How to compute the convex hull of a binary shape? A real-time algorithm to compute the convex hull of a binary shape.

Jonathan Fabrizio

Received: 04 April 2023 / Accepted: 21 August 2023

Abstract

In this article, we present an algorithm to compute the convex hull of a binary shape. Efficient algorithms to compute the convex hull of a set of points had been proposed long time ago. For a binary shape, the common practice is to rely on one of them: to compute the convex hull of binary shape, all pixels of the shape are first listed, and then the convex hull is computed on this list of points. The computed convex hull is finally rasterized to provide the final result (as, for example, in the famous scikit-image library [6]). To compute the convex hull of an arbitrary set of points, the points of the list that lie on the outline of the convex hull must be selected (to simplify, we call these points "*extrema*"). To find them, for an arbitrary set of points, it is necessary to browse all the points but not in the particular case of a binary shape. In this specific situation, the extrema necessarily belong to the inner boundary of the shape. It is a waste of time to browse all the pixels as it is possible to discard most of them when we search for these extrema. Based on this analysis, we propose a new method to compute the convex hull dedicated to binary shapes. This method browses as few pixels as possible to select a small subset of boundary pixels. Then it deduces the convex hull only from this subset. As the size of the subset is very small, the convex hull is computed in real time. We compare it with the commonly used methods and common functions from libraries to prove that our approach is faster. This comparison shows that, for a very small shape, the difference is acceptable, but when the area of the shape grows, this difference becomes significant. This leads us to conclude that substituting current functions to compute convex hull of binary shapes with our algorithm in frequently used libraries would lead to a great improvement.

Keywords: convex hull - binary shape - real-time

Jonathan Fabrizio ORCID
LRE, EPITA,
14-16, rue Voltaire, 94270 Le Kremlin-Bicêtre, France
Tel.: +33 1 53 14 59 40
E-mail: jonathan.fabrizio@lrde.epita.fr

1 Introduction

Convex hull computation is a major task. It is used in various situations in 2D as well as in 3D (and even more), and most of the time this task requires efficiency. It is the cornerstone of many algorithms [28] like in [24, 37, 36] and inspires many models and solutions [38, 15].

Several solutions have already existed for a long time, and these solutions are still up to date. Current research does not have proposed a new method that has defeated these old algorithms (for mono-thread approaches); they remain reference algorithms, even if there are some tries [25, 7]. Anyway, research in this topic is still active, but the vast majority of the works focus on specific cases. With the growing usage of GPU, current research works focus also on parallel algorithms and algorithms efficient on GPU. However, for the simple case of a binary shape in image processing, the common strategy is still to list all the pixels of the shape, to compute the convex hull using an efficient algorithm and then to go back to the image to fill in and rasterize the computed convex hull. It is the choice of popular tools such as the famous scikit-image library [6]: the function `kimage.morphology.convex_hull_image` lists all the pixels in the shape and then computes the convex hull using `scipy.spatial.ConvexHull` which relies on the Qhull library [5]. MATLAB [2] has a similar strategy with its function `bwconvhull`. It is the custom for this specific case.

Based on this analysis, let's have a look to the reference algorithms able to compute the convex hull of a set of points. The keystone of these algorithms is to be able to select in the set of points, as fast as possible, the points that belong to the boundary of the convex hull. (To simplify, we call these points "*extrema*" or "*extreme points*.")

In the following, we will note n the number of input points and h the number of extreme points.

A reference algorithm to compute the convex hull of a set of points is the Graham's algorithm [26]. It is the choice of the Pylene library, for example, [4]. There are some variants of this algorithm [10]. It is also used as a step in more complex algorithms [16]. Graham's scan starts by searching the lowest y-coordinate point P. Then, all points P' are sorted according to the angle

between the line (PP') and the x-axis. All points are successively visited in that order to determine whether they belong to the boundary of the convex hull or not. This leads to require $O(n \log n)$ time. In [10], instead of sorting points according to the angle, Andrew proposes to sort points according to the x-coordinates of points. He also proposes to determine the convex hull in two steps: the upper part and the lower part.

Another algorithm is the *QuickHull* algorithm [12]. A well-known implementation is provided in the Qhull library [5]. A specific version of this algorithm for 2D spaces has been exposed by Eddy in [18]. A non-recursive version of this variation has been exposed by Bykat in [14]. We then expect that Bykat's version be faster than Eddy's version even if they both require $O(nh)$ time in the worst case. The QuickHull algorithm starts by finding the leftmost and the rightmost points (P_1 and P_2, respectively) in the set of points. These two points belong to the convex hull. They determine a line (P_1P_2) which separates the set of points into two subsets. Then, the algorithm searches for the point P_3 which is the furthest from this line. This point also belongs to the boundary of the convex hull. All points in the triangle $P_1P_2P_3$ can be discarded as they do not belong to the boundary of the convex hull. Then, we iterate over the line (P_2P_3) and so on. The algorithm stops when there are no remaining points.

The algorithm proposed by Akl and Toussaint [8] relies also on a similar strategy to try to discard some points. The idea in 2D is to form a convex quadrilateral using four extreme points and then discard points located inside the quadrilateral. It requires $O(n \log n)$ time.

Another algorithm is the algorithm proposed by Melkman [30]. This algorithm computes the convex hull, while the points are being added. It requires $O(n)$ time.

To finish, we can cite the classical Jarvis march [27] (gift wrapping algorithm) which requires $O(nh)$ time. The idea is to start from the leftmost point P (which obviously belongs to the outline of the convex hull). Then, the idea is to browse all other points in order to find the leftmost point P' for which there is no point on the left of $[PP']$. This idea is repeated until we go back to point P in order to wrap all other points. This march is used in other algorithms like in Chan's algorithm. Chan [16] proposes to compute a partition of the points and compute the convex hull of each subset (with Graham algorithm for example [26]) and then deduce the final convex hull by a Jarvis march over the convex hull of all subsets. This leads to require $O(n \log h)$ time.

