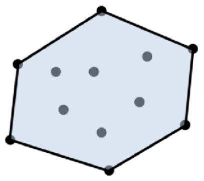
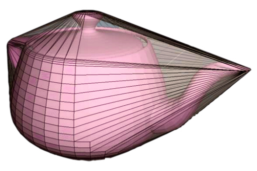



The convex hull

Domenico Ferraro
d.ferraro7@studenti.unipi.it

February 8, 2022

Abstract

Given a set of points, a convex hull is the subset of points that forms the smallest convex polytope which encloses all the points in the set. Computing the convex hull is a problem in computational geometry and a number of algorithms have been developed to compute it. This report describes the main algorithms to compute the convex hull in 2D, 3D and N-dimensions. For each algorithm we will look at the pseudocode, a running example to visualize how the algorithm works and finally a C++ implementation. We will also solve some competitive programming problems that involves the convex hull.

Contents

1 Introduction

In geometry, the convex hull of a set of N-dimensional points is the smallest subset of points that forms the smallest convex polytope which encloses all the points in the set. The convex hull in 2-dimensions is a convex polygon, which is the 2D example of a polytope. We call a shape convex, if for any two points that are inside the shape, the segment between these two points is also inside the shape.

The convex hull is one of the first problems that was studied in computational geometry and the algorithmic problem is fundamental for it. Convex hulls have wide applications in mathematics, statistics, combinatorial optimization, economics, geometric modeling, and ethology.

1.1 General cases

It is possible to generalize the problem of finding the convex hull of a set of points in N-dimensions into three general cases: - the convex hull of a single point or two points - the convex hull of three points - the convex hull of $n > 3$ points

Computing the convex hull of a single point or two points is trivial because the convex hull is just the given set of points. Computing the convex hull of three points differs if the points are stored in clockwise order or counter-clockwise order.

Given three points, a, b, and c, the order of the points can be checked by computing the cross-product of the two vectors $(b_x, b_y) - (a_x, a_y)$ and $(c_x, c_y) - (a_x, a_y)$, which is defined as a 2×2 determinant:

$$\textit{counter-clockwise} \iff \begin{bmatrix} b_x - a_x & b_y - a_y \\ c_x - a_x & c_y - a_y \end{bmatrix} > 0$$

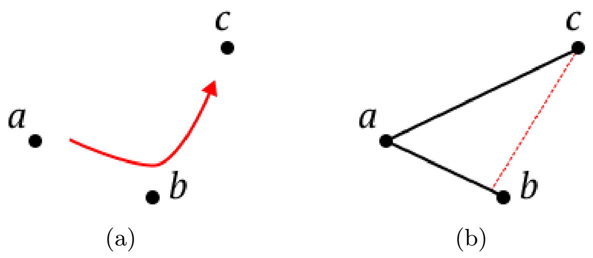


Figure 1: Example of three points in counter-clockwise order (a). As can be seen, the cross product is positive (b).

Computing the convex hull of $n > 3$ points is not as trivial as the previous cases and it is needed an algorithm approach. During the years, computer scientists have developed a multitude of algorithms, each with its advantages and disadvantages.

The algorithmic problem of finding the convex hull of a set of n points can be solved in time $O(n \log n)$ for two or three dimensional point sets. In higher dimensions, it can be solved at the worst case in time relative to the output complexity given by the upper bound theorem.

1.2 Upper Bound theorem

According to the upper bound theorem, the number of edges of the convex hull of n points in d-dimensional space is $O(n^{\lfloor d/2 \rfloor})$ [de Berg et al., 2008]. In particular

- in 1 dimension, the set of points lie on a line and the convex hull is just the minimum and the maximum points, so the number of edges is $O(1)$
- in a 2-dimensional space, the convex hull is the polygon that contains all the points. At the worst case, for example, the points can form a square and the convex hull of a square contains all the points, so the number of edges is $O(n)$
- in a 3-dimensional space, the convex hull is a convex polytope so, similarly to the 2-dimensional case, at the worst case the number of edges is $O(n)$

2 Algorithms

2.1 Jarvis's Algorithm (Gift Wrapping)

The simplest algorithm for computing convex hulls is the Jarvis's Algorithm. This algorithm is usually called Jarvis's march, after R. A. Jarvis, who published it in 1973 [Jarvis, 1973]. It is also referred to as the gift-wrapping algorithm because it simulates the process of wrapping a piece of paper around the given points. The approach can be extended to higher dimensions.

The algorithm begins by computing the leftmost point l (i.e, the point whose x-coordinate is smallest). The reason behind this first step is that we are sure that the leftmost point must be a convex hull vertex.

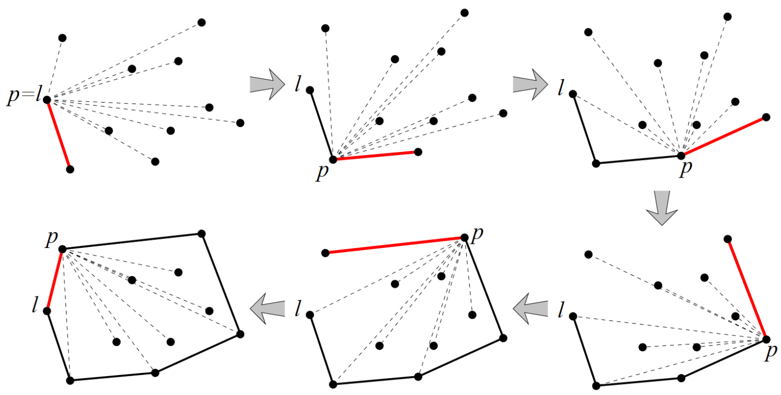


Figure 2: Execution of the Jarvis's Algorithm [Erickson, 2004]

Given a convex hull vertex p_i and its successor p_{i+1}, all the other points lie on the left of the line $p_i\ p_{i+1}$. Then, starting with l and continuing until we reach l again, the algorithm finds each successive convex hull vertex. Each successor p_{i+1} can be computed in linear time by performing a series of $O(n)$ comparisons.

```
JarvisMarch($S$)
1   $hullPoint$ = index of leftmost point // The leftmost point is guaranteed to be part of the CH($S$)
2   $i = 0$
3   **repeat**
4       $CH[i] = S[hullPoint]$   // CH will be the set of points which form the convex hull
5       $endPoint = (hullPoint + 1)\ \%\ |S|$       // Make sure $hullPoint \neq endPoint$
6       **for** $j = 0$ **to** $|S|$
7           **if** $S[j]$ is on right of line from $CH[i]$ to $S[endPoint]$
8               $endPoint = j$                  // Found greater right turn, update endpoint
9       $i = i + 1$
10      $hullPoint = endPoint$
11  **until** $hullPoint == 0$
12  **return** CH
```

Algorithm analysis

The pseudocode shows us that the worst-case running time is $O(n^2)$. However, Jarvis's march is extremely fast if the convex hull has not so many vertices. We know that for each convex hull vertex, the algorithm spends $O(n)$ time to find the successor. Then, a better way to write the running time is $O(nh)$, where h is the number of convex hull vertices. In the worst case we get a time complexity of $O(n^2)$ because $h = n$, but in the best case $h = 3$, and the algorithm only needs $O(n)$ time. Computational geometers call this an *output-sensitive algorithm.* In computer science, an output-sensitive algorithm is an algorithm whose running time depends on the size of the output, instead of, or in addition to, the size of the input. [Wikipedia, 2021]

Implementation

The following is a C++ implementation of the Jarvis's march algorithm. It stores the convex hull into a C++ vector which will contain the convex hull points in counter-clockwise order.

```cpp
std::vector<Point> jarvismarch(const std::vector<Point> &points) {
    std::vector<Point> ch;
    int leftmostPointIndex = leftmost_point_index(points);
    int hullPointIndex = leftmostPointIndex;
    Point hullPoint = points[hullPointIndex];
    do {
        ch.push_back(hullPoint);
        int endPointIndex = (hullPointIndex + 1) % points.size();
        for (int i = 0; i < points.size(); ++i) {
            if (orientation(hullPoint, points[i], points[endPointIndex]) == CCW)
                endPointIndex = i;
        }
        hullPoint = points[endPointIndex];
        hullPointIndex = endPointIndex;
    } while(hullPointIndex != leftmostPointIndex);

    return ch;
}
```

Code 1: C++ implementation of the Jarvis's algorithm

The code is totally inspired by the pseudocode, and the key parts are the usage of the functions `leftmost_point_index()` and `orientation()`. The first is used to compute, respectively, the index of the leftmost point. The second one is used to test if the points are in counter-clockwise order to check if a greater right turn has been found.

