

Analysis of an optimal output-sensitive algorithm for bidimensional convex hulls

Alberto Parravicini

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1. Theoretical Background

The first chapter contains the theoretical background necessary to understand the later sections of the report.

1.1 Convex Hulls

Given an Euclidean Space of d -Dimensions E^d and a set of points P in defined in this space,

Definition 1.1.1. The **Convex Hull** $CH(P)$ of P is the *minimal convex set* containing all the points in P .

A *convex set* is a set in which, $\forall x, y \in S$, the segment $xy \subseteq S$. [5]

Equivalently, one can define the *Convex Hull* as the intersection of *all convex sets* that contains S , as the intersection of *all halfspaces* (the set of points on the side of a plane) that contain S , or as of the union of *all convex combinations* of the points in S , i.e. the points $CH(S)$ are such that: [6]

$$\sum_{i=1}^{|S|} w_i \cdot x_i, \forall x_i \in S, \forall w_i : w_i \geq 0 \text{ and } \sum_{i=1}^{|S|} w_i = 1 \quad (1)$$

In this report, it is assumed that the Euclidean Space is **bidimensional**, unless otherwise specified.

The computation of the *Convex Hull* is a classical problem of *computational geometry*, and finds applications in collision detection algorithms [5] and visual pattern matching [3], among others.

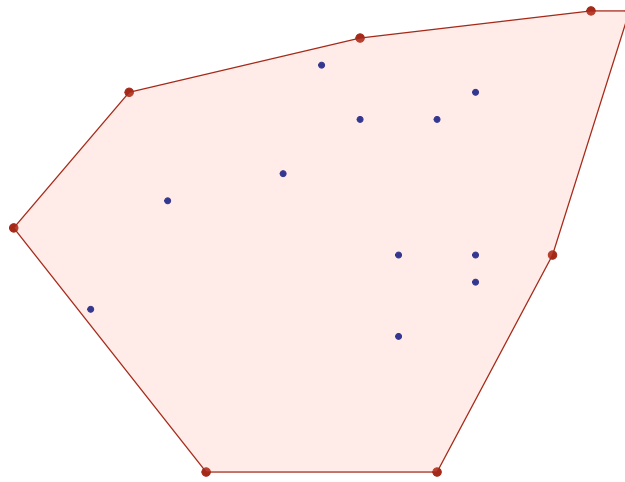


Figure 1: *Example of convex hull in 2 dimensions.*

1.2 Output-sensitive algorithms

Usually, the running time of algorithms is based on the size of their inputs. The complexity of *output-sensitive* algorithms, however is a function of both the input size the output size. [1]

It can be difficult to compare output-sensitive algorithms with algorithms whose complexity is purely input-dependent, unless it is possible to determine an upper bound on the output size. This is the case of convex hull algorithms, as presented in the following sections.

2. Algorithms for computing convex hulls

In the literature there exists a wide number of algorithms for computing convex hulls in 2 or more dimensions. In this chapter, *Jarvis March* and *Graham Scan* are presented, two commonly employed algorithms for computing 2-dimensional convex hulls. Then, these two algorithms are used to as building blocks of an *optimal algorithm*, originally presented by T. M. Chan. [1] This last algorithm is described in details, and an implementation is provided.

Note: usually, the algorithms assume points to be in *general position*, i.e. no three points form a straight line (the points are not *collinear*).

This apparently restricting assumption is justified by the existence of the so-called **Perturbation Methods**: in short, the idea is to move every point by an infinitesimal amount, as to remove the collinearity. [2]

In practice, it is generally more efficient to modify the algorithms directly, to handle these special cases. Examples of how to do so are given in the following sections.

2.1 Jarvis March

2.1.1 Introduction

Jarvis March, also referred as *Gift Wrapping*, is a simple algorithm that exploit the following property of convex hulls:

given an edge of the convex hull, it is clear that the next edge to be added to the hull is the one that maximizes the angle between the last edge and the new one.