All these algorithms are rather old but still up to date. Current research tries to improve the speed by improvement of existing method or by using parallel computing or also focus on specific cases. It is uncommon to propose a new method for the general case.
A usual strategy to save time is to preprocess the input points. Recently Alshamrani et al. [9] propose to improve the speed of convex hull computation for a large set of points by preprocessing points and discarding many of them with a simple test to decrease the number of considered points and then speed up the convex hull computation. After this preprocessing step, the convex hull computation relies on the Graham algorithm [26]. They show that this strategy for a large set of points is faster than the Graham algorithm [26] without preprocessing and also faster than the classical Jarvis march [27]. The idea of preprocessing is also investigated in [31]. Another strategy to improve the speed is to use parallel computing; to do so, current research explores the usage of the GPU [33]. Similar to the work on preprocessing exposed in [9], preprocessing is also investigated [34] on the GPU. Current research also focuses on specific cases or situations. Nguyen et al. have proposed a solution for the specific case of computing the convex hull of a collection of disks [32, 29]. Chan and Chen [17] consider solutions that require limited storage and make only a few passes over the input. Dynamic convex hull has also been investigated [13] which allows to efficiently update the convex hull if we remove or add points in the set of points. Obviously, higher dimensions have been investigated like in [23, 22]. To finish, note that the deep neural network has been used to compute approximation of the convex hull [11].

In this article, we expose an algorithm to compute the convex hull specifically in the case of a binary shape. In this case, we may not need to consider all pixels to compute the convex hull. It is sufficient to browse the points on the boundary of the shape. We propose:

- a way to find extreme points without neither scanning all pixels of the shape nor computing the edge of the shape,
- a new method to compute the convex hull of a binary shape, thanks to our way of selecting extreme points,
- a comparison between our method with other methods from reference libraries and tools like OpenCV [3],

To summary, we propose a new fast strategy to compute the convex hull of a binary shape to use instead of the commonly used algorithms in popular libraries. This algorithm is compatible with real-time applications without relying on a GPU.

Our article is organized as follows: in Sect. 2 we expose all the steps of the algorithm. In Sect. 3 we compare our method with multiple algorithms and in Sect. 4 we conclude.

2 Algorithm

2.1 Overview

We want to compute the convex hull of the binary shape (a set) as illustrated in Fig. 1. The convex hull

$\mathcal{C}$ of the shape S noted Hull(S) is the smallest convex set which includes S. $\mathcal{C}$ is convex; this means:

$$\forall (x, y) \in \mathcal{C}^2 \Rightarrow [x, y] \subset \mathcal{C} \quad (1)$$

In 2D, the convex hull of a set of points is the smallest convex polygon, that encompasses all the points in the set. For a binary shape, it is the smallest convex set that encompasses the binary shape.

To be able to compute this convex hull, we first start by computing the outline of the convex hull (Fig. 1b) of the initial shape (Fig. 1a). To do so, we browse the upper part of the shape to find the potential upper maxima. We do the same with the lower part. Among all detected points, we select those who really belong to the boundary of the convex hull. To finish, we connect all selected points to get the outline of the convex hull, and then we fill in the region to get the complete convex hull (Fig. 1c). Even if the extreme points of the convex hull must lie over the boundary of the shape, we want to avoid computing the boundary of the shape. We try to browse as few pixels as possible to find these extreme points in order to save time.

The complete scheme of the method is illustrated in Fig. 2, and every step is exposed in the following subsections.

2.2 Step-by-step algorithm

We want to compute the convex hull of a binary shape (as in Fig. 2a). We look over every step of our algorithm in the following subsections.

2.2.1 Find left and right boundaries

The first step is to find the first vertical segment of the shape (the leftmost vertical segment) and the last one (the rightmost vertical segment) as illustrated in Fig. 2b. It is important because these two segments obviously belong to the outline of the convex hull. There is no difficulty for this step, assuming we know the bounding box of the shape. You can start from the four corners of the bounding box of the shape and browse pixels vertically until you hit the shape. The four intersection pixels define the two searched segments. The process is summarized in Algorithm 1. Another solution is to go throw the first and last columns to note the first and last encountered points. It is just important to notice that the segment may have holes or may be 1 pixel long (Fig. 3).

Algorithm 1: Find left and right boundaries

Data: p_upper: the point (the coordinates (x, y)) of the upper point of the bounding rectangle of the binary shape, p_lower: the point (the coordinates (x, y)) of the lower point of the bounding rectangle of the binary shape. I: the image

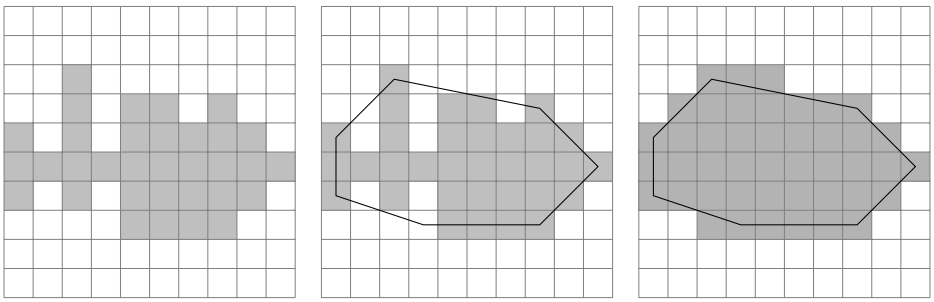
Result: p_1, p_2, p_3, p_4: the 2 extremities of the first and last vertical segments of the shape

$p = p\_upper$;
while $I[p] \neq foreground$ **do**
  $p.y = p.y - 1$
end
$p_1 = p$;
$p = (p\_upper.x, p\_lower.y)$;
while $I[p] \neq foreground$ **do**
  $p.y = p.y + 1$;
end
$p_2 = p$;
$p = p\_lower$;
while $I[p] \neq foreground$ **do**
  $p.y = p.y + 1$;
end
$p_3 = p$;
$p = (p\_lower.x, p\_upper.y)$;
while $I[p] \neq foreground$ **do**
  $p.y = p.y - 1$;
end
$p_4 = p$;
return p_1, p_2, p_3, p_4;

2.2.2 Find upper and lower extrema candidates

The second step is to find points of the shape that belong to the outline of the convex hull. These points belong obviously to the boundary of the shape and are not inside the shape.

For simplicity, we call these points "ascending extrema" (for those who lie on the upper part of the



(a) A binary shape. (b) A binary shape with the convex outline of the shape. (c) The convex hull of the binary shape.

