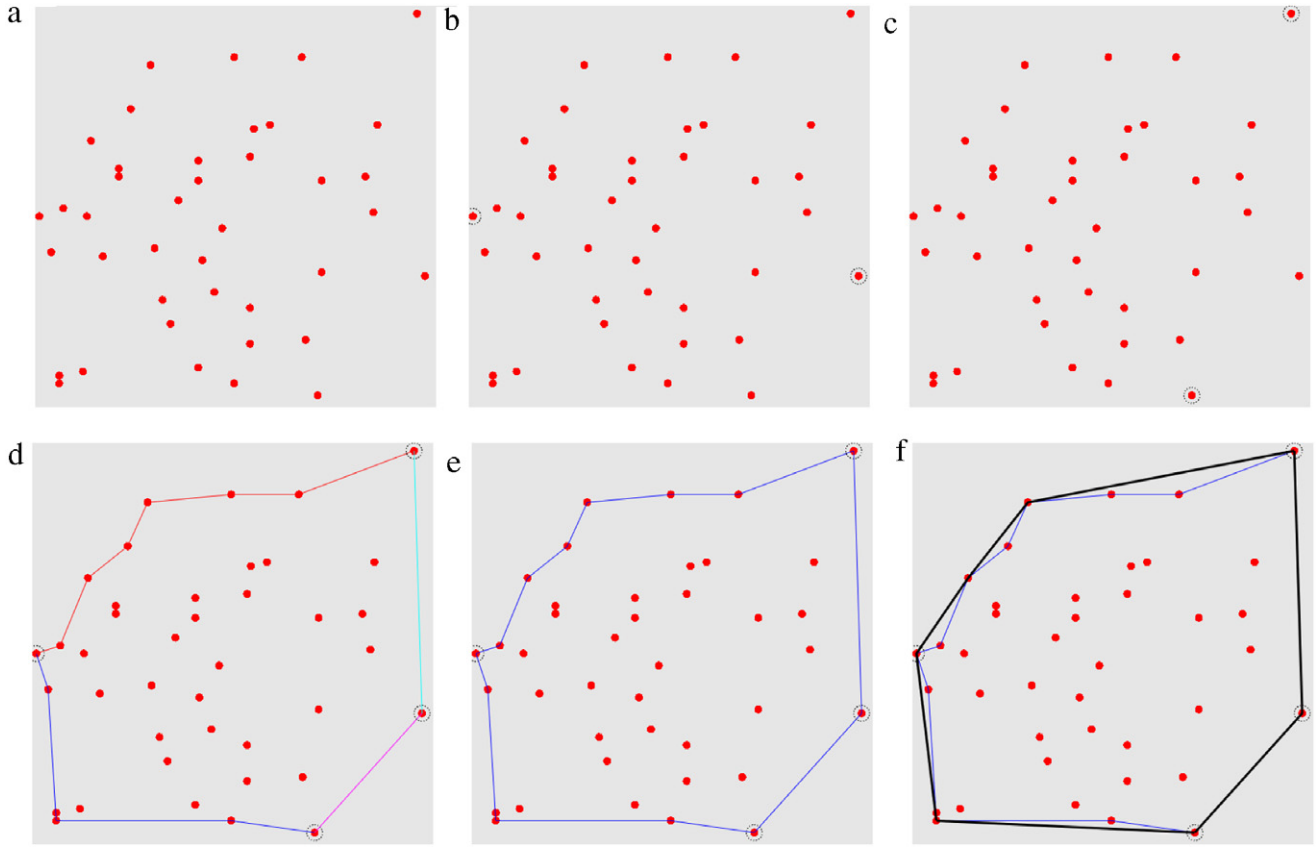

Fig. 1. The algorithm steps: (a) set of 40 points $\mathbb{P}$ sorted along x in the domain $D = [0, 100] \times [0, 100]$; (b) west and east poles found (dotted circles); (c) south and north poles found (dotted circles); (d) lateral hulls: $\mathbb{H}_{SW}$ in blue, $\mathbb{H}_{SE}$ in magenta, $\mathbb{H}_{NE}$ in cyan, and $\mathbb{H}_{NW}$ in red; (e) approximate convex hull $\mathbb{A}$ in blue after connecting lateral hulls counterclockwise; (f) convex hull $\mathbb{H}$ after discarding concave vertices of $\mathbb{A}$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Incremental algorithms start from an initial convex hull (e.g., a triangle), checking then whether each of the remaining points belongs to the current convex hull or not. If any of these remaining points is outside the current convex hull, then the convex hull is updated accordingly. Quickhull [17], divide-and-conquer [1] and incremental [18] algorithms are some representatives of this category.

The convex algorithm proposed in this article, called total order heuristic-based convex hull (TORCH) algorithm, is in its essence a sorting algorithm, being its geometric computations reduced to a minimum. As a result of such a sorting procedure we immediately obtain an approximate convex hull (i.e., a non-convex hull) that contains all the extreme vertices of convex hull and a few concave vertices. The last stage of the algorithm consists in discarding those concave vertices from the approximate convex hull using the geometric operation CCW that is employed in Andrew's monotone chain. Our algorithm was designed for serial computing, so that we do not use any GPU resources or other parallel resources in any way. The main contribution of the TORCH algorithm lies in the use of comparison operators to build up the convex hull from the four extremal points. Besides, TORCH outperforms Quickhull, which likely is the fastest amongst the currently known algorithms.

3. The algorithm

The general idea of the algorithm is that a convex hull is like a 2-dimensional ball (or circle), and as every single 2-dimensional ball it has four turning points or poles, that is, west, east, south and north. The west pole is the leftmost point, the east pole is the rightmost point, the south pole is the bottommost point, and the north pole is the topmost point. This is similar to the computation of the elimination quadrangle of the Quickhull algorithm (cf. [17]), whose vertices are precisely those poles.