2.2 Graham's scan

Just as in Jarvis's march, Graham's scan starts by finding the left-most point l and it selects the bottom-most one if there is a tie. Then the algorithm does two main steps. The first one explicitly sorts the points in counter-clockwise order around the point l. This can be done in $O(n \log n)$ time with any sorting algorithm (quicksort, mergesort, etc...). The comparison between two points, let's say a and b, is made by testing if the triple lab is in counter-clockwise order or not. After sorting them, if you connect the points in counter-clockwise order you get a simple polygon with n points like the following one:

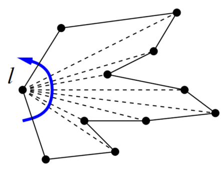


Figure 3: The polygon after the sorting phase [Erickson, 2004]

Now, the algorithm does the second and last step: it converts the polygon into the convex hull. To do that, for each point in the sorted sequence, it adds the current point to the convex hull. Then, it determines if the last 3 points in the convex-hull make a right turn. If they do, then the second-last point is not part of the convex hull, and it lies inside it, so it is removed from the convex-hull. The algorithm does it multiple times until the last 3 points do a left turn.

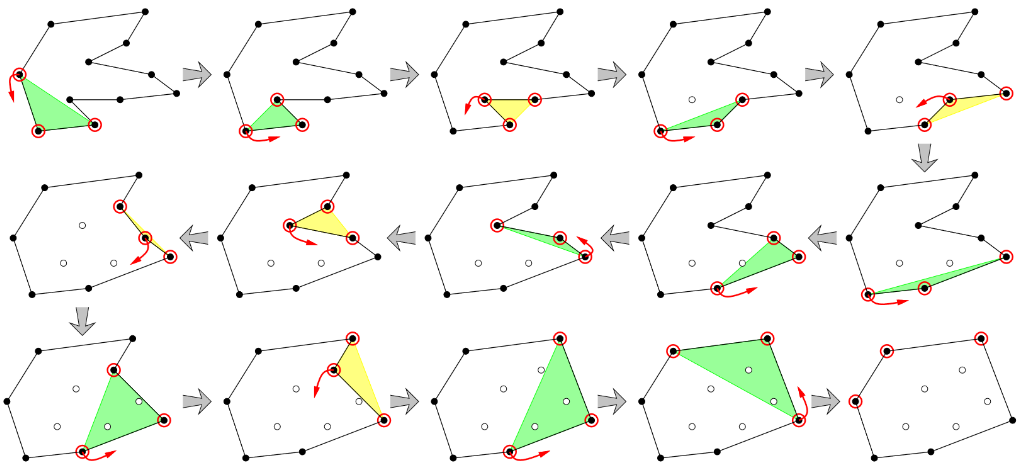


Figure 4: Converting the polygon into the convex hull [Erickson, 2004]

```
GRAHAMSCAN($S$)
1   $p_0$ = left-most point or bottom-most point if a tie
2   SORTCCW($S, p_0$) // sort the points in counterclockwise order around $p_0$
3   $CH$ = empty stack
4   PUSH($p_0, CH$)
5   PUSH($p_1, CH$)
6   PUSH($p_2, CH$)
7   for $i = 3$ to $|S|$
8       while notCCW(NEXTTOTOP($CH$), TOP($CH$), $p_i$)
9           POP($CH$)
10      PUSH($p_i, CH$)
11  return $CH$
```

The function NEXTTOTOP() returns the item one entry below the top of the stack and the function TOP() returns the topmost element. Both functions just return the point, without changing the stack. Given three points, a, b and c, the predicate CCW() tests if abc is in counter-clockwise order.

At the end of the algorithm, the stack contains the convex hull, where the points are oriented counter-clockwise and p_0 is the first point.

Algorithm analysis

Sorting the points has time complexity $O(n \log n)$, while the time complexity of the loop is $O(n)$ because each point is considered at most twice. Each point can appear only once as a second last point because in a "left turn" the algorithm advances to the next point, and in a "right turn" the point is removed. The overall time complexity is therefore $O(n \log n)$, since the time to sort dominates the time to actually compute the convex hull.

Implementation

The following is a C++ implementation of the Graham's scan algorithm.

```cpp
std::vector<Point> grahamscan(std::vector<Point> &points) {
    std::vector<Point> ch;
    int leftmostIndex = leftmost_point_index(points);
    Point leftmost = points[leftmostIndex];
    std::swap(points[leftmostIndex], points[0]);

    // Sort points in ccw order from the leftmost
    std::sort(points.begin()+1, points.end(), [&leftmost](Point &p1, Point &p2) {
        int orien = orientation(leftmost, p1, p2);
        if (orien == COL) // Points are colinear. Compare by distance
            return dist_sq(leftmost, p2) >= dist_sq(leftmost, p1);

        return orien == CCW;
    });

    ch.push_back(leftmost);
    ch.push_back(points[1]);
    ch.push_back(points[2]);
    for (int i = 3; i < points.size(); ++i) {
        while(ch.size() > 1 && orientation(ch[ch.size() - 2], ch.back(), points[i]) != CCW)
            ch.pop_back();
        ch.push_back(points[i]);
    }

    return ch;
}
```

Code 2: C++ implementation of the Graham's scan algorithm

While it is inspired by the pseudocode, this implementation doesn't use a stack but instead it uses a C++ vector from the C++ standard library. This approach makes easier to retrieve the point next to the top point. If the points in the convex hull are k then the point next to the top is retrieved by just getting the $k - 2$ element.

After computing the leftmost point, Graham's scan needs to sort the points in counter-clockwise order from the leftmost point before computing the convex hull. It is important to pay attention at this step: the sorting phase checks the order between the given point and the leftmost one. Then, sorting together with the leftmost point will result into an unexpected behaviour. Because of that, the leftmost point is swapped with the point at the first position and the sorting is done from the second point to the last one. The function used to sort the points is the C++ `sort()` function from the C++ standard library.

Because *Point* is a custom class and the sorting is done in uncommon way, this implementation passes to the `sort()` function the function that should be used to compare the points. To keep the code shorter, the function is presented as a lambda function. The comparison is done between the two points and the leftmost one, but, if the three points are colinear, then the comparison is done by distance.

2.3 Quickhull

Quickhull is a method of computing the convex hull that also works in n-dimensional space. It was invented in 1996 by Barber and Dobkin and it is a quick-sort like algorithm for convex hull, from which its name derives. It is essentially an iterative algorithm that adds individual points one point at a time to an intermediate hull and uses a divide and conquer approach.

```
QuickHull($S$)
 1  $l$ = left-most point
 2  $r$ = right-most point
 3  $S1$ = points in S that are on the right side of the oriented line from $l$ to $r$
 4  $S2$ = points in S that are on the right side of the oriented line from $r$ to $l$
 5  $CH$ = {} // It will be the set of points which form the convex hull
 6
 7  FindHull($CH, S1, l, r$)
 8  FindHull($CH, S2, r, l$)
 9
10  return $CH$
```

Initially, the algorithm finds the leftmost l and rightmost r points and it divides the points in two sets: the set of points that lie on the left of the segment lr and the set of points that lie on the right.

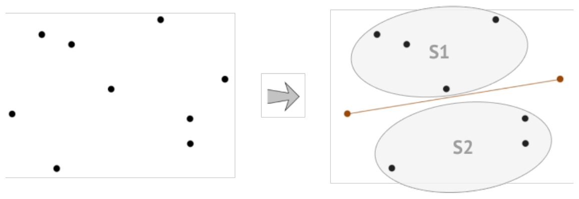


Figure 5: Example of how the algorithm initially divides the set of points

Then, it calls the FindHull procedure for both sets. The procedure finds the farthest point from the segment and adds it to the convex hull. This produces a triangle made by the following three vertices: the farthest point f and the two points l and r that made the segment.