Figure 1: a/ a binary shape, b/ a binary shape with the outline of the convex hull, and c/ the convex hull of this shape (with the outline of the convex hull).

shape) and "descending extrema" (for those who lie on the lower part of the shape).

To find these points, we browse pixels horizontally, from left to right (Fig. 2c), starting from the pixel just above the left most segment of the shape. When we hit a pixel of the shape, we browse pixels vertically until we find the last pixel, and then we add this pixel to a list of candidates. We restart browsing the pixels horizontally, starting above the last detected pixel. We stop when we reach the right boundary (or the top of the bounding box). We show the process in Algorithm 2. We do the same but starting above the rightmost segment. Again we do the same but from right to left (Fig. 2d); Algorithm 2 must be adapted for these cases.

This provides two lists of possible points (the blue one and the magenta one) on the outline of the convex hull, for the upper part of the shape. We do the same with the lower part of the shape (Fig. 2c and 2d). We get also two lists of possible extreme points. It is important to notice that:

- these selected points may or may not belong to the outline, we will have to filter out some of them later,
- an ascending extremum detected from left to right (in blue) cannot be on the right of an ascending extremum detected from right to left (in magenta). It can also have a gap between the last blue and the first magenta. This remark is also valid for descending extrema (as illustrated in Fig. 2d).

Note that we can improve the speed of this step:

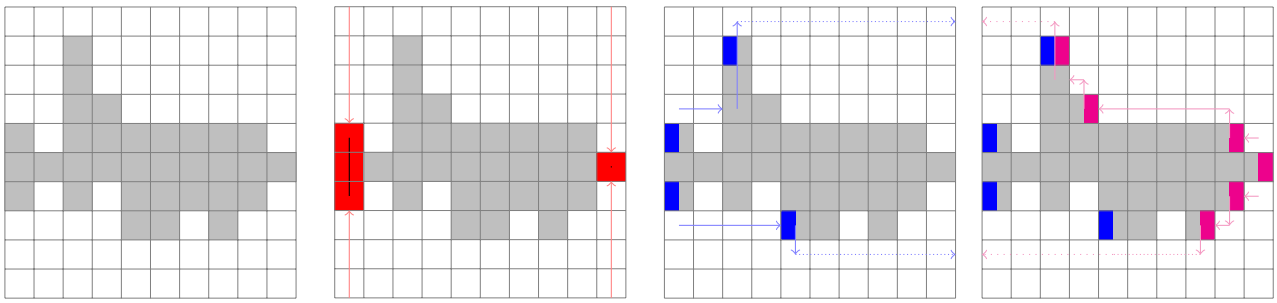
- when searching magenta maxima, we are not obliged to go to the left border; we can stop earlier, when we reach the last blue maxima,
- when reaching the upper bound (the lower bound respectively) of the bounding box, we can stop because this means that we have detected the higher (respectively the lower) point.

In Fig. 2d, the two dotted arrows illustrate the tests that can be simplified and avoided.

Algorithm 2: Find left-to-right upper extrema cadidates

Data: p_upper, p_lower: Coordinates of the lower left and upper right points of the bounding rectangle of the shape. I: the image. p_start: the upper point of the left most segment of the shape.
Result: $candidates\_upper\_lr$: An array with the potential extreme points.

```
p = (p_start.x, p_start.y + 1);
while p.x < p_upper.x do
    p.x = p.x + 1;
    if I[p] ≠ background then
        while I[p] ≠ background do
            p.y = p.y + 1;
            // if (p.y>p_upper.y) break ;
        end
        candidates_upper_lr.append((p.x, p.y − 1));
        // if (p.y>p_upper.y) break
    end
end
return candidates_upper_lr;
```

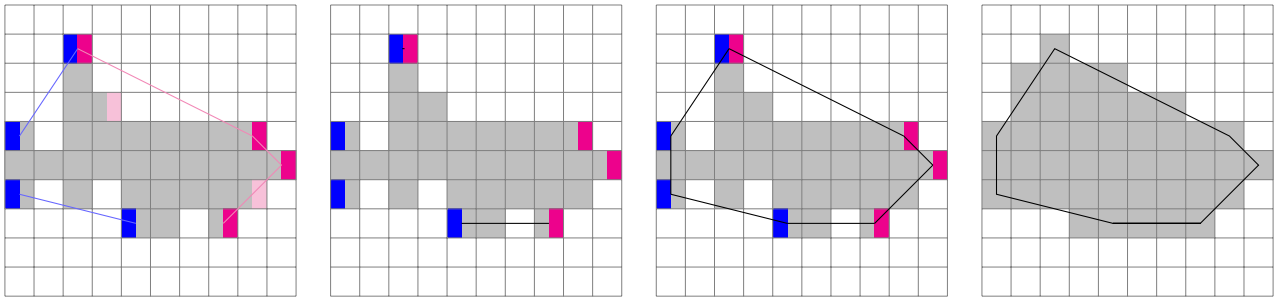


(a) A binary shape.

(b) Find the left and right boundaries.

(c) Find, from left to right, ascending and descending extrema.

(d) Find, from right to left, ascending and descending extrema.



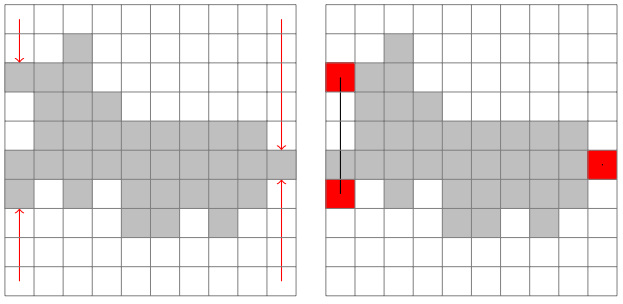
(e) Join extrema and discard extrema not on the convex hull.

(f) Join last left extrema with last right extrema.

(g) Take the union of edges detected in b/ e/ and f/.

(h) Fill in the polygon.

Figure 2: Step-by-step algorithm.



(a) First and last vertical segments detection: The first or the last vertical segments may content holes.

(b) First and last vertical segments detection.