The algorithm proposed in the present article distinguishes from others, including Andrew's and Quickhull, in the manner how the four hulls between the turning points (or poles) are calculated. In general terms, and after allocating memory for input points, our algorithm consists of the following steps:

(1) Sort the point set in the x-direction of the domain (Fig. 1(a)).
(2) Find the leftmost and rightmost points (inside dotted circles) shown in Fig. 1(b).
(3) Find the bottommost and topmost points (inside dotted circles) shown in Fig. 1(c).
(4) Find the four lateral hulls between turning points (Fig. 1(d)).
(5) Construct approximate convex hull (in blue) by merging the four lateral hulls (Fig. 1(e)).
(6) Inflate the approximate convex hull towards the convex hull in black (Fig. 1(f)). This inflating operation is also called here *convexification*.

Let us then describe these steps of the algorithm in more detail in the following subsections.

3.1. Sorting points

After allocating the entire set $\mathcal{P} = \{p_i\}_{i=0,\ldots,k-1}$ of input points in a 1-dimensional array, we proceed to their lexicographical sorting in the x-direction, as in Andrew's algorithm [16]. This results in a sorted array $\mathbb{P}$. For this purpose, we have adopted the introspective sort (or introsort) algorithm due to Musser [19],

Algorithm 1 Finding Southwest Hull

Ensure: $\mathbb{H}_{SW}$: southwest hull
Ensure: $\mathbb{P}$: sorted set of points
Ensure: s: index of the south point of $\mathbb{P}$
1: **procedure** SouthwestHull($\mathbb{H}_{SW}$)
2: $min \leftarrow \mathbb{P}[0].y$ ▷ west pole is the first minimum
3: $\mathbb{H}_{SW} \leftarrow H \cup \mathbb{P}[0]$ ▷ add west pole to $\mathbb{H}_{SW}$
4: **for** $i = 1 \rightarrow s$ **do** ▷ find sequence of minima
5: **if** $\mathbb{P}[i].y \leq min$ **then**
6: $min \leftarrow \mathbb{P}[i].y$
7: $\mathbb{H}_{SW} \leftarrow \mathbb{H}_{SW} \cup \mathbb{P}[i]$
8: **end if**
9: **end for**
10: **end procedure**

Algorithm 2 Finding Southeast Hull

Ensure: $\mathbb{H}_{SE}$: southeast hull
Ensure: $\mathbb{P}$: sorted set of points
Ensure: s: index of the south point of $\mathbb{P}$
1: **procedure** SoutheastHull($\mathbb{H}_{SE}$)
2: $min \leftarrow \mathbb{P}[N-1].y$ ▷ east pole is the first minimum
3: $\mathbb{H}_{SE} \leftarrow \mathbb{H}_{SE} \cup \mathbb{P}[k-1]$ ▷ add east pole to $\mathbb{H}_{SE}$
4: **for** $i = k-2 \rightarrow s$ **do** ▷ find sequence of minima
5: **if** $\mathbb{P}[i].y \leq min$ **then**
6: $min \leftarrow \mathbb{P}[i].y$
7: $\mathbb{H}_{SE} \leftarrow \mathbb{H}_{SE} \cup \mathbb{P}[i]$
8: **end if**
9: **end for**
10: **end procedure**

which combines the good parts of the quicksort and heapsort algorithms; in particular we have used the introsort algorithm wrapped in the `sort` method for arrays of the C++ Standard Template Library (STL).

3.2. Finding west and east points

The west (leftmost) and east (rightmost) points of the original set $\mathcal{P}$ are precisely the first and last points of sorted array $\mathbb{P}$. That is, in respect to $\mathbb{P}$, the index of the west point is $w = 0$, while the index of the east point is $e = k - 1$.

3.3. Finding south and north points

We could determine the south (bottommost) and north (topmost) points of $\mathcal{P}$ by sorting its points in the y-direction. But, the idea is not to sort again the sorted array $\mathbb{P}$ because that procedure would change the indexing order of its elements. Instead, we intend to determine the indices s and n of the south and north points, respectively. Basically, this procedure consists in determining the points with minimum y (south) and maximum y (north).

3.4. Finding lateral hulls

The main novelty of our algorithm lies in the way we construct the four lateral hulls through sequences of minima between turning points (i.e., west pole W, east pole E, south pole S and north pole N). As shown further ahead, this results in a point elimination procedure that is much faster than the famous quick elimination of the Quickhull algorithm. In fact, unlike the Quickhull algorithm, there is no need for calculating distances and membership geometric operations, which are moderately time-consuming operations. TORCH replaces these time-consuming operations by rather quicker operations, known as comparison operations, $\leq$ and $\geq$, as shown in Algorithms 1–2. Therefore, TORCH only uses sorting operations (i.e., no geometric operations) in determining the approximate convex hull of a set of points.

3.4.1. Southwest hull

Finding the southwest hull $\mathbb{H}_{SW}$ is a procedure that is described in Algorithm 1. Let us then see how we build up $\mathbb{H}_{SW}$ from W down to S, where W is the highest minimum and S is the lowest minimum of the sequence of minima in the y-direction downwards. For this purpose, let us also imagine a drop of water falling down from W towards S by iterating through points of $\mathbb{P}$. In order to guarantee that the drop of water keeps descending from W to S, we pick up successive points between W and S in $\mathbb{P}$ with decreasing y values. Points that do not satisfy this monotonic criterion $\leq$ (cf. line 5 in Algorithm 1) are discarded straight away. Note that the sequence of minima in the y-direction downwards is obtained by iterating on the x-direction sorted array $\mathbb{P}$, more specifically between the points W and S, whose array indices are $w = 0$ and s, respectively, as illustrated by the polyline in blue in Fig. 1(d).