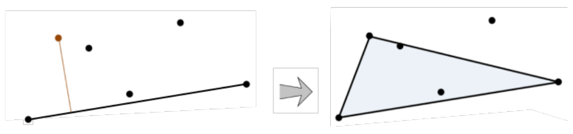


Figure 6: Example of the triangle made by the farthest point and the given segment

At this point, the triangle partition the remaining points into three sets: the set of points that lie inside the triangle, the set of points that lie outside of the triangle which are in front of the segment fr, and the set of points that lie in front of the segment lf. The procedure, then, can ignore the points inside the triangle (because they are not part of the convex hull) and calls recursively on the other two sets. The following pseudocode summarizes the FindHull procedure:

```
FindHull($CH, S_k, a, b$)
 1  if $S_k$ is empty
 2      return
 3  $f$ = farthest point in $Sk$ from segment $ab$
 4  // The points $a$, $b$, and $f$ partition the remaining points of $S_k$ into 3 subsets: $S0$, $S1$, and $S2$
 5  $S1$ = points in $S_k$ that are on the right side of the oriented line from $a$ to $f$
 6  $S2$ = points in $S_k$ that are on the right side of the oriented line from $f$ to $b$
 7
 8  Add point $f$ to $CH$ at the location between $a$ and $b$
 9  FindHull($CH, S1, a, f$)
10  FindHull($CH, S2, f, b$)
```

Algorithm analysis

On average the algorithm works quite well, but it is slow in cases of high symmetry or when the points lie on the circumference of a circle. At the worst case the time complexity of the algorithm is $O(n^2)$ and at the average case it is $O(n \log n)$.

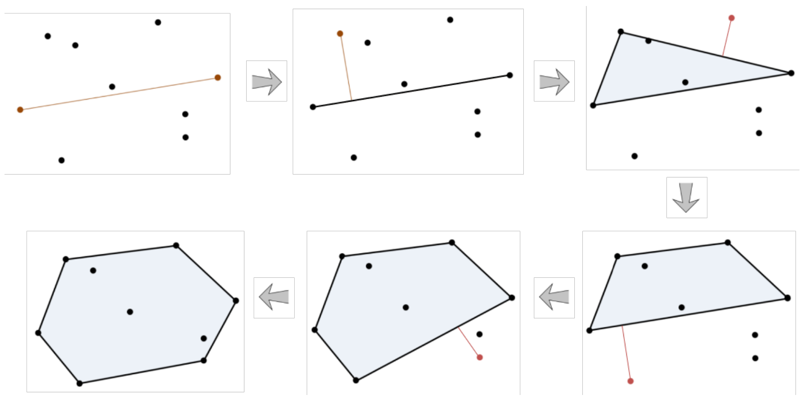


Figure 7: Entire execution of the Quickhull algorithm

Implementation

The following is a C++ implementation of the QuickHull procedure.

```cpp
std::vector<Point> quickhull(const std::vector<Point> &points) {
    Point leftmost = leftmost_point(points);
    Point rightmost = rightmost_point(points);
    std::vector<Point> ch;
    std::vector<Point> rightside;
    std::vector<Point> leftside;

    // Split the points by the line from the leftmost to the rightmost
    for (Point point: points) {
        int orien = orientation(leftmost, rightmost, point);
        if (orien == CCW) { // this point is on the right of the line
            leftside.push_back(point);
        } else if (orien == CW) { // this point is on the left of the line
            rightside.push_back(point);
        }
    }

    findhull(ch, rightside, leftmost, rightmost);
    findhull(ch, leftside, rightmost, leftmost);

    return ch;
}
```

Code 3: C++ implementation of the QuickHull procedure

The implementation doesn't differ too much from the pseudocode. There are no particular cases to handle or particular functions to call. The only functions are the simple ones to retrieve the leftmost and the rightmost points and the function to get the orientation of three points. The key part of the implementation of the QuickHull algorithm is the `findhull()` function, which is the following:

```cpp
void findhull(std::vector<Point> &ch, const std::vector<Point> &points,
                const Point &left, const Point &right) {
    int farPointIndex = -1;
    int maxDistance = 0;

    // Find the farthest point from the line left-right
    for (int i = 0; i < points.size(); ++i) {
        // distance from the point to the line left-right
        int distance = abs((points[i].y - left.y) * (right.x - left.x) -
                           (right.y - left.y) * (points[i].x - left.x));
        if (distance > maxDistance) {
            farPointIndex = i;
            maxDistance = distance;
        }
    }

    // there are no remaining points
    if (farPointIndex == -1) {
        ch.push_back(left);
        return;
    }

    // Divide the set of points into two subsets
    std::vector<Point> rightside; // points to the right of the line right-max
    std::vector<Point> leftside;  // points to the left of the line left-max
    for (int i = 0; i < points.size(); ++i) {
        if (i == farPointIndex) continue; // avoid checking the max again

        int orienWithLeft = orientation(left, points[farPointIndex], points[i]);
        if (orienWithLeft == CW) {
            // the point lies to the right of the line left-far
            rightside.push_back(points[i]);
        } else if (orientation(right, points[farPointIndex], points[i]) == CCW) {
            // the point lies to the left of the line right-far
            leftside.push_back(points[i]);
        }
    }

    findhull(ch, rightside, left, points[farPointIndex]);
    findhull(ch, leftside, points[farPointIndex], right);
}
```

Code 4: C++ implementation of the FindHull procedure

The function `findhull()` takes as input the subset of points, a leftmost point *left* and a rightmost point *right*. All the points in the input subset lie to the right of the line *left-right*.

The function initially computes the farthest point *far* from the line *left-right*, then divides the points and does the two recursive calls. To stop the recursion, it is not enough to check if the input subset is empty: if all the remaining points lie exactly on top of the line *left-right* (i.e, their distance from the line is zero) they must be discarded. So, after looking for the most distant point, if it is not found then the set is empty or all the subset of points lie above the line. When this happens, `findhull()` adds the leftmost point to the hull and the function stops without going further into the recursive calls. Because of that, during the execution, the hull is built in counter-clockwise order.

To split the set of points, the function iterates through all the points in the subset and for each point *p* it checks if the line *left-far-p* does a clockwise turn. If it does, then *p* lies to the right of the line. Otherwise it checks if the *right-far-p* does a counter-clockwise turn. It it does, then the point liew to the left of the line.

2.4 Divide and conquer

Divide and conquer algorithms solve problems by dividing them into smaller instances, solving each instance recursively and merging the corresponding results to a complete solution. The following algorithm is a divide and conquer approach for the construction of the convex hull. It was invented by Preparata and Hong in 1977 [Franco P. Preparata, 1977].

```
PREPARATA($S$)
1  $m = \lfloor n/2 \rfloor$
2  $S1 = S[0..m]$
3  $S2 = S[m+1..n]$
4  $CH1 =$ PREPARATA($S1$)
5  $CH2 =$ PREPARATA($S2$)
6  $CH =$ merge $CH1$ and $CH2$
7  return $CH$
```

The algorithm requires that the set of points is sorted by x-coordinate, from the leftmost to the rightmost point. The algorithm recursively does the following steps. Given the point in the middle p_m, it partitions the set of points into the set of points on the left of p_m (which contains p_m as well) and the set of points on the right of p_m. Then, it recursively computes the convex hull of both sets and merges the resulting subhulls together by finding the tangents that connect the two hulls from above and below.

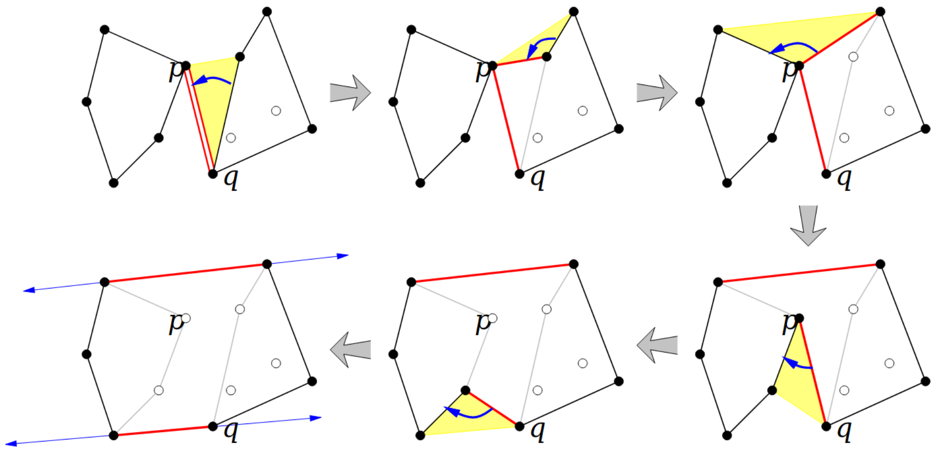


Figure 8: How the merge of two adjacent convex hulls is done [Erickson, 2004]

To merge the subhulls, the algorithm creates two segments, called *bridges*, between the rightmost point of the left subhull and the leftmost point of the right subhull. As long as there is a clockwise turn at either endpoint of either bridge, the point is removed and its two neighbors became connected. As soon as the turns at both endpoints of both bridges are counter-clockwise, the algorithm stops. Note that, after merging the subhulls, the bridges lie on the upper and lower common tangent lines of the two subhulls.