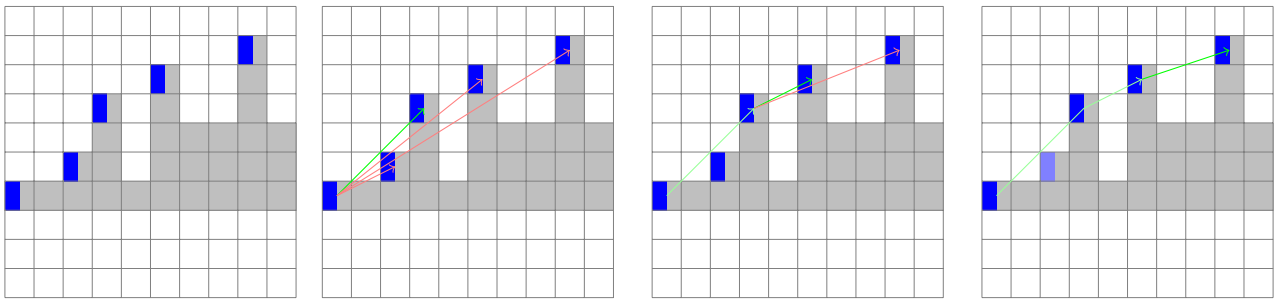
Figure 3: Finding the first and the last vertical segments.

2.2.3 Linking successive ascending extrema (respectively descending extrema)

The goal is to find the outline of the convex hull of the shape (Fig. 2e and 2f). To do so, we have to browse ascending and descending extrema and select the ones that lie on the boundary and discard all the others (Fig. 4). Let's start with the first ascending extrema (the blue ones). We start from the first ascending extremum (the leftmost blue upper extremum) and try to join it with all the following ones in the first list. We keep only the segment with the higher slope (Fig. 4b). Then, we start from the linked extremum and link it with all the remaining following extrema (Fig. 4c). Again, we keep the segment with the higher slope. We keep on processing extrema until reaching the last extremum (Fig. 4d). We discard all unlinked extrema. This process is exposed in Algorithm 3. You have to do the same with ascending extrema detected from right to left (the magenta ones) and also descending extrema (the blue ones and the magenta ones). To finish, we have to link the last ascending extrema detected from left to right (the rightmost blue ascending extrema) with the last detected from right to left (the leftmost magenta ascending extrema) (and respectively the descending extrema) (Fig. 2f).

2.2.4 Filling the shape

Now, we have the correct list of the extreme points (in practice, it is more or less, the concatenation of the



(a) The collection of extrema.

(b) We link the first extrema with all the other and keep the higher slope.

(c) Link the selected extrema with all the following and keep the higher slope.

(d) Loop until the last extremum is linked and discard unlinked extrema.

Figure 4: Link extrema, example with left to right ascending extrema.

Algorithm 3: Linking, from left to right, successive ascending extrema

Data: *candiates_upper_lr*: An array with the potential extreme points.
Result: *extrema_upper_lr*: An array with the extreme points.

```
extrema_upper_lr.append(candiates_upper_lr[0]);

i = 0;
j = i + 1;
while j < size(candiates_upper_lr) do
    selected = j;
    max_slope =
     slope(candiates_upper_lr[i], candiates_upper_lr[j]);

    for j = i + 2 to
     size(extrema_upper_lr) − 1 do
        current_slope =
         slope(candiates_upper_lr[i], candiates_upper_lr[j])
         if current_slope ≥ max_slope then
            selected = j;
            max_slope = current_slope;
        end
    end
    extrema_upper_lr.append(
    candiates_upper_lr[selected]);
    i = selected;
    j = i + 1;
end
returncandiates_upper_lr;
```

four arrays computed by the Algorithm 3). We then have the outline of the shape (Fig. 2g) which is the union of results of steps illustrated in Figs. 2b, 2e and 2f. It remains only to fill in the shape (Fig. 2h).

In our implementation, we keep a track (from previous steps), for each column, of the height of the pixel on the lower outline and the height of the pixel on the higher outline in two separate arrays. We only have to draw a vertical line from the lower pixel to the higher pixel, and we loop on every column.

Note that depending on the architecture, it may be more efficient to draw lines horizontally instead of vertically.

3 Evaluation

In this section, we evaluate our algorithm. To do so, we have automatically extracted multiple shapes and characters from images to create a database (Fig. 5). To do so, we segment and label various pictures and extract randomly some connected component. We store individually each connected component in a file (with only two colors, the shape and the background). The created database contains 21 262 images, each containing only one shape, (i.e., one connected component). To perform a fine comparison, we classify every shape of the database according to the area of the input shape: as the complexity of most algorithms depends on the area of the shape, we have split the database into different subsets according to the area of the shape. The maximum area in this dataset is around 75000 pixels. We have split the image of the database into five different categories:

Category	Area (pixel)
1	< 100
2	≥ 100 and < 500
3	≥ 500 and $< 1\,000$
4	$\geq 1\,000$ and $< 1\,500$
5	$\geq 1\,500$

The database is openly available at https://www.lrde.epita.fr/dload/papers/fabrizio.23.jrtip.tar.bz2

Figure 5: Samples of the evaluation database.

On this database we measure the speed of our algorithm, and we compare with multiple algorithms. We compare our algorithm with the optimized implementation of Graham [26] provided in Pylene [35]. We use various algorithms implemented in the CGAL library [1]: Andrew variant of Graham algorithm [10], Bykat algorithm [14], Akl and Toussaint algorithm [8], Melkman algorithm [30], Eddy [18] and Jarvis algorithm [27]. We also compare with the algorithm Torch presented in [25, 7] and with QHull presented in [12] using the implementation provided in [5]. Finally, we compare with the implementation provided by OpenCV [3]. For OpenCV, we follow the documentation: we compute the contour of the shape, and we compute the convex hull only on these points (however we use a morphological operator to find the inner boundary instead of the suggestion of the documentation to use Canny as it is much faster). We measure the time to process all shapes for every subset and provide the mean time per shape. We measure also the total time (no matter the category), but this measure must be taken very cautiously as our category are unbalanced (it may vary according to the proportion of each category and then depends on the final application). Every time we rank a method, we compute a statistical test between the method and the following one in the ranking to measure the representativeness of the ranking. When the probability is high, we can consider the two following methods as joint. When the probability is low, we can consider the ranking as correct and representative.