Additionally, assuming that we build the hull in counter-clockwise order, one can easily see that no points can lie on the right of the edges of the hull.

This last observation makes it possible not to compute explicitly the angle between pairs of edges, which is in general a slow procedure subject to numerical approximation. Indeed, to find whether a point r lies on the right of a given

segment going through two points p, q , it is enough to evaluate:

$$Det \left(\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix} \right)$$

If the determinant is < 0 , the point r is on the right of pq .

As implementative detail, the first point to be added to the hull is the leftmost point in our set, as it is always part of the final hull.

2.1.2 Pseudocode

In this section is given the pseudocode of *Jarvis March*: [6]

Algorithm 1: Jarvis March

Input: a list S of bidimensional points.

Output: the convex hull of the set, sorted counterclockwise.

$hull = []$

$x_0 =$ the leftmost point.

$hull.push(x_0)$

Loop $hull.last() \neq hull.first()$

$candidate = S.first()$

foreach p in S **do**

if $p \neq hull.last()$ and p is on the right of the segment " $hull.last()$,
 $candidate$ " **then**

$candidate = p$

if $candidate \neq hull.first$ **then** $hull.push(candidate)$ **else break**

return $hull$

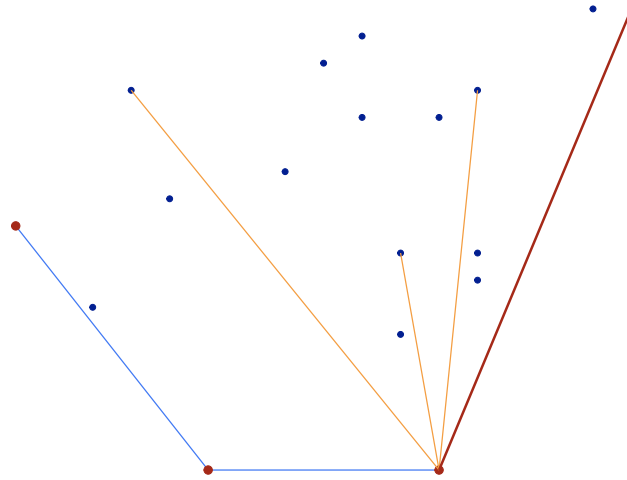


Figure 2: A step of the Jarvis March. The thick line is the next segment that will be added to the hull.

2.1.3 Complexity Analysis

From the previous pseudo-code it can be seen that the algorithm is composed of two nested loops.

The outer loop will be executed h times, where h is the size of the final hull. This happens as the outer cycle breaks when the next point to be added is already in the hull.

The inner loop scans all the points in s .

As a result, the overall complexity of the algorithm is $O(hn)$.

2.2 Graham Scan

2.2.1 Introduction

Another well-known convex hull algorithm is the so-called **Graham Scan**. The core idea of the algorithm is to preprocess the points by sorting them in counter-clockwise order around the leftmost point of the set S .

From here, it is possible to compute the hull in an incremental fashion, by making use of a stack-like structure that allows to process each of the point only once.

What the algorithm does is to push sequentially the points on the stack (which represents the hull). If the top three points on the stack cause a right turn, the second-to-last point is removed.

The idea, once again, is that the edges of a counter-clockwise sorted hull will make only left turns.

2.2.2 Pseudocode

In this section is given the pseudocode of *Graham Scan*: [6]

Algorithm 2: Graham Scan

Input: a list S of bidimensional points.

Output: the convex hull of the set, sorted counterclockwise.

hull = []

x_0 = the leftmost point.

Put x_0 as the first element of S .

Sort the remaining points in counter-clockwise order, with respect to x_0 .