3.4.2. Southeast hull

We also find the southeast hull $\mathbb{H}_{SE}$ as a sequence of minima. Nevertheless, the construction of this hull starts at a different input point, more specifically at the east point E, although it also finishes at the south point S of $\mathbb{P}$; that is, the water now flows from E to S. So, by traversing the x-direction sorted array $\mathbb{P}$ backwards from the $(k-1)$-th point down to sth point, we build up $\mathbb{H}_{SE}$ as the result from collecting successive minima in between (cf. lines 4–9 in Algorithm 2). The southeast hull is the polyline in magenta depicted in Fig. 1(d).

3.4.3. Northwest hull

The computation of the northwest hull $\mathbb{H}_{NW}$ is similar to the one of the southwest hull $\mathbb{H}_{SW}$, but upside down. In fact, we find the northwest hull as a sequence of maxima starting at the west point and finishing at the north point of $\mathbb{P}$. Therefore, the water climbs the hill from W to N challenging the gravity. For this, one searches for the sequence of consecutive maxima in the y-direction as one traverses $\mathbb{P}$ from the 0-th point (i.e., west point W) to the nth point (i.e., north point N). The northwest hull is the polyline in red depicted in Fig. 1(d). Note that the algorithm that determines $\mathbb{H}_{NW}$ is conceptually identical to the one that finds $\mathbb{H}_{SW}$, since we replace the variable *min* by the variable *max*, and the operator $\leq$ by $\geq$ in line 5 of Algorithm 1.

3.4.4. Northeast hull

Likewise, we can find the northeast hull $\mathbb{H}_{NE}$ as a sequence of maxima in the y-direction starting at the east point E and finishing at the north point N of $\mathbb{P}$. This means that we have to traverse $\mathbb{H}_{NE}$ from the end position $k-1$ of the east point E up to the position n of the north point N. The northeast hull is the polyline in cyan depicted in Fig. 1(d). Therefore, the algorithm to determine $\mathbb{H}_{NE}$ is similar to the one that finds $\mathbb{H}_{SE}$, with the variable *min* replaced by the variable *max*, and the operator $\leq$ replaced by the operator $\geq$ in line 5 of Algorithm 2.

In short, building up the south hulls, $\mathbb{H}_{SW}$ and $\mathbb{H}_{SE}$, reduces to the computation of successive minima in the y-direction downwards, which essentially involves the arithmetic comparison operation $\leq$. On the other hand, finding the north hulls, $\mathbb{H}_{NW}$ and $\mathbb{H}_{NE}$, involves the computation of successive maxima in respect to increasing y through the arithmetic comparison operation $\geq$. Summing up, building up the lateral hulls of a point set can be reduced to a sorting procedure because they are totally ordered sets; more specifically, $\mathbb{H}_{SW} = (H_{SW}, \overset{y}{\leq})$, $\mathbb{H}_{SE} = (H_{SE}, \overset{y}{\leq})$, $\mathbb{H}_{NW} = (H_{NW}, \overset{y}{\geq})$, and $\mathbb{H}_{NE} = (H_{NE}, \overset{y}{\geq})$, where H_{SW}, H_{SE}, H_{NW} and H_{NE} are the corresponding sets of points.

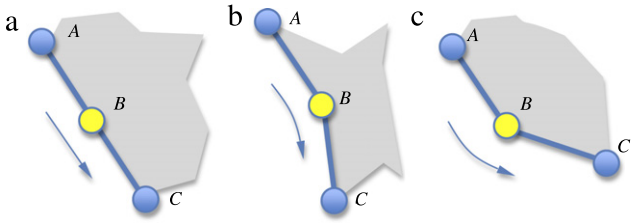

Fig. 2. The convexity test for the intermediate vertex B: (a) colinear; (b) concave; (c) convex.

3.5. Joining lateral hulls

Before proceeding any further, we have to point out that the four lateral hulls $\mathbb{H}_{SW}$, $\mathbb{H}_{SE}$, $\mathbb{H}_{NW}$ and $\mathbb{H}_{NE}$ are approximate hulls. So, joining these lateral hulls in a counterclockwise manner results in the approximate convex hull $\mathbb{A} = \mathbb{H}_{SW} \bowtie \mathbb{H}_{SE} \bowtie \mathbb{H}_{NE} \bowtie \mathbb{H}_{NW}$, where $\bowtie$ is the concatenation operation. Note that $\mathbb{A}$ contains the extreme vertices of the final convex hull $\mathbb{H}$ and a number of concave boundary points. Fig. 1(e) shows the approximate convex hull $\mathbb{A}$ in blue.

3.6. Computing the convex hull

The previous steps of the TORCH algorithm only involve sorting operations. The resulting approximate convex hull $\mathbb{A}$ encloses the input set of points $\mathcal{P}$. Besides, the boundary of $\mathbb{A}$ consists of the convex points of the convex hull $\mathbb{H}$ to be calculated, together with a number of concave points. Therefore, we only need to discard those concave points of $\mathbb{A}$ to obtain $\mathbb{H}$, a procedure that can be accomplished using the well-known CCW operation due to Andrew [16]. Intuitively, this amounts to inflate the approximate convex hull $\mathbb{A}$ to this way getting the convex hull $\mathbb{H}$. This inflating procedure is detailed in Algorithm 3.

Let $A = (A_x, A_y)$, $B = (B_x, B_y)$, and $C = (C_x, C_y)$ three consecutive vertices of the approximate convex hull $\mathbb{A}$. The inflation procedure is based on the following condition:

$$\delta_x.\Delta_y - \delta_y.\Delta_x \leq 0 \tag{1}$$

where $\delta_x = B_x - A_x$, $\delta_y = B_y - A_y$, $\Delta_x = C_x - A_x$, $\Delta_y = C_y - A_y$.