We provide two different comparisons. Firstly, we provide the comparison of the time to compute extreme points of the convex hull. It is the natural output of all these algorithms (except ours). Second, as our goal is to compute the convex hull of a binary shape, we also compare the time to compute the convex hull of the binary shape. This means that we add the time to draw to final convex polygon in the measure.

All the measures are performed on a 4.00GHz Intel$^{(R)}$ Core$^{(TM)}$ i7-6700K CPU.

3.1 Time to detect extreme points

To compare the speed of all these algorithms, we compare the time to compute the extrema. It is the initial output of all compared algorithms; then, these algorithms are used without any modification. On the contrary, our algorithm starts from the initial shape but the output is also binary shape. Then, for this comparison we only measure the time of our algorithm when it reaches the step illustrated in Fig. 2e. At this point, we are able to provide a correct list of extrema.

The results are exposed in Tables 1 and 2.

Firstly, we see that our method is faster than all other methods except for very small shapes where Graham implementation in Pylene is faster. The statistical tests validate the ranks of our method.

Second, we notice that our method is much less impacted by the area of the shape. By design, contrary to all others, our method is impacted by the width plus the height of the bounding box of the shape but not the area of the shape - which is a great advantage over all other methods. There is no big difference in time for very small shapes, but there is a gap for big shapes (Table 1). Even more the speed of our method on very big shapes competes with the speed of other methods on very small shapes. It is validated by the standard deviation on the whole database (Table 2). Note that OpenCV is also less affected by the size of the shape as on small shape it is ranked almost last, and on big shapes it is ranked 3rd.

We see also that Melkman's method reaches quickly the second place. By design Bykat's method is faster than Eddy's method; it is consistent with our evaluation. Jarvis big difference in the time spent by Jarvis march over small shapes and over big shapes. The QHull implementation is a bit deceptive. Furthermore, it is not able to manage shape with a simple horizontal line or simple vertical line, so we were obliged to manage this special case.

This first evaluation shows that our method outperforms most other methods simply to find extreme points of the binary shape.

3.2 Time to compute and draw the convex hull

To goal of our method is to compute the convex hull of a binary shape and to compute the binary shape of the convex hull. Other compared methods are not designed for and are not able to compute the final binary shape. To allow the comparison, we have added a simple step: we fill in the convex polygon. Again, we measure the time according to the area of the shapes. All the measures are exposed in Table 3 and in Table 4.

We see that our method outperforms all other methods. The ranking remains globally the same with very slight modifications. Our method reaches the first place even for small regions. The difference with and without drawing the convex hull is smaller for our algorithm compared to Graham implementation by Pylene. Even if the modification is very small, it is enough to change the ranking. The explanation seems to be that the drawing is easier with the extrema provided by our algorithm (two lists of extrema, one for the lower points and one for the higher points), compared to the

Area of the shapes (pixels)	Number of shapes	Method	Rank	Cumulative time (s)	Time per shape in mean (ms)	Student test
< 100	3195	**Graham (Pylene)**	**1**	**0.008075**	**0.002527**	0.000000
		Our	2	0.008827	0.002763	0.000000
		Melkman (CGAL)	3	0.012397	0.003880	0.000000
		Akl Toussaint (CGAL)	4	0.014846	0.004647	0.000000
		Bykat (CGAL)	5	0.016659	0.005214	0.001089
		Graham Andrew (CGAL)	6	0.017180	0.005377	0.000000
		Torch	7	0.019386	0.006068	0.000000
		Eddy (CGAL)	8	0.024843	0.007775	0.000000
		Jarvis (CGAL)	9	0.030710	0.009612	0.000000
		OpenCV	10	0.048496	0.015179	0.000000
		QHull	11	0.113024	0.035375	
≥ 100 and < 500	11031	**Our**	**1**	**0.042670**	**0.003868**	0.000000
		Graham (Pylene)	2	0.096971	0.008791	0.000000
		Melkman (CGAL)	3	0.117720	0.010672	0.000000
		Akl Toussaint (CGAL)	4	0.145559	0.013195	0.000000
		Bykat (CGAL)	5	0.166408	0.015085	0.000061
		Torch	6	0.174380	0.015808	0.000000
		Graham Andrew (CGAL)	7	0.210898	0.019119	0.000000
		Eddy (CGAL)	8	0.282300	0.025592	0.000044
		OpenCV	9	0.301670	0.027347	0.000000
		Jarvis (CGAL)	10	0.503573	0.045651	0.000000
		QHull	11	0.617158	0.055948	
≥ 500 and < 1000	3463	**Our**	**1**	**0.018028**	**0.005206**	0.000000
		Graham (Pylene)	2	0.079673	0.023007	0.612507
		Melkman (CGAL)	3	0.080400	0.023217	0.000000
		Akl Toussaint (CGAL)	4	0.107172	0.030948	0.000000
		Bykat (CGAL)	5	0.116847	0.033742	0.011687
		Torch	6	0.120515	0.034801	0.000000
		OpenCV	7	0.166071	0.047956	0.201157
		Graham Andrew (CGAL)	8	0.170330	0.049186	0.000000
		Eddy (CGAL)	9	0.212502	0.061364	0.000000
		QHull	10	0.319649	0.092304	0.000000
		Jarvis (CGAL)	11	0.461119	0.133156	
≥ 1000 and < 1500	1331	**Our**	**1**	**0.007140**	**0.005364**	0.000000
		Melkman (CGAL)	2	0.048646	0.036548	0.000000
		Graham (Pylene)	3	0.054810	0.041179	0.000000
		Akl Toussaint (CGAL)	4	0.069883	0.052504	0.000051
		Bykat (CGAL)	5	0.072569	0.054522	0.000000
		Torch	6	0.074939	0.056303	0.000000
		OpenCV	7	0.086405	0.064917	0.000000
		Graham Andrew (CGAL)	8	0.113215	0.085060	0.000000
		Eddy (CGAL)	9	0.139875	0.105090	0.000000
		QHull	10	0.176440	0.132562	0.000000
		Jarvis (CGAL)	11	0.330541	0.248341	
≥ 1500	2242	**Our**	**1**	**0.020787**	**0.009272**	0.000000
		Melkman (CGAL)	2	0.242490	0.108158	0.000000
		OpenCV	3	0.341315	0.152237	0.566382
		Graham (Pylene)	4	0.348931	0.155634	0.157416
		Bykat (CGAL)	5	0.368778	0.164486	0.600380
		Akl Toussaint (CGAL)	6	0.376028	0.167720	0.232955
		Torch	7	0.394009	0.175740	0.000000
		Graham Andrew (CGAL)	8	0.666928	0.297470	0.000333
		Eddy (CGAL)	9	0.764354	0.340925	0.983669
		QHull	10	0.764884	0.341162	0.000000
		Jarvis (CGAL)	11	2.229410	0.994385	

Table 1: Time comparison to compute the extreme points of the convex hull according to the size of the input shape. The table provides the rank of the methods according to the speed. The last column provides a statistical test between two rows to ensure that the comparison of successive mean times is relevant or not.