Add the first 3 points in S to the hull.

forall the remaining points in S **do**

while hull.second_to_last(), hull.last(), p form a right turn **do**

 hull.pop()

 hull.push(p)

return hull

2.2.3 Implementation

```
# Compute the convex hull of the given list of points by using Graham scan
# Inspired by "http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/"
def convex_hull_graham_scan(input_points):
    # Copy the input points, so that it is possible to modify them
    points = list(input_points)
    convex_hull = []

    # Find the point with the smallest x.
    smallest_x_point_index = 0
    for index, p in enumerate(points):
        if (p.x < points[smallest_x_point_index].x) or \
            ((p.x == points[smallest_x_point_index].x) and
             (p.y < points[smallest_x_point_index].y)):
            smallest_x_point_index = index

    # Put the point with smallest x at the beginning of the list.
    points[0], points[smallest_x_point_index] = points[
        smallest_x_point_index], points[0]

    # Order the list with respect to the angle that each point forms
    # with the anchor. Given two points a, b, in the output a is before b
    # if the polar angle of a w.r.t the anchor is bigger than the one of b,
    # in counter-clockwise direction.
    anchor = Point(points[0].x, points[0].y)
    points = [anchor] + radial_sort(points[1:], anchor, cw = False)

    # If more points have the same angle w.r.t. the anchor, keep only the farthest one.
    # Used to deal with collinear points-
    i = 1
    while i < len(points) and
        (orientation_test(anchor, points[i], points[(i + 1) % len(points)]) == 0):
        points.pop((i + 1) % len(points))

    # Add the first 3 points to the convex hull.
    # The first 2 will be for sure part of the hull.
    convex_hull += points[0:3]

    for p in points[3:]:
        # While the i-th point forms a non-left turn with the last 2 elements
        # of the convex hull...
        while orientation_test(convex_hull[-2], convex_hull[-1], p) <= 0:
            # Delete from the convex hull the point that causes a right turn.
            convex_hull.pop()
        # Once no new right turns are found, add the point that gives a left turn.
        convex_hull.append(p)

    return convex_hull
```

2.2.4 Complexity Analysis

The complexity analysis of Graham Scan can be split in two parts: first, the counter-clockwise sorting of the points, which can be done in $O(n \cdot \log n)$ time; second, the scan of all the points in S , and, if needed, the scan of the hull stack.

The two nested loops might lead to imagine a quadratic complexity, but in practice it isn't possible to remove from the stack elements that haven't been seen yet by the external loop. Indeed, each point is added to the hull once, and removed at most once.

Consequently, the cost of the second part of the algorithm is $O(n)$, which results in an overall cost of $O(n \cdot \log n)$.

Now, it would be interesting to compare the complexity of *Jarvis March* to the one of *Graham Scan*. However, being *Jarvis March* is an *output-sensitive* algorithm, this comparison is not straightforward: if the size of the hull h is smaller than $\log n$, then *Jarvis March* would perform better; without a-priori knowledge of the points distribution, however, it isn't possible to state something like that.

2.3 T. Chan's Optimal Algorithm for Bidimensional Hulls

2.3.1 Introduction

Starting from the considerations of the last section, it would be interesting to have an algorithm which can, regardless of the input, perform better than both *Jarvis March* and *Graham Scan*.

Kirkpatrick and *Seidel* [4] built a $O(n \cdot \log h)$ algorithm to compute the convex hull of a set of bidimensional points. Their algorithm is however quite complex, and hard to implement in a practical context.

To overcome the issue, *T. Chan* [1] built an $O(n \cdot \log h)$ algorithm that uses *Jarvis March* and *Graham Scan* as building blocks to compute the convex hull in a simple, yet optimal, way.

The algorithm initially splits the list S , of size n in groups of size at most m . As such, there will be $\lceil n/m \rceil$ groups.

Then, the convex hull of each group is computed, by using *Graham Scan* (or another $O(n \cdot \log n)$ algorithm). The idea of this preprocessing step is that, in a given group, only the points in the partial hull will have a chance to be part of the final hull. So, by computing the $\lceil n/m \rceil$ partial hulls, it is possible to discard a significant portion of the points in S . Without loss of generality, it is possible to

assume that the partial hulls will be returned with a counter-clockwise sorting.