The condition (1) features a convexity test (cf. lines 13–18 in Algorithm 3). More specifically, this condition represents the slope variation from the line segment $\overline{AB}$ to the line segment $\overline{BC}$. This is illustrated in Fig. 2, where we observe three distinct cases:

- *Colinearity*. When the segments $\overline{AB}$ and $\overline{AC}$ are colinear, their slopes are identical, that is,

$$\frac{\Delta_y}{\Delta_x} = \frac{\delta_y}{\delta_x}. \tag{2}$$

  This relation also establishes that the two right triangles with hypotenuses given by $\overline{AB}$ and $\overline{AC}$ are similar.
- *Concavity*. When the segments $\overline{AB}$ and $\overline{AC}$ are not colinear and the slope of $\overline{AC}$ is less than the slope of $\overline{AB}$, that is,

$$\frac{\Delta_y}{\Delta_x} < \frac{\delta_y}{\delta_x}. \tag{3}$$

- *Convexity*. When the segments $\overline{AB}$ and $\overline{AC}$ are not colinear and the slope of $\overline{AC}$ is greater than the slope of $\overline{AB}$, that is,

$$\frac{\Delta_y}{\Delta_x} > \frac{\delta_y}{\delta_x}. \tag{4}$$

Algorithm 3 Inflating approximate convex hull

Ensure: $\mathbb{A}$: approximate convex hull

```
1:  procedure InflatingApproxConvexHull(𝔸)
2:      m ← #𝔸                                   ▷ number of points of 𝔸
3:      count ← 0                                ▷ counter for backtacking of 𝔸
4:      if m ≥ 3 then
5:          for i = 0 → m − 1 do
6:              A ← 𝔸[(i − count) mod m]
7:              B ← 𝔸[(i + 1) mod m]
8:              C ← 𝔸[(i + 2) mod m]
9:              δx ← Bx − Ax
10:             δy ← By − Ay
11:             Δx ← Cx − Ax
12:             Δy ← Cy − Ay
13:             if δx.Δy − δy.Δx ≤ 0 then        ▷ B is concave
14:                 count ← count + 1
15:             else                             ▷ B is convex
16:                 count ← 0
17:                 ℍ ← ℍ ∪ {B}
18:             end if
19:         end for
20:     end if
21: end procedure
```

By combining the conditions (2) and (3) we obtain the condition (1). This condition (1) is algebraically equivalent to counterclockwise (CCW) condition given by the cross product of the two vectors $\overrightarrow{AB} = (\delta_x, \delta_y)$ and $\overrightarrow{AC} = (\Delta_x, \Delta_y)$ that is defined by the following 2×2 determinant

$$\begin{vmatrix} \delta_x & \delta_y \\ \Delta_x & \Delta_y \end{vmatrix} \leq 0.$$

This counterclockwise test is analogous to sorting two numbers in the sense that either they are in the counterclockwise order or in the opposite order (i.e., clockwise order).

Looking at Algorithm 3, we also observe the inflation process hides the following details:

- The modulo operator in lines 6–8 allows us to iterate on the circular sequence of vertices of the approximate convex hull. This is relevant to consider the ending and beginning elements of $\mathbb{A}$ as consecutive elements of the sequence.
- The variable *count* is necessary to stop the progression of the first element A of the sequence when the previous intermediate element B is concave (cf. line 6). This variable *count* also allows us to backtrack in the sequence when two or more concave vertices are found in a row.

4. Complexity analysis

Let us now proceed to the time complexity analysis of the TORCH algorithm (cf. Section 3). The complexity analysis is done step-by-step as follows:

- Step 1 performs a *lexicographical sorting* using the introsort algorithm, which is known to have time complexity of $\mathcal{O}(n \log n)$ [19].
- Step 2 performs two accesses to the sorted array $\mathbb{P}$; more specifically, it retrieves the first and the last elements of $\mathbb{P}$, i.e., the west and east points.
- Step 3 calculates the south and north poles by scanning $\mathbb{P}$ once; so the time complexity of this step is linear, that is, $\mathcal{O}(n)$.
- Step 4 calculates each lateral hull using the principle of a drop of water rolling down the hill, which is well-known in the watershed algorithms in image processing [20]. Note that we examine each point of the sequence only once, so this step has

Table 1
Time performance (s) of Graham's, Andrew's, Chan's, Jarvis', Quickhull, and TORCH algorithms for large, random point multisets in quad domains.

n	Jarvis	Graham	Chan	Andrew	Quickhull	TORCH
1,000,000	0.329722	0.307002	0.299087	0.165532	0.145762	0.112451
2,000,000	0.734261	0.638333	0.622929	0.345677	0.290111	0.236948
4,000,000	1.392391	1.328317	1.298776	0.720534	0.580402	0.484930
8,000,000	3.458807	2.751630	2.695643	1.504804	1.158312	1.021565
16,000,000	7.663690	5.710389	5.616076	3.200170	2.341315	2.137340
32,000,000	13.542048	11.850699	11.696112	6.665556	4.882123	4.399567
64,000,000	27.679890	24.366206	24.492884	13.744016	9.715212	9.075017
128,000,000	58.923422	50.388065	52.194574	30.575382	22.944189	19.018568

$\mathcal{O}(n)$ time complexity in finding the subsequences of minima of lateral hulls, discarding at the same time most of the interior points. The worst case happens when the minimum of each lateral sequence is one of its endpoints, reaching then its complexity a maximum of $\mathcal{O}(3n/2)$.