Area of the shapes (pixels)	Number of shapes	Method	Rank	Cumulative time (s)	Time per shape in mean (ms)	Standard deviation	Student test
All	21262	**Our**	**1**	**0.097452**	**0.004583**	0.030156	0.000000
		Melkman (CGAL)	2	0.501653	0.023594	0.052811	0.000000
		Graham (Pylene)	3	0.588460	0.027677	0.086375	0.000000
		Akl Toussaint (CGAL)	4	0.713489	0.033557	0.086441	0.103929
		Bykat (CGAL)	5	0.741261	0.034863	0.079033	0.016195
		Torch	6	0.783229	0.036837	0.089893	0.000000
		OpenCV	7	0.943956	0.044396	0.076572	0.000000
		Graham Andrew (CGAL)	8	1.178551	0.055430	0.155563	0.000000
		Eddy (CGAL)	9	1.423874	0.066968	0.165871	0.000000
		QHull	10	1.991155	0.093649	0.147875	0.000000
		Jarvis (CGAL)	11	3.555353	0.167216	0.576035	

Table 2: Time comparison to compute the extreme points of the convex hull. The table provides the rank of the methods according to the speed. The last column provides a statistical test between two rows to ensure that the comparison of successive mean times is relevant or not.

counterclockwise list of extrema.

Again, as for previous results (Table 1), results exposed in Table 3 show that the area of the input shape impacts the speed of all other algorithms (it is consistent with the complexity of these algorithms). OpenCV is less affected and again for small shapes it is ranked almost last and for big shapes, it is ranked 3rd. Our algorithm is also not so much affected. It is a great advantage in favor of our algorithm.

Furthermore, if we compare results from Table 1 and Table 4 we notice that our method is faster to compute the final binary shape than other methods to compute only extreme points. This proves that the common strategy that consists in providing the list of the points of the shape and applying a generic algorithm to compute the convex hull can be substituted by a much better, specific strategy - as the one we propose in this article.

3.3 Overall comparison

The evaluation shows that our method is:

- much faster than others,
- much more stable according to the area of the input shape.

The order of magnitude of the time spent is lower with our method than with other algorithms and popular libraries (like with the reference library OpenCV). The explanation is quite simple.

- Firstly, other methods are obliged to browse all the pixels, while we browse only a very small subset of the pixels of the bounding rectangle of the shape. Exactly, if the dimension of the bounding rectangle is $wi \times he$, we browse at most $4he + 4wi$ pixels (Algorithm 1 needs $2he$ at most: he for the first vertical line, and again he for the last vertical line, and Algorithm 2 and variations need at most $2he + 4wi$: one instance need to browse the shape horizontally (wi) and in mean half of the height of the shape ($\frac{he}{2}$). So, $2he+4\left(wi + \frac{he}{2}\right) = 4he+4wi$, while others browse $he \times wi$ pixels. With the improvements suggested in Sect. 2.2.2, we can even decrease the maximum number of browsed pixels to $4he + 2wi$ (because one instance browses only $\frac{he}{2} + \frac{wi}{2}$ pixels in mean).
- Second, the other methods have to find the extreme points in the complete sets of pixels of the shape: even if the algorithm to find these extreme points have an acceptable complexity (for example, $O(n \log n)$ with n the number of input points in the shape for the Graham's algorithm), when the size of the shape increase, it is rapidly penalizing. On the contrary, we search for the extreme points in a very tiny subset of the boundary pixels of the shape. Then, even if the Algorithm 3 requires roughly $O(m^2)$ for the selection of these extreme points (Sect. 2.2.3; Fig. 4), the research is not penalizing because m, the number of candidate pixels, is very low. Furthermore, this subset is split into four subsets, reducing again the size of the data to process leading to around $4\left(\frac{m}{4}\right)^2$ instead of m^2 (usually with $m \ll wi$ but in the worst case $m = 2wi$).

The improvement brought by our algorithm is a major improvement in terms of speed. But it is not the only one. Our method also saves memory because, for big shapes we do not need to list all the points in a separate data structure, and we avoid the data duplication. For small shapes, the difference is negligible but for big shapes it is a waste of memory. It is another advantage of our method.

The limitation of our method is that the method is designed to compute the convex hull of one connected component. It is then difficult to compute the convex hull of a set of unconnected shapes. The reason is the way you search for extreme points (Sect. 2.2.2). If they are some holes, we may miss some points when we browse horizontally the pixels. It is possible to overcome this limitation: we could draw lines from one point of the first shape to all other shapes of the set. With the Bresenham's line algorithm, it is not costly. But it is not always possible. We could also modify the way we found the possible extreme points (Sect. 2.2.2) using a strategy similar to the way we found the left

and right boundaries (Sect. 2.2.1). But it is a bit more time consuming.

Another limitation is that it is not easy to implement the search for extreme points in parallel. You can split the search for the candidates and the selection among the candidates into four different process easily but if you want to use more thread, it is more difficult (obviously the drawing is not a problem).

Finally, there are lot of possible improvements (we have already listed some in the description). For example, we may process the shape horizontally or vertically depending on the width and the height of the shape or depending on the architecture of the memory.

4 Conclusion

In this article, we have proposed a new algorithm to compute the convex hull of a binary shape. We have shown that the development and the use of a specific algorithm to compute the convex hull of a binary shape save a lot of time. Using a non-specific algorithm, even if the algorithm is efficient (with a good complexity), may waste a part of the time browsing all the pixels. Unfortunately, it is the current usage in famous libraries (scikit-image, OpenCV, MATLAB, etc.) to rely on a non-specific algorithm. With our algorithm, we are able to consider only a subset of edge points, and we even avoid computing the edge of the shape, and we are able to focus directly on a very small subset of them (which is much faster).