Algorithm analysis

It is clear that this algorithm resembles to QuickSort. The running time is given by the recurrence $T(n) = O(n) + T(k) + T(n-k)$, just like QuickSort, where k is the number of points to the right of the middle point p_m. Just like QuickSort, a deterministic approach to chose p_m will result into a $O(n^2)$ time complexity at worst case. If p_m is chosen randomly, then the running time is expected to be $O(n \log n)$. Note that there are input where this algorithm is not practical.

Implementation

The implementation of this algorithm follows the typical divide and conquer approach. It uses the function `ch_three_points()` which computes the convex hull of three points.

```cpp
std::vector<Point> preparata(std::vector<Point> &points, int l, int r) {
    if (r - l == 1) return { points[l] };
    if (r - l == 2) return { points[l], points[l+1] };
    if (r - l == 3) return ch_three_points(points[l], points[l+1], points[l+2]);

    int m = (l+r) / 2;
    std::vector<Point> ch1 = preparata(points, l, m);
    std::vector<Point> ch2 = preparata(points, m, r);

    return merge(ch1, ch2);
}
```

Code 5: C++ implementation of the Preparata procedure

The important phase of the algorithm is of course the merging phase, which is implemented like the following:

```cpp
std::vector<Point> merge(const std::vector<Point> &ch1, const std::vector<Point> &ch2) {
    int ch1_rightmost = rightmost_point_index(ch1);
    int ch2_leftmost = leftmost_point_index(ch2);

    // compute upper bridge
    std::pair<int, int> up_bridge = upper_bridge(ch1, ch1_rightmost, ch2, ch2_leftmost);

    // compute lower bridge
    std::pair<int, int> lo_bridge = lower_bridge(ch1, ch1_rightmost, ch2, ch2_leftmost);

    // merge the subhulls
    std::vector<Point> ch_merged;
    ch_merged.push_back(ch1[up_bridge.first]);
    int i = up_bridge.first;
    while (i != lo_bridge.first) {
        i = (i+1) % ch1.size();
        ch_merged.push_back(ch1[i]);
    }

    ch_merged.push_back(ch2[lo_bridge.second]);
    i = lo_bridge.second;
    while (i != up_bridge.second) {
        i = (i+1) % ch2.size();
        ch_merged.push_back(ch2[i]);
    }

    return ch_merged;
}
```

Code 6: Merging two subhulls into one

The function to compute the upper bridge is given below. The one for computing the lower bridge is the same but it is specular. To find the upper bridge, the algorithm starts with two points, one from each subhull, and then iteratively goes clockwise or counter-clockwise from the first or the second points until the line that connects the two points is the upper tangent line of the two subhulls.

```cpp
std::pair<int, int> upper_bridge(const std::vector<Point> &ch1, int ind_ch1,
                                 const std::vector<Point> &ch2, int ind_ch2) {
    bool done = false;
    int next_ind2 = (ch2.size() + ind_ch2 - 1) % ch2.size();
    int next_ind1 = (ind_ch1 + 1) % ch1.size();
    while(!done) {
        done = true;
        while(orientation(ch2[ind_ch2], ch1[ind_ch1], ch1[next_ind1]) == CW) {
            ind_ch1 = next_ind1;
            next_ind1 = (next_ind1 + 1) % ch1.size();
        }

        while(orientation(ch1[ind_ch1], ch2[ind_ch2], ch2[next_ind2]) == CCW) {
            ind_ch2 = next_ind2;
            next_ind2 = (ch2.size() + next_ind2 - 1) % ch2.size();
            done = false;
        }
    }

    return std::make_pair(ind_ch1, ind_ch2);
}
```

Code 7: Computing the upper bridge

2.5 Monotone chain

The Andrew's Monotone chain method is a simple way to compute the convex hull of a set of points in 2-dimensions. First thing first it sorts the points by x-coordinate, and in case of a tie, by y-coordinate. After the sorting phase, it constructs the lower hull by starting from the leftmost point and trying to go counter-clockwise as much as possible until it reaches the rightmost point. Then, it constructs the upper hull going back from the rightmost point to the leftmost and trying to go counter-clockwise as much as possible. The algorithm's pseudocode is given below.

```
MonotoneChain($S$)
1   Sort($S$) // Sort the points by x-coordinate or by y-coordinate in case of a tie
2   $CH$ = {$S[0]$} // It will be the set of points which form the convex hull
3   $i = 0$
4   while $i < |S|$
5       if $|CH| >= 2$ and CW(NextToTop($CH$), Top($CH$), $S[i]$)
6           Pop($CH$) // Found clockwise turn, update the hull
7       else
8           Push($CH, S[i]$)
9           $i = i + 1$
10
11  $i = |S| - 2$
12  while $i >= 0$
13      if $|CH| >= 2$ and CW(NextToTop($CH$), Top($CH$), $S[i]$)
14          Pop($CH$) // Found clockwise turn, update the hull
15      else
16          Push($CH, S[i]$)
17          $i = i - 1$
18
19  return $CH$
```

The following figure gives a visualization of how the algorithm constructs the upper hull. During the iteration it keeps track of the intermediate convex hull by keeping the sequence of the previous scanned points that have counter-clockwise order. It starts from the leftmost point and scans the sorted points until it reaches the rightmost point. For each point, if the last point in the intermediate convex hull and the current point are in clockwise order, then the algorithm removes the last point from the intermediate convex hull and evaluates again the current point until it is in counter-clockwise order with the last point of the intermediate convex hull.

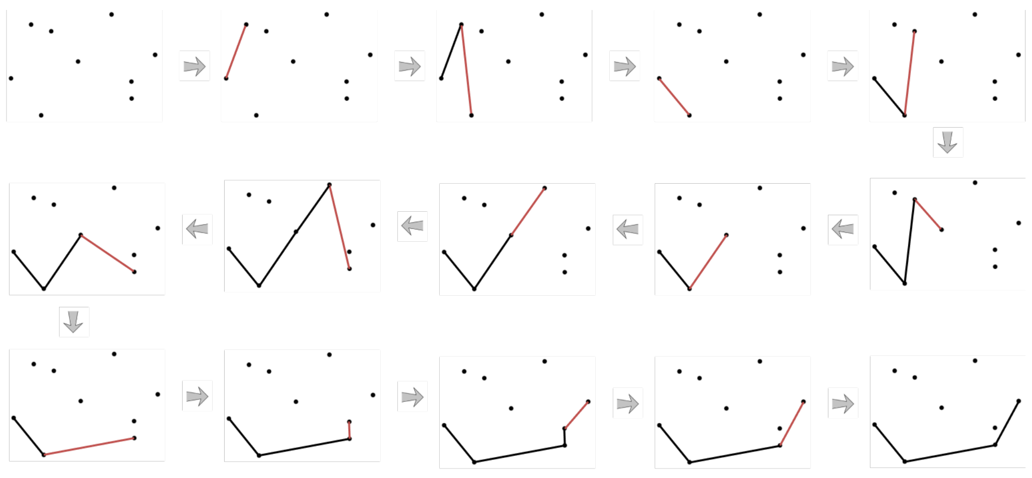


Figure 9: Example execution of how the Monotone chain algorithm constructs the upper hull

Then, the algorithm applies the same approach to construct the lower hull by scanning the points from the rightmost to the leftmost and testing if the points have a counter-clockwise order.

Algorithm analysis

The construction of the upper and lower hulls of the points have time complexity of $O(n)$ time, because the algorithm linearly scans each point. The overall time complexity is therefore $O(n \log n)$, since the time to sort dominates the time to actually compute the convex hull.

Implementation

```cpp
std::vector<Point> monotonechain(std::vector<Point> &points) {
    // Sort points by x-coordinate or by y-coordinate if there is a tie
    std::sort(points.begin(), points.end(), [](Point &p1, Point &p2) {
        return p1.x == p2.x ? p1.y < p2.y : p1.x < p2.x;
    });

    std::vector<Point> ch { points[0] };
    // build lower hull
    int i = 1;
    while (i < points.size()) {
        if (ch.size() >= 2 && orientation(ch[ch.size() - 2], ch.back(), points[i]) == CW) {
            ch.pop_back();
        } else {
            ch.push_back(points[i]);
            i++;
        }
    }

    // build upper hull
    i = points.size() - 2;
    while (i >= 0) {
        if (ch.size() >= 2 && orientation(ch[ch.size() - 2], ch.back(), points[i]) == CW) {
            ch.pop_back();
        } else {
            ch.push_back(points[i]);
            i--;
        }
    }

    if (ch[0] == ch.back())
        ch.pop_back();

    return ch;
}
```

Code 8: C++ implementation of Monotone Chain algorithm

2.6 Incremental convex hull algorithm

The Incremental convex hull algorithm computes the convex hull by iteratively adding points [Gang Mei, 2012]. The algorithm iterates through all the points and, for each point, it tests if the point is inside or outside the hull. If a point is already in the hull then it is ignored, otherwise it is added to the hull. If a point is outside of the hull then it can "see" a certain amount of other points in the hull. Doesn't matter how many of them the point can see, there will always be two points, called *transition points*, that are the last two points that the point can see (after them, all the other points cannot be seen by the current point). So the algorithm removes the other ones from the hull and the point is added to the hull between the two transition points.