- Step 5 concatenates the four lateral sequences of minima into a single one counterclockwise. Assuming that these four sequences have the same number $h/4$ of vertices on average, that is, h is the total number of vertices belonging to the approximate convex hull $\mathbb{A}$, we conclude that appending each vertex of the last three sequences to the first sequence in a row is a procedure with asymptotic linear complexity, that is, its time complexity is $\mathcal{O}(h)$, even considering that only $3h/4$ vertices will be examined on average, not the vertices of the first sequence.
- Step 6 tends to examine the vertices of the approximate convex hull $\mathbb{A}$ once (i.e., h) because of the single loop *for* instruction in Algorithm 3. Therefore, its asymptotic time complexity is $\mathcal{O}(h)$.

Taking into consideration the previous analysis, we conclude that the TORCH algorithm has $\mathcal{O}(n \log n)$ time complexity, though the last two steps take $\mathcal{O}(h)$. In terms of storage, the complexity is then $\mathcal{O}(n + h) = \mathcal{O}(n)$, i.e., it is asymptotically linear.

5. Experimental results

The TORCH algorithm was implemented using the C++ programming model. No multi-threading or parallel programming assets (e.g., CUDA by NVIDIA) were used in our implementation. All the experiments were conducted on a MacPro with an 2x2.66 GHz Quand-Core Intel Xeon CPU, 8 GB 1066 MHz of DDR3, running the Mac OS X operating system, version 10.9.3.

5.1. Benchmarking algorithms and datasets

For testing, TORCH was compared to five competitor algorithms. The first three are input-sensitive algorithms, and are the following:

- *Graham scan algorithm* [14]. We implemented this algorithm in C++ without computing angles explicitly, as explained at http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan.
- *Andrew's monotone chain algorithm* [16]. We also coded this algorithm in C++.
- *Quickhull algorithm* [17]. We did not implement this algorithm. Instead, we use its original version taken from http://www.qhull.org.

The time complexity of the aforementioned three algorithms is $\mathcal{O}(n \log n)$, where n denotes the number of input points. Additionally, we compared TORCH with two output-sensitive algorithms, namely:

- *Jarvis' gift wrapping algorithm* [15]. The time complexity of this algorithm is $\mathcal{O}(n\,h)$, where h is the number of extreme points of convex hull $\mathbb{H}$. For speed purposes, we also implemented this algorithm in C++ without computing angles explicitly.
- *Chan algorithm* [21]. The time complexity of this algorithm is $\mathcal{O}(n \log h)$. As known, this algorithm combines Graham's algorithm together with Jarvis' algorithm. The former is applied to each equally-sized subset of a partition of the set of input points, while the latter operates on the extreme points of sub-hulls as a whole. We also coded this algorithm in C++.

In regard to datasets, we carried out a number of experiments using three families of point multisets: random point multisets within quads, random point multisets within circles, and definite point multisets. Each point is herein defined as a pair of floating-point numbers. Recall that, unlike a set, a multiset is a collection that admits duplicates in its elements.

5.2. Random point multisets in quads

We randomly generated point multisets in the quad domain $D = [0.0, 0.0] \times [100.0, 100.0]$ with an increasing number of points, as shown in Table 1. We started with multisets with 1 million points (1×10^6) and finished with 128 million points (128×10^6). For each multiset in Table 1, we evaluated the average time for 10 reps (or repetitions), that is, the number of times we run the program for multisets with the same number of points. These point sets are publicly available at http://github.com/mosqueteer/TORCH/quadsets/.

As can be observed, Jarvis', Graham's, and Chan's algorithms are capable of determining the convex hull for 3 million points within the time window of one second, while Andrew's, Quickhull, and TORCH algorithms calculate the convex hull of 6, 7, and 8 million points, respectively, within the same time window. That is, TORCH outperforms Quickhull (i.e., the fastest algorithm among the others above) with a *point speedup* of about 1.15×, approximately, for the same time window of one second, i.e., TORCH copes with 1.15 times more points than Quickhull, per time unit. In terms of time performance, TORCH is 1.17× faster than Quickhull, approximately, for the same amount of points. As expected, the convex hull of each point multiset listed in Table 1 consists of a number of extreme points in the interval [34, 53], regardless of the algorithm used, what is explained by the massive density of points in the square domain D.

The time results listed in Table 1 are graphically expressed in Fig. 3. Although the theoretical complexity of the six benchmarking algorithms in Table 1 is not linear (cf. Section 4), we observe from Fig. 3 that their experimental complexity time seems to be linear. Nevertheless, as shown in Fig. 3(b), TORCH is clearly faster than any of the other five benchmarking algorithms listed above.

5.3. Random point multisets in circles

We also randomly generated point multisets in the circle domain D inscribed in the quad $[0.0, 0.0] \times [100.0, 100.0]$, as shown in Table 2. Basically, we proceeded in the same manner as above, but we used circles, rather than quads, with millions of points. These circular point sets are publicly available at http://github.com/mosqueteer/TORCH/circlesets/.

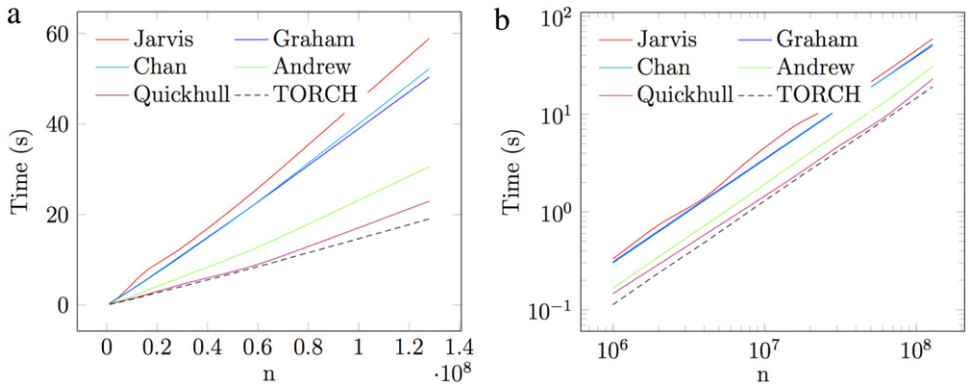

Fig. 3. Experimental time complexity of Graham's, Andrew's, Quickhull and TORCH algorithms (with reference to Table 1) for large, random point sets in quad domains: (a) linear–linear scale; (b) log–log scale.