In our evaluation, we have proven that our algorithm is much faster than other libraries and algorithms. We defeat other libraries (like the reference tools OpenCV). We have also shown that the impact of the area of the shape is high on all other algorithms while our executing time remains low even on big shapes. Our algorithm is then very stable against the area of the shape. Our algorithm is then perfectly compatible with a real-time application.

Furthermore, the list of extreme points is ordered in a clever way, facilitating the polygon filling step as we have the ordered list of lower extrema and the ordered list of higher extrema, which simplifies the final drawing. Substituting the current strategy for computing the convex hull of binary shape in the common tools with this new one would save a lot of time in many applications. The only restriction of this strategy is that it is expected to have only one connected component in the binary shape (if they are not aligned).

This algorithm has been successfully used in different applications: for skew estimation in [19] and text detection in [20, 21].

This article is the opportunity to encourage library maintainers to use dedicated algorithms for convex hull of a binary shape. Even if our algorithm is somewhat naive, we do believe, and we have proven that substituting common algorithms with our algorithm in frequently used and popular libraries would have a great impact on the efficiency of these libraries and then on a lot of software programs.

The computation of the convex hull of a binary shape is important. We have improved this computation, but with the growing usage of GPUs it is now important to work on a GPU implementation of such algorithm (i.e., an algorithm dedicated to binary shape, able to focus on a subset of outer pixels) on a GPU. Also, we have to be able to manage efficiently shapes split in multiple connected components.

References

[1] Cgal: The computational geometry algorithms library. www.cgal.org.

[2] Matlab. www.mathworks.com.

[3] Opencv. opencv.org.

[4] Pylene. gitlab.lrde.epita.fr/olena/pylene.

[5] Qhull. www.qhull.org.

[6] scikit-image: Image processing in python. scikit-image.org.

[7] Torch implementation. https://github.com/mosqueteer/TORCH/.

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

[9] Reham Alshamrani, Fatimah Alshehri, and Heba Kurdi. A preprocessing technique for fast convex hull computation. *Procedia Computer Science*, 170:317–324, 2020.

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

[11] Randall Balestriero, Zichao Wang, and Richard G. Baraniuk. Deephull: Fast convex hull approximation in high dimensions. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3888–3892, 2022.

[12] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, dec 1996.

[13] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 617–626. IEEE, 2002.

[14] Alex Bykat. Convex hull of a finite set of points in two dimensions. *Inf. Process. Lett.*, 7:296–298, 1978.

[15] Hakan Cevikalp, Hasan Serhan Yavuz, and Bill Triggs. Face recognition based on videos by using convex hulls. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(12):4481–4495, 2020.

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

[17] Timothy M Chan and Eric Y Chen. Multi-pass geometric algorithms. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 180–189, 2005.

Area of the shapes (pixels)	Number of shapes	Method	Rank	Cumulative time (s)	Time per shape in mean (ms)	Student test
< 100	3195	**Our**	**1**	**0.008573**	**0.002683**	0.000000
		Graham (Pylene)	2	0.010530	0.003296	0.000000
		Melkman (CGAL)	3	0.015161	0.004745	0.000000
		Graham Andrew (CGAL)	4	0.019004	0.005948	0.814418
		Bykat (CGAL)	5	0.019043	0.005960	0.299884
		Akl Toussaint (CGAL)	6	0.019217	0.006015	0.000000
		Torch	7	0.022776	0.007129	0.000000
		Eddy (CGAL)	8	0.027531	0.008617	0.000000
		Jarvis (CGAL)	9	0.033126	0.010368	0.000000
		OpenCV	10	0.052473	0.016424	0.000000
		QHull	11	0.123898	0.038779	
≥ 100 and < 500	11031	**Our**	**1**	**0.052781**	**0.004785**	0.000000
		Graham (Pylene)	2	0.116287	0.010542	0.000000
		Melkman (CGAL)	3	0.139475	0.012644	0.000000
		Bykat (CGAL)	4	0.183351	0.016621	0.612101
		Akl Toussaint (CGAL)	5	0.184146	0.016693	0.753503
		Graham Andrew (CGAL)	6	0.184810	0.016754	0.000000
		Torch	7	0.200652	0.018190	0.000000
		Eddy (CGAL)	8	0.302518	0.027424	0.000086
		OpenCV	9	0.320546	0.029059	0.000000
		Jarvis (CGAL)	10	0.520799	0.047212	0.000000
		QHull	11	0.678900	0.061545	
≥ 500 and < 1000	3463	**Our**	**1**	**0.025994**	**0.007506**	0.000000
		Graham (Pylene)	2	0.091312	0.026368	0.306697
		Melkman (CGAL)	3	0.093238	0.026924	0.000000
		Bykat (CGAL)	4	0.126536	0.036539	0.974724
		Akl Toussaint (CGAL)	5	0.126572	0.036550	0.269309
		Graham Andrew (CGAL)	6	0.128301	0.037049	0.000054
		Torch	7	0.135977	0.039266	0.000000
		OpenCV	8	0.173518	0.050106	0.000000
		Eddy (CGAL)	9	0.226117	0.065295	0.000000
		QHull	10	0.347952	0.100477	0.000000
		Jarvis (CGAL)	11	0.470653	0.135909	
≥ 1000 and < 1500	1331	**Our**	**1**	**0.012065**	**0.009064**	0.000000
		Melkman (CGAL)	2	0.055846	0.041958	0.000000
		Graham (Pylene)	3	0.060791	0.045673	0.000000
		Bykat (CGAL)	4	0.078072	0.058657	0.766640
		Akl Toussaint (CGAL)	5	0.078206	0.058757	0.250616
		Graham Andrew (CGAL)	6	0.078726	0.059148	0.000000
		Torch	7	0.083490	0.062728	0.000000
		OpenCV	8	0.089761	0.067439	0.000000
		Eddy (CGAL)	9	0.147521	0.110835	0.000000
		QHull	10	0.190199	0.142899	0.000000
		Jarvis (CGAL)	11	0.335468	0.252042	
≥ 1500	2242	**Our**	**1**	**0.044210**	**0.019719**	0.000000
		Melkman (CGAL)	2	0.272068	0.121351	0.000000
		OpenCV	3	0.348798	0.155575	0.045627
		Graham (Pylene)	4	0.375958	0.167689	0.392168
		Bykat (CGAL)	5	0.388438	0.173255	0.957988
		Akl Toussaint (CGAL)	6	0.389142	0.173569	0.667959
		Graham Andrew (CGAL)	7	0.394949	0.176159	0.012750
		Torch	8	0.432974	0.193119	0.000000
		Eddy (CGAL)	9	0.795824	0.354962	0.581521
		QHull	10	0.810618	0.361560	0.000000
		Jarvis (CGAL)	11	2.247249	1.002341	