After finding the partial hulls, *Jarvis March* comes into play: once again, given the last edge $p_{k-1}p_k$ belonging to the final hull, the next edge p_kp to be added is the one that maximizes the angle between the two edges; equivalently, the next point p to be added is the one that is on the right of every other point, with respect to the last hull edge $p_{k-1}p_k$.

However, given the last edge of the hull and the partial hulls, it isn't necessary to evaluate every point of the hulls to find p : in fact, in a partial hull the point p that maximizes the angle $\angle p_{k-1}p_kp$ is the one that forms the right tangent p_kp to the polygon.

As the final hull will have size h , h steps of *Jarvis March* will be required.

Now, it is reasonable to ask what should be the appropriate value of m .

Let's imagine that the hull size H is known: it turns out that by setting $H = m$, the algorithm will have optimal complexity (the details of the proof are found in section 2.3.4, Complexity Analysis).

However, the value of h isn't known: to solve the problem, the algorithm is called multiple times, and at each iteration i , the algorithm sets $m = H = 2^{2^i}$.

2.3.2 Pseudocode

In this section is given the pseudocode of *Chan's algorithm*: [1]

Algorithm 3: ChanHullStep, a step of Chan's algorithm

Input: a list S of bidimensional points, the parameters m, H
Output: the convex hull of the set, sorted counterclockwise, or an empty list, if H is $< h$
Partition S into subsets $S_1, \dots, S_{\lceil n/m \rceil}$.
for $i = 1, \dots, \lceil n/m \rceil$ **do**
 Compute the convex hull of S_i by using Graham Scan, store the output in a counter-clockwise sorted list.
 $p_0 = (0, -\infty)$
 $p_1 =$ the leftmost point of S .
for $k = 1, \dots, H$ **do**
 for $i = 1, \dots, \lceil n/m \rceil$ **do**
 Compute the points $q_i \in S$ that maximizes $\angle p_{k-1}p_kq_i$, with $q_i \neq p_k$, by performing binary search on the vertices of the partial hull S_i .
 $p_{k+1} =$ the point $q \in \{q_1, \dots, q_{\lceil n/m \rceil}\}$.
 if $p_{k+1} = p_t$ **then** return $\{p_1, \dots, p_k\}$
return *incomplete*

Algorithm 4: Chan's algorithm

Input: a list S of bidimensional points
Output: the convex hull of the set
for $i = 1, 2, \dots$ **do**
 $L = \text{ChanHullStep}(S, m, H)$, where $m = H = \min\{|S|, 2^{2^i}\}$
 if $L \neq \text{incomplete}$ **then** return L

2.3.3 Implementation

The previous algorithm has been implemented in **Python**. What follows is an extract of the code, with the necessary comments. Note that the implementation of some sub-functions is not shown here. The full code can be found [here](#).

```

def hull_2d_step(points, m, H):
    # Partition the points in groups of size at most m.
    points_in_groups = list(chunks(points, m))
    hulls = []
    final_hull = []

    # Compute the convex hull of each group, and store its vertices in ccw order.
    for group_i in points_in_groups:
        hulls.append(convex_hull_graham_scan(group_i))
    # Leftmost point of the list
    final_hull.append(Point(-MIN_VALUE, -MIN_VALUE))
    # Find the next point, ccw, belonging to the hull of p_1
    p_k = Point(MIN_VALUE, MIN_VALUE)

    # Hull index of the last point added to the final hull,
    # and during the loop of the hull that is a candidate for containing
    # the next point of the final hull.
    current_hull_number = -1
    # Position of the last point added to the final hull,
    # inside its partial hull
    pos_in_last_hull = -1

    # Hull number of the last point inserted, updated after inserting a new point
    # in the final hull.
    old_hull = -1

    # Find the leftmost point.
    for hull_index, hull_i in enumerate(hulls):
        for i in range(0, len(hull_i)):
            if hull_i[i].x < final_hull[0].x:
                final_hull[0] = hull_i[i]
                current_hull_number = hull_index
                pos_in_last_hull = i

    for k in range(0, H):
        # Compute the next point in the hull of hull[-1],
        # also store the index of that hull
        p_k = hulls[current_hull_number][(pos_in_last_hull + 1) %
                                         len(hulls[current_hull_number])]
        old_hull = current_hull_number

    for hull_index, hull_i in enumerate(hulls):
        # Find the bottom tangent to from p_{k-1}, p_k to a point q_i in hull_i
        if hull_index != old_hull:
            temp_p = find_tangent_bin_search(hull_i, final_hull[-1])
            if temp_p:
                temp_tan = temp_p["tan_point"]
                temp_tan_index = temp_p["tan_index"]

                # Test if the tangent point lies on the left of
                # the segment hull[-1], p_k:
                # If so, the angle given by the tangent point is bigger,
                # and we have a new candidate.
                o = orientation_test(final_hull[-1], temp_tan, p_k)

                # If angle (hull[-2], hull[-1], temp_p) >