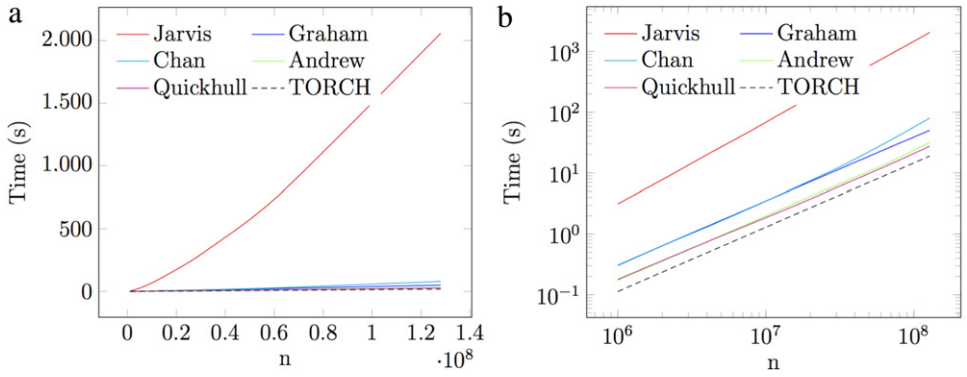

Fig. 4. Experimental time complexity of Graham's, Andrew's, Quickhull and TORCH algorithms (with reference to Table 1) for large, random point sets in circle domains: (a) linear–linear scale; (b) log–log scale.

Table 2
Time performance (s) of Graham's, Andrew's, Chan's, Jarvis', Quickhull, and TORCH algorithms for large, random point multisets in circle domains.

n	Jarvis	Graham	Chan	Andrew	Quickhull	TORCH
1,000,000	3.136980	0.306178	0.302420	0.174327	0.178834	0.112998
2,000,000	7.782843	0.641630	0.640255	0.359685	0.365521	0.236446
4,000,000	19.911608	1.333837	1.305911	0.743585	0.741249	0.490542
8,000,000	50.224761	2.756475	2.731776	1.566118	1.491341	1.020428
16,000,000	130.301081	5.702252	5.827951	3.281981	2.985678	2.139037
32,000,000	322.259015	11.798317	12.804678	6.758322	6.151319	4.418132
64,000,000	805.303988	24.410642	30.874927	13.847511	12.718213	9.112764
128,000,000	2058.367112	50.317197	80.086781	31.656362	27.384132	18.934334

Table 3
Time performance (s) of Graham's, Andrew's, Chan's, Jarvis', Quickhull, and TORCH algorithms for small, definite point multisets.

Point set	*n*	Jarvis	Graham	Chan	Andrew	Quickhull	TORCH
Airplane	2,919	0.000755	0.000567	0.000561	0.000415	0.000459	0.000219
Al Capone	3,618	0.001282	0.000701	0.000718	0.000488	0.000659	0.000284
Vessel	4,041	0.000785	0.000765	0.000793	0.000501	0.000679	0.000288
Formica	8,718	0.001828	0.001850	0.001872	0.001228	0.001245	0.000670
T800 Head	35,925	0.013727	0.007954	0.007662	0.005133	0.004585	0.002533
Bugatti	320,119	0.236878	0.078228	0.078132	0.042651	0.041861	0.022919

A brief glance at Table 2 shows that Jarvis' algorithm is not capable of finding the convex hull for 1 million points within the time window of one second. In contrast, Graham's, Chan's, Andrew's, Quickhull, and TORCH algorithms calculate the convex hull of 3, 3, 6, 6, and 8 million points, respectively, within the same time window. Thus, TORCH performs better than Quickhull with a *point speedup* of about 1.33×, approximately, for the same time window of one second, i.e., TORCH processes 1.33 times more points than Quickhull, per time unit.

Also, TORCH outperforms Quickhull with a *time speedup* of about 1.46×, on average, for the same amount of points. Note that the convex hull of each point multiset listed in Table 2 has a number of extreme points in the interval [340, 1732], independently of the algorithm used. In general, more input points means more output or extreme points.

By comparing the results shown in Tables 1 and 2, we see that the time performance of Graham's, Andrew's, and TORCH algorithms are invariant to the type of point set in practice. Even so, we

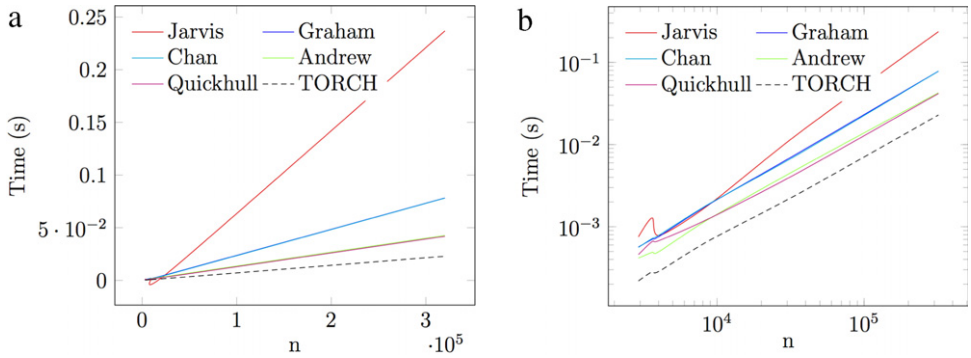

Fig. 5. Experimental time complexity of Graham's, Andrew's, Quickhull and TORCH algorithms (with reference to Table 3) for small, definite point sets: (a) linear–linear scale; (b) log–log scale.