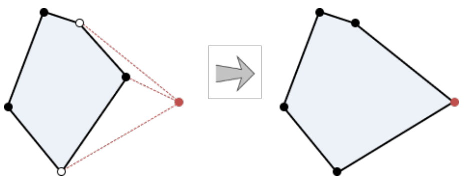


Figure 10: Here the red point can see three points. The transition points are the white points, because after them there are only points not visible from the red point. The transition points should remain into the hull, while the other non-transition points that are visible from the red point should be removed from the hull.

```
INCREMENTALCH($S$)
1  SORT($S$) // Sort the points from the leftmost to the rightmost (i.e, by x-coordinate)
2  $CH = \{S[0], S[1], S[2]\}$
3  for $i = 3$ to $|S|$
4      $t_1, t_2 =$ TRANSITIONPOINTS($CH, S[i]$)
5      REMOVEBETWEEN($CH, t_1, t_2$) // Remove the points between the transition points
6      ADDBETWEEN($CH, S[i], t_1, t_2$) // Add the current point between the transition points
7  return $CH$
```

To compute if a point is outside or inside the hull, the algorithm should compare the point with each point in the hull. This will take linear time, so the overall time complexity of the algorithm will become $O(n^2)$. However, the Incremental convex hull algorithm uses a better approach, and it can compute if a point lies inside or outside the hull faster. To do that, the algorithm first sorts the points by x-coordinate. Since the points are sorted, each new point considered must see the previous added point. Any vertex traversed to find the transition point is removed.

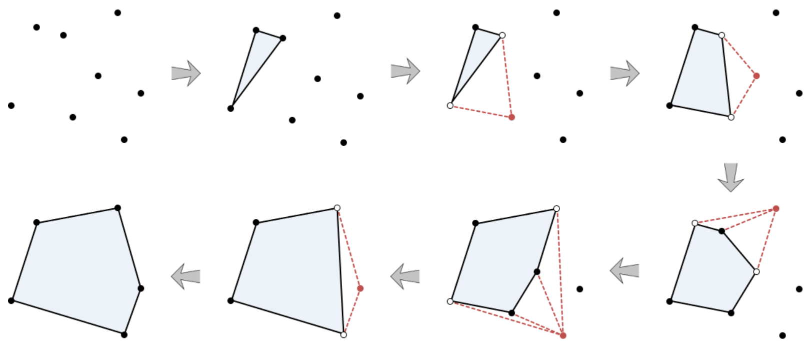


Figure 11: Execution of the incremental convex hull algorithm

Algorithm analysis

By sorting the points it is possible to compute the transition points in constant time. So, computing the convex hull takes $O(n)$ time. The overall time complexity is therefore $O(n \log n)$, since the time to sort dominates the time to actually compute the convex hull.

Implementation

A good implementation of the Incremental convex hull algorithm has to efficiently manage the convex hull points because it is needed to add/remove points between two transition positions.

The STL C++ vector is not suitable because removing points from two positions will also need to shift to the left some points from the farthest position. An approach with a double linkedlist can be useful but it will be sensible to cache misses and should be avoided as well. The following is a C++ implementation of the algorihtm and uses a different and more efficient approach.

```cpp
std::vector<Point> incremental(std::vector<Point> points) {
    // Sort points by x-coordinate or by y-coordinate if there is a tie
    std::sort(points.begin(), points.end(), [](Point &p1, Point &p2) {
        return p1.x == p2.x ? p1.y < p2.y : p1.x < p2.x;
    });
    std::vector<std::pair<int, int>> adj(3);
    // init the adjacency vector in counter-clockwise order
    if (orientation(points[0], points[1], points[2]) == CCW)
        adj = { {1, 2}, {1, 0}, {0, 1} }; // order is 0 -> 1 -> 2
    else
        adj = { {2, 1}, {0, 2}, {1, 0} }; // order is 0 -> 2 -> 1

    int t1, t2, curr, next;
    for (int i = 3; i < points.size(); ++i) {
        // find the transition points t1 and t2
        curr = i-1;
        next = adj[i-1].first;
        while(orientation(points[i], points[curr], points[next]) == CW) {
            curr = next;
            next = adj[next].first;
        }
        t1 = curr;
        curr = i-1;
        next = adj[i-1].second;
        while(orientation(points[i], points[curr], points[next]) == CCW) {
            curr = next;
            next = adj[next].second;
        }
        t2 = curr;
        // remove points between t1 and t2
        adj[t2].first = i; adj[t1].second = i;
        // add current point between t1 and t2
        adj[i].first = t1; adj[i].second = t2;
    }
    // compute the convex hull vector from the adjacency vector
    std::vector<Point> ch;
    int i = 0;
    do {
        ch.push_back(points[i]);
        i = adj[i].first;
    } while(i != 0);

    return ch;
}
```

Code 9: C++ implementation of the Incremental convex hull algorithm

This implementation uses a matrix of two columns and n rows, where n is the number of the given points. The $i-th$ row stores the indexes of the successor and predecessor of the $i-th$. In other words, it is like an adjacency matrix of a graph in which the nodes are the convex hull points and each node has two edges. With this approach, removing and adding points between the two transition points is very efficient, it has linear time complexity, it does not need to shift any points and it is not sensible to cache misses.

2.7 Kirkpatrick–Seidel algorithm

The Kirkpatrick–Seidel algorithm, named after its inventors, David G. Kirkpatrick and Raimund Seidel, is a divide and conquer algorithm [David G. Kirkpatrick, 1986]. Like the monotone chain method, the Kirkpatrick–Seidel algorithm computes the upper-hull and the lower-hull and then merges them. The algorithm to compute the upper-hull is given below. The lower-hull can be computed in a similar manner and is omitted from further discussion.

Compared to the divide and conquer algorithm by Preparata, the Kirkpatrick–Seidel algorithm reverses the order of the dividing and merging steps. Instead of dividing first and merging after, the algorithm first merges the upper subhulls by computing the upper bridge that connects the two sub hulls and then divides the points. Because of that, this algorithm is also called by its authors as *Marriage Before Conquest.*

```
MBCUpperHull($S$)
1  $med$ = the index of the median of x-coordinates of points in $S$
2  $L = S[0..med]$
3  $R = S[med + 1..|S|]$
4  $p, q$ = upper bridge of L and R, $p \in L$ and $q \in R$
5  $L' = \{ l \in L \mid x(l) \leq x(p) \}$
6  $R' = \{ r \in R \mid x(r) \geq x(q) \}$
7  $LUH$ = MBCUpperHull(L')
8  $RUH$ = MBCUpperHull(R')
9  return the concatenation $LUH$, $pq$, $RUH$
```

As per the definition of a convex hull, all the points below the upper bridge can not be a part of the upper hull. Dividing the points after computing the bridge allows to efficiently evaluate only the points that can possibly be part of the hull. In the case of divide and conquer algorithm by Preparata, finding bridges constitute the "merge" step of the algorithm. In this algorithm, finding bridges are the primary algorithmic step.

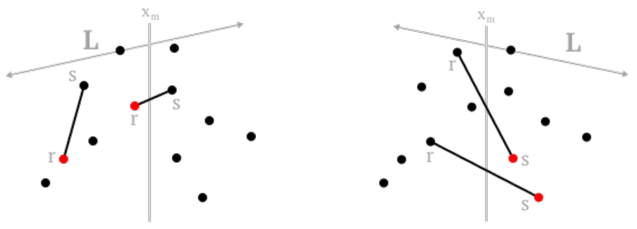
The algorithm works by finding the median x-coordinate and then finding the bridge. The points in the bridge are added to the hull. The remaining points are split into two sets: the set of points to the left of the bridge, and the set of points to the right. The algorithm is called recursively on each subset.