Table 3: Time comparison to compute the binary shape of the convex hull according to the size of the input shape. The table provides the rank of the methods according to the speed. The last column provides a statistical test between two rows to ensure that the comparison of successive mean times is relevant or not.

Area of the shapes (pixels)	Number of shapes	Method	Rank	Cumulative time (s)	Time per shape in mean (ms)	Standard deviation	Student test
All	21262	**Our**	**1**	**0.143623**	**0.006755**	0.031462	0.000000
		Melkman (CGAL)	2	0.575787	0.027081	0.058348	0.000001
		Graham (Pylene)	3	0.654878	0.030800	0.091449	0.000000
		Bykat (CGAL)	4	0.795440	0.037411	0.081442	0.912755
		Akl Toussaint (CGAL)	5	0.797283	0.037498	0.081674	0.618605
		Graham Andrew (CGAL)	6	0.805790	0.037898	0.084040	0.000207
		Torch	7	0.875869	0.041194	0.098539	0.000000
		OpenCV	8	0.985096	0.046331	0.075707	0.000000
		Eddy (CGAL)	9	1.499512	0.070525	0.171304	0.000000
		QHull	10	2.151566	0.101193	0.155653	0.000000
		Jarvis (CGAL)	11	3.607296	0.169659	0.578666	

Table 4: Time comparison to compute the binary shape of the convex hull. The table provides the rank of the methods according to the speed. The last column provides a statistical test between two rows to ensure that the comparison of successive mean times is relevant or not.

[18] William F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3(4):398–403, dec 1977.

[19] Jonathan Fabrizio. A precise skew estimation algorithm for document images using knn clustering and fourier transform. *IEEE International Conference on Image Processing*, pages 2585–2588, 2014.

[20] Jonathan Fabrizio, Matthieu Cord, and Beatriz Marcotegui. Text extraction from street level images. In *City Models, Roads and Traffic (ISPRS Workshop - CMRT09)*, Paris, France, 2009.

[21] Jonathan Fabrizio, Myriam Robert-Seidowsky, Séverine Dubuisson, Stefania Calarasanu, and Raphaël Boissel. Textcatcher: a method to detect curved and challenging text in natural scenes. *International Journal on Document Analysis and Recognition (IJDAR)*, 19:99–117, 2016.

[22] Mingcen Gao, Thanh-Tung Cao, Ashwin Nanjappa, Tiow-Seng Tan, and Zhiyong Huang. ghull: A gpu algorithm for 3d convex hull. *ACM Transactions on Mathematical Software (TOMS)*, 40(1):1–19, 2013.

[23] Mingcen Gao, Thanh-Tung Cao, Tiow-Seng Tan, and Zhiyong Huang. Flip-flop: convex hull construction via star-shaped polyhedron in 3d. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 45–54, 2013.

[24] Thomas Giorginis, Stefanos Ougiaroglou, Georgios Evangelidis, and Dimitris A Dervos. Fast data reduction by space partitioning via convex hull and mbr computation. *Pattern Recognition*, 126:108553, 2022.

[25] Abel JP Gomes. A total order heuristic-based convex hull algorithm for points in the plane. *Computer-Aided Design*, 70:153–160, 2016.

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

[27] 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.

[28] MA Jayaram and Hasan Fleyeh. Convex hulls in image processing: a scoping review. *American Journal of Intelligent Systems*, 6(2):48–58, 2016.

[29] Josef Kallrath, Joonghyun Ryu, Chanyoung Song, Mokwon Lee, and Deok-Soo Kim. Near optimal minimal convex hulls of disks. *Journal of Global Optimization*, 80(3):551–594, 2021.

[30] Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Inf. Process. Lett.*, 25:11–12, 1987.

[31] Debashis Mukherjee. Reduction of two-dimensional data for speeding up convex hull computation. *arXiv preprint arXiv:2201.11412*, 2022.

[32] Linh Kieu Nguyen, Chanyoung Song, Joonghyun Ryu, Phan Thanh An, Nam-Dũng Hoang, and Deok-Soo Kim. Quickhulldisk: A faster convex hull algorithm for disks. *Applied Mathematics and Computation*, 363:124626, 2019.

[33] Artem Potebnia and Sergiy Pogorilyy. Innovative gpu accelerated algorithm for fast minimum convex hulls computation. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 555–561. IEEE, 2015.

[34] Jiayu Qin, Gang Mei, Salvatore Cuomo, Guo Sixu, and Yixuan Li. Cudachpre2d: A straightforward preprocessing approach for accelerating 2d convex hull computations on the gpu. *Concurrency and Computation Practice and Experience*, 32, 04 2019.

[35] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. An image processing library in modern c++: Getting simplicity and efficiency with generic programming. In Bertrand Kerautret, Miguel Colom, Daniel Lopresti, Pascal Monasse, and Hugues Talbot, editors, *Reproducible Research in Pattern Recognition*, pages 121–137, Cham, 2019. Springer International Publishing.

[36] Sam Safavi and Usman A. Khan. Localization in mobile networks via virtual convex hulls. *IEEE Transactions on Signal and Information Processing over Networks*, 4(1):188–201, 2018.

[37] Zeyu Shen, Mingyang Zhao, Xiaohong Jia, Yuan Liang, Lubin Fan, and Dong-Ming Yan. Combining convex hull and directed graph for fast and accurate ellipse detection. *Graphical Models*, 116, 2021.

[38] Nikolay M Sirakov. A new active convex hull model for image regions. *Journal of Mathematical Imaging and Vision*, 26(3):309–325, 2006.