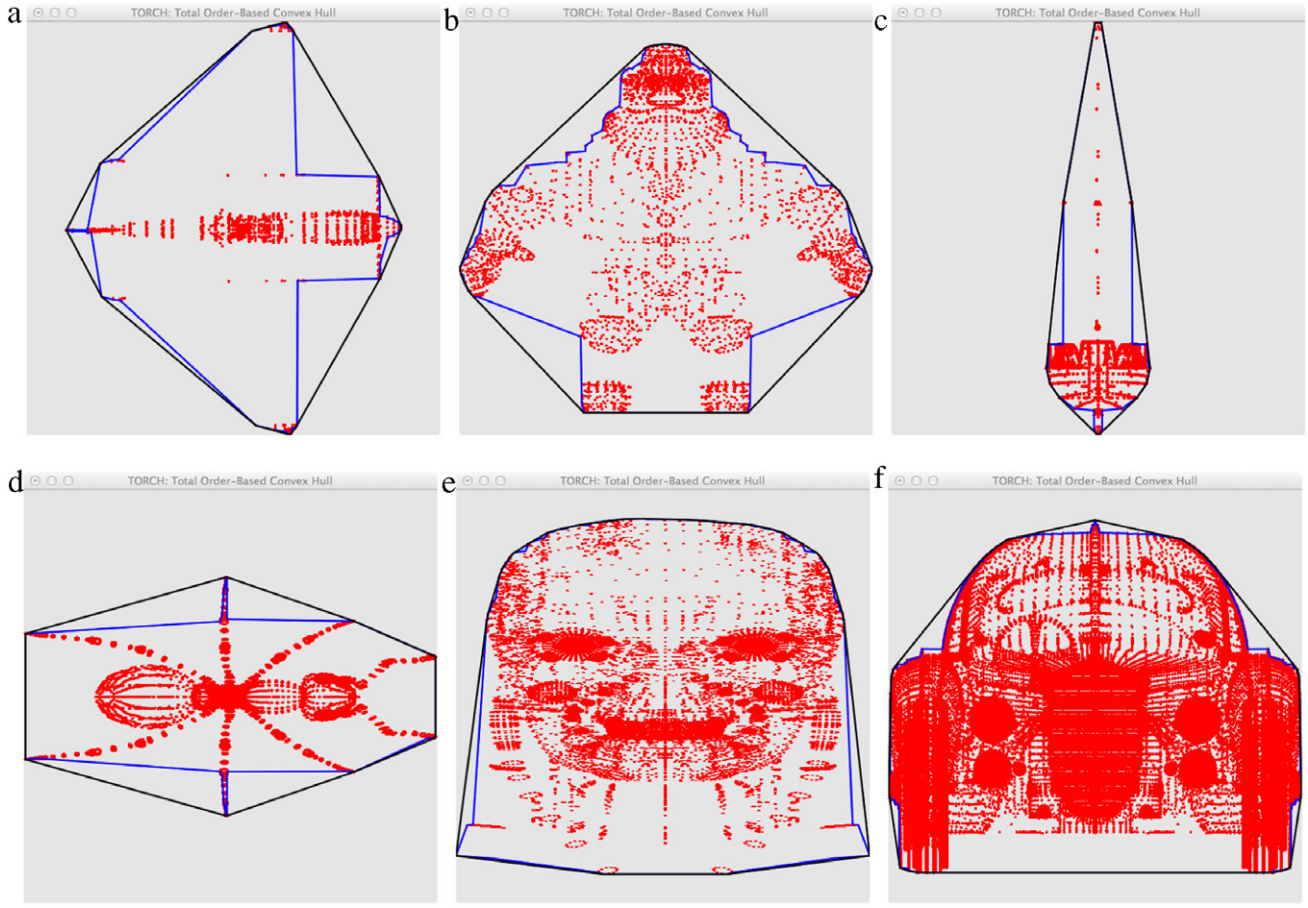

Fig. 6. Convex hulls (in black) partially overlapping approximate convex hulls (in blue): (a) Airplane's point set; (b) Al Capone's point set; (c) Vessel's point set; (d) Formica's point set; (e) T800 head point set; (f) Bugatti's point set. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

note that Graham's and TORCH are slightly faster for circle point sets than for quad point sets; on the contrary, Andrew's is slightly faster for quads.

In regard to Quickhull, it is clearly sensitive to the shape of the input point set, because its time performance degrades when we use circle point sets rather than quad point sets with the same size. This inferior time performance is more noticeable when one uses Jarvis' algorithm, as shown in Fig. 4, what is explained by the fact that its typical wrapping process is more laborious for round shapes; this problem can be mitigated by previously sorting the points in x direction. Chan's algorithm also works worse for circle point sets than for quad point sets because it results from the combination of Jarvis' and Graham's algorithms.

5.4. Definite point multisets

We also used small point multisets concerning specific 2D objects, as those depicted in Fig. 6, which are publicly available at http://github.com/mosqueteer/TORCH/definitesets/. Each one of these multisets was produced from the projection of the 3D mesh vertices (in OBJ format) onto one of the Cartesian planes (e.g., xy-plane defined by $z = 0$). The contours in black represent the convex hulls of such objects. As shown in Fig. 6, these black contours partially overlap the corresponding approximate convex hulls in blue.