Finding the Bridge

Computing the bridge can be done in $O(n)$ time thanks to a *Prune and Search* technique which allows to eliminate a good many of the points that do not define the upper bridge [Chen, 2019]. More formally, the bridge is a line t that passes through one point from the left subset and one point from the right subset, such that none of the given points lie above t.

The set of points is partitioned into two subsets by the vertical line defined by the median x_m. We can have an upper supporting line L which is tangent to the set of points at a point p_t. If an upper supporting line contains points to the left of x_m and also points to the right, then the supporting line contains the bridge. Otherwise, we have the following observations. Given any two points r and s from the set of points, such that r is to the left of s

- if $x(p_t) < x_m$ and $slope(\overline{\mathrm{rs}}) \geq slope(L)$, then the bridge doesn't pass through r
- if $x(p_t) \geq x_m$ and $slope(\overline{\mathrm{rs}}) < slope(L)$, then the bridge doesn't pass through s



(a) (b)

Figure 12: Example of the previous observations. The red points can be removed.

To prove our observations, let's just think about the first case for now, when p_t has x-coordinate $< x_m$. In that case L doesn't pass through any point to the right of the median, and it passes through p_t, which is to the left. Because of that, the slope of the bridge must be lower than the slope of L. So, if $slope(\overline{rs}) < slope(L)$ (Figure 12b), then the bridge cannot pass through r because if it does then it will pass below s and it cannot be possible by the definition of bridge.

Similarly, when L contains only points to the right of the median, which means that p_t has x-coordinate $\geq x_m$, then the slope of the bridge must be greater than the slope of L. So, if $slope(\overline{rs}) \geq slope(L)$ (Figure 12a) then the point s is not part of the bridge, otherwise it will imply that the bridge passes below r but it is a contradiction so the point s cannot be part of the bridge.

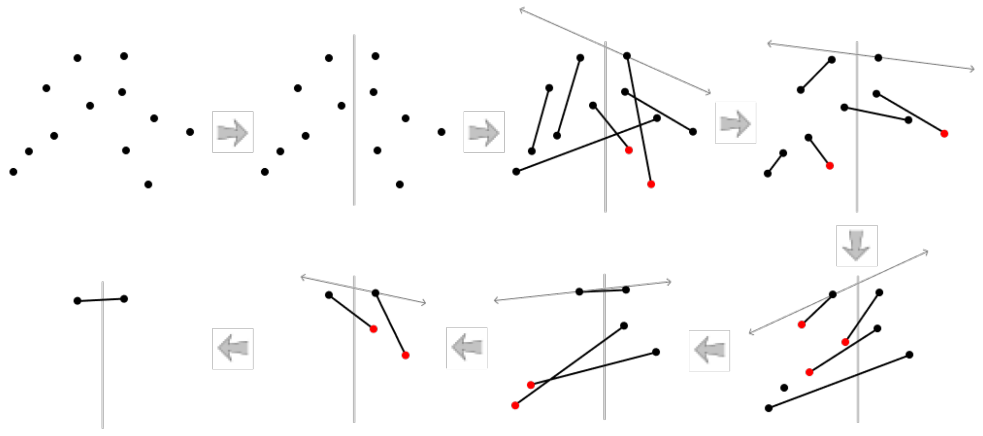


Figure 13: Computing the bridge with the *Prune and Search* technique

These remarks suggest how the *Prune and Search* method works. The algorithm randomly pairs the given points and finds the median slope α of the $\frac{n}{2}$ segments defined by those pairs. Then it computes a line L of slope α that passes through a point p_t such that it is an upper supporting line for the set of points. Because α is the median slope, then half pairs have slope less than α, and half have greater slope. To prune the set of possible bridge points, the algorithm iterates through all the pairs rs and removes r if $x(p_t) \geq x_m$ and $slope(\overline{rs}) < \alpha$ or removes s if $x(p_t) < x_m$ and $slope(\overline{rs}) > \alpha$. The Figure 13 shows an example of how the bridge is computed by the *Prune and Search* technique. Therefore, after every iteration $\frac{1}{4}$ of the remaining points are removed from consideration as bridge points [Souvaine, 2005].

Algorithm analysis

Given a set of n points, the Kirkpatrick–Seidel algorithm computes the convex hull in $O(n \log h)$ time, where h is the number of convex hull vertices. Thus, the algorithm is output-sensitive: as said before, its running time depends on both the input size and the output size.

Implementation

The important part of the implementation of the Kirkpatrick–Seidel algorithm is how it computes the upper bridge with *Prune and Search* technique. Initially the algorithm pairs the points and this is straightforward and can be done by iterating through all the points. If the number of points is odd then the last point is ignored. During the iteration it is possible to compute the slope of each segment and to compute the average later.

Then, it is needed to find a supporting line for the set of points which passes through a point p_t. Because we know the slope of the line, it is possible to calculate the equation of the line passing through a point. From the equation, it is also possible to compute where the line intersects the y-axis. The point p_t is that point such that when the line passes through it, it intersects the y-axis higher compared when the same line passes through the other points. This approach to compute p_t can be done by iterating through all the points, computing the equation of the line and the intersection with the y-axis.

The C++ implementation of how the Kirkpatrick–Seidel algorithm computes the upper bridge is given below. The approach for the lower bridge is the same but it is simmetrical.

```cpp
std::pair<Point, Point> upper_bridge(const std::vector<Point> &points, int med) {
    // handle simple and borderline cases
    if (points.size() == 2) return std::make_pair(points[0], points[1]);
    if (points.size() == 3) return upper_bridge_three_points(points, med);

    // compute the pairs and the avg slope
    int number_of_pairs = points.size() / 2;
    double slope_avg = 0;
    std::vector<std::pair<int, int>> pairs(points.size() / 2);
    for (int i = 0; i < points.size()-1; i += 2) {
        if (points[i].x < points[i+1].x) pairs[i/2] = std::make_pair(i, i+1);
        else pairs[i/2] = std::make_pair(i+1, i);
        slope_avg += SLOPE(points[pairs[i/2].first], points[pairs[i/2].second]);
    }
    std::vector<Point> remaining;
    if (points.size() % 2 == 1) {
        // one point is alone, count it but with a slope = 0
        number_of_pairs++;
        remaining.push_back(points.back());
    }
    slope_avg = slope_avg / number_of_pairs;

    // find max_c and top point
    double max_c = INT_MIN, c;
    Point top_point;
    for (auto pt: points) {
        // Given the point (xp, yp), y - yp = m(x - xp) => if x=0 then y = yp - m*xp
        c = pt.y - (slope_avg * pt.x);
        if (c > max_c) {
            max_c = c;
            top_point = pt;
        }
    }

    // prune phase
    for (auto pair: pairs) {
        bool prune = false;
        if (top_point.x < med) {
            if (SLOPE(points[pair.first], points[pair.second]) >= slope_avg) {
                remaining.push_back(points[pair.second]);
                prune = true;
            }
        } else {
            if (SLOPE(points[pair.first], points[pair.second]) < slope_avg) {
                remaining.push_back(points[pair.first]);
                prune = true;
            }
        }
        if (!prune) {
            remaining.push_back(points[pair.first]);
            remaining.push_back(points[pair.second]);
        }
    }

    return upper_bridge(remaining, med);
}
```

Code 10: C++ code for finding the upper bridge

2.8 Chan's algorithm

The Chan's algorithm, named after Timothy M. Chan, who published it in 1993, is an optimal output-sensitive algorithm to compute the convex hull of a set of points, in 2 or 3-dimensional space. It is a combination of divide and conquer and gift wrapping.

The algorithm works by dividing the n points into n/m subsets, each of size at most m, and computing the convex hull of each subset using Graham's scan (for example).

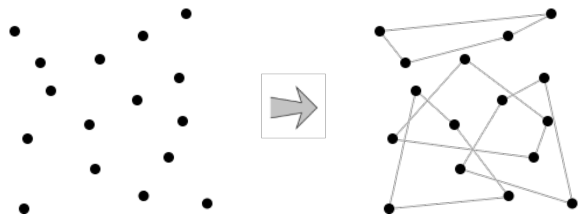


Figure 14: Example of subhulls. Here the set of points is divided into 4 subsets. Note that the subset can intersect each other or not. Figure inspired by [Erickson, 2004]

After computing the n/m subhulls, Chan's algorithm applies Jarvis's march (gift wrapping) algorithm: starting with the rightmost point l, the algorithm successively finds the convex hull point that follows the last computed convex hull point in a counter-clockwise order until it returns to l again.