```

```

        # angle(hull[-2], hull[-1], p_k)
        if o > 0:
            p_k = temp_tan
            current_hull_number = hull_index
            pos_in_last_hull = temp_tan_index
    if old_hull == current_hull_number:
        pos_in_last_hull = (
            pos_in_last_hull + 1) % len(hulls[current_hull_number])
    if p_k == final_hull[0]:
        return final_hull
    final_hull.append(p_k)

return False

def hull_2d(points):
    t = 1
    while True:
        H = min(2**2**t, len(points))
        hull = hull_2d_step(points, H, H)
        if hull:
            return hull
        t += 1

```

2.3.4 Complexity Analysis

The analysis of the running time of *Chan's algorithm* can be decomposed in multiple steps. First, compute the convex hull of each subset of S , by using Graham Scan. As there are $\lceil n/m \rceil$ subsets, each of size m , the cost of this step is $O((n/m) \cdot (m \cdot \log m)) = O(n \cdot \log m)$.

Then, for all the H points in the final hull, we have to compute the tangent to each subset S_i . There will be H points in the final hull, and each step of *Jarvis March* will inspect $\lceil n/m \rceil$ partial hulls.

Finding the tangent is done with binary search in $O(\log m)$, where m is the size of a partial hull.

Consequently, this phase has an overall cost of $O(H(n/m)\log m)$. The total cost of a step of *Chan's algorithm* will thus be $O(n \log m + (H(n/m)\log m) = O(\log m(n(1 + H/m)))$, and it will return the final hull if $H \geq h$, the real hull size.

By choosing $m = H$, the complexity becomes

$$O(\log m(n(1 + H/m))) = O(n \log H)$$

As we don't know in advance the real value of h , it is required to iterate the algorithm multiple times. At each step i , the algorithm sets $m = H = 2^{2^i}$, and as the algorithm ends as soon as $H \geq h$, the number of iterations will be $\lceil \log \log h \rceil$.

The cost of an iteration will be $O(n \log H) = O(n2^i)$. Finally, it is possible to compute the overall cost of *Chan's algorithm* as

$$O\left(\sum_{i=1}^{\lceil \log \log h \rceil} n2^i\right) = O(n2^{\lceil \log \log h \rceil + 1}) = O(n \log h)$$

3. Empirical Analysis of Chan's Algorithm

3.1 Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.2 First Test: increasing number of points, fixed hull size

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.3 Second Test: increasing size of hull, fixed number of points

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus

et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.4 Third Test: increasing number of points with uniform distribution

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.5 Fourth Test: comparison with Graham Scan in case of increasing hull size

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

- [1] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, 1996.
- [2] Ioannis Z. Emiris and John F. Canny. An efficient approach to removing geometric degeneracies. Technical report, 1991.
- [3] A. Goshtasby and G. C. Stockman. Point pattern matching using convex hull edges. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(5):631–637, 1985.
- [4] David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm ? Technical report, 1983.
- [5] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.
- [6] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985.