The time results for the points sets depicted in Fig. 6 are listed in Table 3, and graphically represented in Fig. 5. Let us advance

that the number of extreme vertices of each convex hull does not exceed 80 in any case. Besides, after a brief glance at the linear–linear graph and log–log-graph shown in Fig. 5(a) and (b), respectively, we note that TORCH also outperforms the classic algorithms in the computation of the convex hull of any small, definite point set, not matter whether they are input-sensitive or output-sensitive.

6. Conclusions

This paper has proposed a sorting-based algorithm to compute the convex hull (and also the approximate convex hull) of a set of points in $\mathbb{R}^2$. This algorithm, named TORCH, outperforms Quickhull. This is important because Quickhull is known as the fastest algorithm we find in the literature. But, more importantly, it is the simplicity of the algorithm, which is essentially due to sorting operations. The only geometric computations done by the algorithm are those concerning the process of convexification (or inflating) of the approximate convex hull $\mathbb{A}$ into the convex hull $\mathbb{H}$. TORCH source code (including input point sets) is publicly available at http://github.com/mosqueteer/TORCH/.

In short, the algorithm performs inexpensive sorting operations to find $\mathbb{A}$, and expensive geometric operations to find $\mathbb{H}$ from $\mathbb{A}$. The speed of the algorithm is explained by the fact that the number of geometric operations is negligible compared to the number of sorting operations carried out by the algorithm. In the near future, we intend to design an even faster algorithm without using the first step of the algorithm relative to sorting in the x direction. Based on the same leading idea of the algorithm herein described, we can generalize TORCH to higher dimensions, without using the preliminary sorting of the input point set.

Acknowledgments

The author is very grateful to anonymous reviewers for their valuable questions and suggestions, which contributed to significantly improve the paper. This work was supported by FCT (Fundação para a Ciência e Tecnologia) project UID/EEA/50008/2013.

References

[1] Preparata FP, Hong SJ. Convex hulls of finite sets of points in two and three dimensions. Commun ACM 1977;20(2):87–93.
[2] Srungarapu S, Reddy D, Kothapalli K, Narayanan P. Fast two dimensional convex hull on the GPU. In: Proceedings of the 25th IEEE international conference on advanced information networking and applications. (AINA'11), IEEE Press; 2011. p. 7–12.
[3] Goldluecke B, Cremers D. Convex relaxation for multilabel problems with product label spaces. In: Daniilidis K, Maragos P, Paragios N, editors. Proceedings of the 11th European conference on computer vision. (ECCV'10), Lecture notes in computer science, vol. 6315. Berlin (Heidelberg): Springer; 2010. p. 225–38.
[4] Toussaint G, Foulsen R. Some new algorithms and software implementation methods for pattern recognition research. In: Proceedings of the 3rd IEEE international computer software and applications conference. (COMPSAC'79), IEEE Press; 1979. p. 55–8.
[5] Liu-Yu S, Thonnat M. Description of object shapes by apparent boundary and convex hull. Pattern Recognit 1993;26(1):95–107.
[6] Hahn H, Han Y. Recognition of 3D object using attributed relation graph of silhouette's extended convex hull. In: Proceedings of the 2nd international conference on advances in visual computing. (ISVC'06), Lecture notes in computer science, vol. 4292. Springer-Verlag; 2006. p. 126–35.
[7] Sherali HD, Smith J, Selim SZ. Convex hull representations of models for computing collisions between multiple bodies. European J Oper Res 2001; 135(3):514–26.
[8] Okada K, Inaba M, Inoue H. Walking navigation system of humanoid robot using stereo vision based floor recognition and path planning with multi-layered body image. In: Proceedings of the 2003 IEEE/RSJ international conference on intelligent robots and systems, Vol. 3. (IROS'03), IEEE Press; 2003. p. 2155–60.
[9] Strandberg M. Robot path planning: An object-oriented approach (Ph.D. thesis), Stockholm (Sweden): Automatic Control, Department of Signals, Sensors and Systems, Royal Institute of Technology (KTH); 2004.
[10] Fuentes O. Automatic determination of stellar atmospheric parameters using neural networks and instance-based machine learning. Exp Astron 2001; 12(1):21–31.
[11] Amundson NR, Caboussat A, He J, Seinfeld JH. An optimization problem related to the modeling of atmospheric organic aerosols. C R Math 2005;340(10): 765–8.
[12] Wang Y, Wu L-Y, Zhang J-H, Zhan Z-W, Zhang X-S, Chen L. Evaluating protein similarity from coarse structures. IEEE/ACM Trans Comput Biol Bioinf 2009; 6(4):583–93.
[13] Avis D, Bremner D, Seidel R. How good are convex hull algorithms? Comput Geom 1997;7(5–6):265–301.
[14] Graham RL. An efficient algorithm for determining the convex hull of a finite planar set. Inform Process Lett 1972;1(4):132–3.
[15] Jarvis RA. On the identification of the convex hull of a finite set of points in the plane. Inform Process Lett 1973;2(1):18–21.
[16] Andrew AM. Another efficient algorithm for convex hulls in two dimensions. Inform Process Lett 1979;9:216–9.
[17] Barber CB, Dobkin DP, Huhdanpaa H. The quickhull algorithm for convex hulls. ACM Trans Math Softw 1996;22(4):469–83.
[18] Kallay M. The complexity of incremental convex hull algorithms in $\mathbb{R}^d$. Inform Process Lett 1984;19(4):197.
[19] Musser D. Introspective sorting and selection algorithms. Softw - Pract Exp 1997;27(8):983–93.
[20] Gonzalez R, Woods R. Digital image processing. New Jersey (USA): Prentice Hall, Inc.; 2008.
[21] Chan T. Output-sensitive results on convex hulls, extreme points, and related problems. Discrete Comput Geom 1996;16(4):369–87.