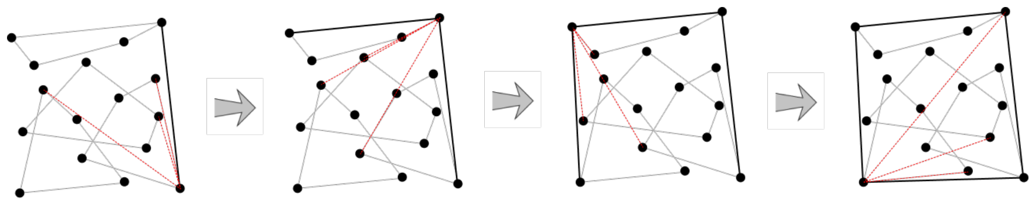


Figure 15: Wrapping the subhulls. The dashed red lines are the other candidates other than the selected one. Figure inspired by [Erickson, 2004]

Timothy Chan in his paper suggests a binary search approach to be used by the Jarvis's march algorithm to efficiently wrap all the subhulls [Chan, 1996]. At each step, given the last computed convex hull point p_i, Jarvis's march needs to find a point p_{i+1} that follows p_i in the convex hull in counter-clockwise order. More formally, it needs to find a point p_{i+1} such that all other points are to the right of the line $\overline{\mathrm{p_i p_{i+1}}}$. The point p_{i+1} is part of one of the n/m subhulls, so the algorithm has n/m possible candidates, one for each subhull. The m points in a subhull are sorted in counter-clockwise order and this allows to use binary search to compute the subhull's candidate point in logaritmic time. At this point, the algorithm can efficiently determine p_{i+1} with the Jarvis's march algorithm, by only considering the n/m possible candidates instead of the whole set of points.

```
ChanAlgorithm($S, m, H$)
1   $P_i, ..., P_{n/m}$ = partition S into $n/m$ subsets
2   for $i = 0$ to $n/m$
3       $CH_i$ = GrahamScan($P_i$) // the vertices are stored in an array in ccw order
4   $p_0 = (0, -\infty)$
5   $p_1$ = rightmost point of $S$
6   for $k = 1$ to $H$
7       for $i = 0$ to $\lceil n/m \rceil$
8           compute $q_i \in P_i$ that maximizes the angle $\overline{\mathrm{p_{k-1} p_k q_i}}$ by binary searching on the
            points in $P_i$
9       $p_{k+1}$ = the point $q$ from $\{q_1, ..., q_{\lceil n/m \rceil}\}$ that maximizes the angle $\overline{\mathrm{p_{k-1} p_k q}}$
10      if $p_{k+1} = p_1$
11          return $\{p_1, ..., p_k\}$
12  return incomplete
```

Guessing the value of m

Since the value of h is not known in advance, the idea is to make multiple passes of the algorithm with increasing values of m. It is important to increase the value properly: if m increases too slowly, the number of iterations may be large; if it increases too quickly, the algorithm terminates successfully but m may be much larger than h and it will produce a time complexity much higher than $O(nlogh)$. One possible strategy is to square the value of m starting from a value of 2. With this squaring strategy, $O(loglogh)$ iterations are made. The following pseudocode shows the described approach.

```
ChanAlgorithm($S$)
1  for $t = 1, 2, ...$
2      $L$ = ChanAlgorithm($S, m, H$), where $m = H = min\{2^{2^t}, n\}$
3      if $L \neq incomplete$
4          return $L$
```

Algorithm analysis

The algorithm can be subdivided into two main phases. The first phase, which consists of partitioning the set of points and computing each subhull, takes $O(\frac{n}{m} m \log m) = O(n \log m)$ time, because each subset has m points, the subhull is computed in $O(m \log m)$ time and we have n/m subsets.

The second and last phase consists of wrapping the subhulls with the Jarvis's march algorithm and a binary search approach. Because of the way the algorithm works, Jarvis's march computes the convex hull in $O(h)$ iterations, where h is the number of convex hull points. For each iteration, to find the the next convex hull point, the algorithm binary searches each subhull to find n/m possible candidates. This search takes $O(\frac{n}{m} \log m)$ time, because binary searching in one subhull costs time $O(\log m)$ and we have n/m subhulls. So, the second phase takes $O(\frac{n}{m} h \log m)$. If m is close to h then the overall time complexity of Chan's algorithm is equivalent to $O(n \log h)$.

To guess the value of m, at each iteration the algorithm squares the value of m. The number of iterations is $O(\lceil \log \log h \rceil)$, because it stops when the value of H reaches or exceeds h. An iteration t has a cost of $O(n \log H) = O(n2^t)$ time (using base-2 logarithms). Therefore, the total running time of the algorithm is $O(\sum_{t=1}^{\lceil \log \log h \rceil} n2^t) = O(n2^{\lceil \log \log h \rceil + 1}) = O(n \log h)$.

Implementation

Before looking at how to implement the Chan's algorithm, let's have a look to how the simple binary search works. The c++ implementation is given below:

```cpp
Point next_candidate(const std::vector<Point> &subhull, const Point &last) {
    unsigned int curr = 0, inc = subhull.size()/2, next, prev;
    Point next_point = subhull[0];
    bool found = false;
    while(!found) {
        next = (curr + 1) % subhull.size();
        prev = (curr - 1) % subhull.size();
        if (orientation(last, subhull[curr], subhull[next]) == CW) {
            curr = (curr + inc) % subhull.size();
        } else if (orientation(last, subhull[curr], subhull[prev]) == CW) {
            curr = (curr - inc) % subhull.size();
        } else {
            next_point = subhull[curr];
            found = true;
        }
        if (inc > 2) inc /= 2;
    }

    return next_point;
}
```

Code 11: Binary search the next candidate from the given subhull

Given the subhull and the last added point into the convex hull, the function binary searches for the next possible candidate. The points in the subhull are sorted in counter-clockwise order. So, it starts from the first element and goes forward when it is needed to go counter-clockwise otherwise it goes backward. Instead of iterating through all the points, the algorithm goes forward/backward a value multiple of two which is smaller after each iteration until if finds the best next candidate.

Then, the C++ implementation of the Chan's algorithm is the following. It calls the Graham Scan function to compute the subhulls and it uses the `next_candidate()` to binary search the next possible candidate from each subhull.

```cpp
std::vector<Point> chan(const std::vector<Point> &points, int m, int h) {
    unsigned int subset_size = std::round((double) points.size() / m);
    std::vector<std::vector<Point>> pts(m);
    std::vector<std::vector<Point>> ch(m);
    // subdivide points into m subsets
    int j = 0;
    for (int i = 1; i <= points.size(); ++i) {
        pts[j].push_back(points[i-1]);
        if (i % subset_size == 0 && j + 1 < m)
            j++;
    }

    // graham scan for each subhull
    for (int i = 0; i < m; ++i) ch[i] = grahamscan(pts[i]);

    std::vector<Point> finalhull { rightmost_point(points) };
    for (int k = 0; k < h; ++k) {
        Point best{ INT_MIN, INT_MIN };
        // jarvis march phase
        for (int i = 0; i < m; ++i) {
            // do not compare last point with itself when the set has only the last point
            if (ch[i].size() == 1 && ch[i][0] == finalhull.back()) continue;
            // find the next candidate via binary search and compare with the best found
            Point next = next_candidate(ch[i], finalhull.back());
            if (best == Point{ INT_MIN, INT_MIN } ||
                orientation(finalhull.back(), best, next) == CW)
                best = next;
        }

        if (best == finalhull[0]) return finalhull;
        finalhull.push_back(best);
    }

    return {};
}
```

Code 12: C++ implementation of Chan's algorithm

It is important to note that, the jarvis phase does not differ too much from the real Jarvis's march algorithm, because the approach is still the same. Thanks to the binary search phase, the number of candidates is lower for each iteration and the algorithm runs efficiently.

3 Higher dimensions

As seen before, in 2 dimensions the convex hull is a convex polygon. A convex polygon is the 2-dimensional example of a polytope, which is a geometric object with flat sides (called faces). In any number of dimensions, the convex hull is a convex polytope. In 3 dimensions, the convex hull is a convex polyhedron, which is a 3-dimensional example of a polytope.

Some algorithms that can compute the convex hull of a set of points in 2-dimensional space can also be extended to work on 3 or higher dimensions.The following are the algorithms that, with the right approach, can compute the convex hull in higher dimensions:

- Incremental convex hull, which is slower but preferred in practice
- QuickHull, which is slower but preferred in practice for dimensions higher than 3
- Divide and conquer, which is elegant but difficult to implement
- Jarvis's march, which is used by the divide and conquer algorithm
- Chan's algorithm, that is preferred in 3 dimensions

3.1 Visibility

Like in the 2-dimensional case, in higher dimensions there is also the need to understand the concept of visibility: Given some points, what are the points that can "see" a given face f of the polytope. For simplicity we can think about a 3-dimensional polytope, the polyhedron. To answer this question we can take the plane h_f that expands by the face f and, if the segment from p to f doesn't pass through any other face of the polytope, then the point is visible from the face. Otherwise, if the segment passes through another face, the point is not visible from the face.

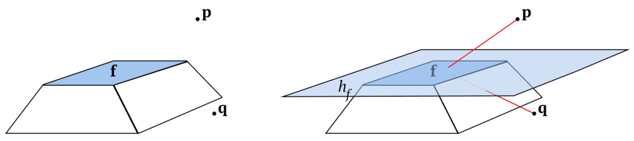


Figure 16: The face f is visible from the point p but not from the point q: the red segment from the face to p does not pass through the polyhedron; the segment from the face to q passes through another face of the polyhedron. This figure is inspired by the book [de Berg et al., 2008].

Given a polytope and a point r, we can then know the set of faces that are visible from the point, and the set of the faces that cannot be seen by the point. To do this we can draw a ray from the point to any other point of the convex hull. For some points, the ray will go inside the polytope but for some other points the ray will just touch the point without going inside the polytope. Such points are the last points that can the point can see. If we look at the projection to a plane we will get a polygon which corresponds to the edge of such points. The edges are called horizon and are the last edges that the point can see.

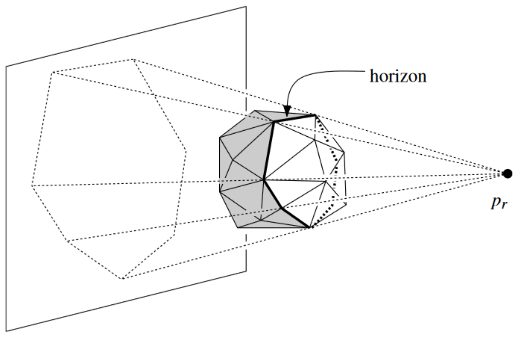


Figure 17: The dashed lines are the rays from the point p_r. The black edges are the horizon. The white faces are the faces visible from the point p_r while the grey faces are on the other side of the horizon, so are not visible. This figure is from the book [de Berg et al., 2008].

3.2 Conflict graph

Given a set of points, the conflict graph allows to keep track of what faces are visible from a point that has not yet been processed. We have a vertex for each point and a vertex for every face of the convex hull. We have an edge between a point p and a face f if the face is visible from the point. The edges are called conflicts because if we want to add p to the convex hull, we have to remove f and all the faces that p can see and we have to add the edges from p to any point in p's horizon.

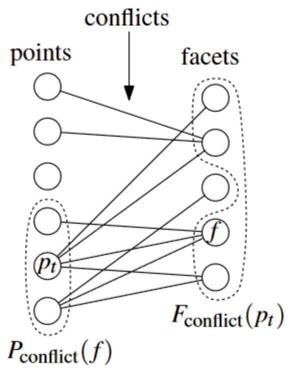


Figure 18: Graphical representation of a conflict graph: the points are the left nodes, the faces are the right nodes and the edges are the conflicts. Here the point p_t can see the face f and other 3 faces, while the face f is visible from p_t and other 2 points. Figure from the book [de Berg et al., 2008].

Given a point p_t and a face f, we can define two useful functions, $P_{conflict}(f)$ and $F_{conflict}(p_t)$. $F_{conflict}(p_t)$ is the set of faces that are visible from the point p_t. $P_{conflict}(f)$ is the set of points that can see the face f.

3.3 Incremental algorithm

The Incremental convex hull algorithm can be extended to compute the convex hull of a set of points in N-dimensions. It is slower than the others but it is preferred in practice. The algorithm starts with a minimal convex hull (for example in 3D is a tetrahedron), then iterates through all the points and adds that points that lie outside of the convex hull. To work in higher dimensions, the approach differs in some steps. First thing first, the algorithm doesn't initially sort the points because it selects the next point randomly. Then, for each point, it checks if the point is outside the convex hull by computing the visible faces from the point.

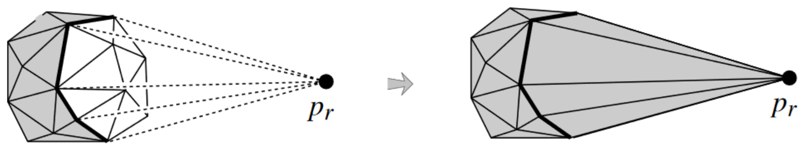


Figure 19: Example of one iteration of the incremental convex hull algorithm applied to a 3-dimensional set of points. The figure is inspired by the book [de Berg et al., 2008].

If the point lies outside, then it is added to the convex hull, the edges connecting the point with the convex hull are also added to it and the faces that became inside the convex hull are removed. The algorithm stops until there are no remaining points. The key of the algorithm is how it computes the faces visible from a point and how it stores the points and the faces to efficiently compute the faces to be removed when a point is added to the hull.

3.4 QuickHull

N-dimensional Quickhull was invented in 1996 by C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa [C. Bradford Barber, 1996]. It does not differ too much from the 2-dimensional case but it is an extension of it. As we know the algorithm begins by computing an initial hull. It does so in higher dimensions but, for example in 3D, it initially computes a triangle, like in the 2D version, and then it finds the furthest point from the triangle plane. This creates an initial hull which has the shape of a tetrahedron.

Then the algorithm partitions the remaining points into the conflict lists of the faces of the initial hull. It iteratively adds new points to the hull by selecting the points with the largest distance from the conflicts. This involves the computation of the selected point's horizon. For each horizon edge, the algorithm creates a face and adds it to the hull. At this step some faces have been closed inside the convex hull, so the algorithm removes them. It continues selecting the points until there are no remaining points.

3.5 Chan's algorithm

Chan's paper about its algorithm for computing the convex hull also describes the approach for higher dimensions. As we know, Chan's algorithm uses Jarvis's march. In higher dimensions it uses higher-dimensional analogue of the gift wrapping algorithm. However, the algorithm in 2-dimensional space also uses Graham's scan, which cannot be extended to work in higher dimensions. It is replaced by the higher dimensions version of the Divide and Conquer algorithm by Preparata. The rest of the algorithm and the overall idea behind it remains unchanged.

3.6 Divide and conquer

The Divide and conquer algorithm by Preparata doesn't change too much when it is extended to compute the convex hull of a set of points in N-dimensions. As usual, the algorithm splits the point set into two sets, calls itself recursively in both sets and the merges the results. The following figure is a graphical representation of computing the convex hull in 3-dimensions.

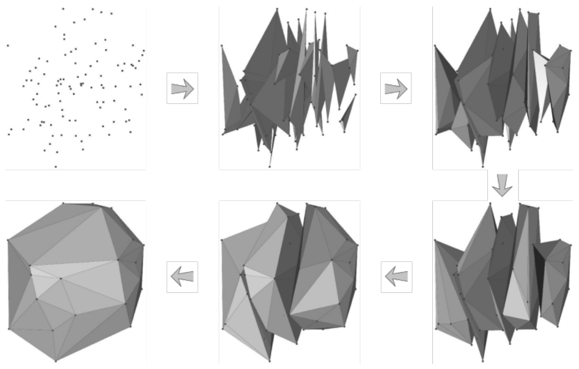


Figure 20: Example of Divide and conquer in 3-dimensions. Images from [Kazhdan, 2016]

The key step of the algorithm is the merging phase. In higher dimensions it uses the gift wrapping approach as shown in the next figure. Initially it determines a supporting line of the convex hulls, by projecting the hulls in 2-dimensions and using the 2D algorithm. Then it uses the gift wrapping approach to create the additional faces to connect the subhulls. Finally, it removes the faces that became hidden.

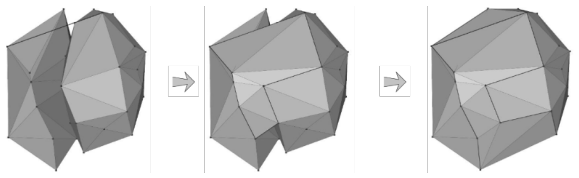


Figure 21: Merging phase of the divide and conquer algorithm in 3D. Images from [Kazhdan, 2016]

4 Competitive programming problems

4.1 Wall (Easy)

This is a simple competitive programming problem from the 2001-2002 ACM Northeastern European Regional Programming Contest and it available online on Timus Online Judge website [Sergey Volchenkov, 2001].

Problem description

A King ordered his chief Architect to build a wall around the King's castle. The King wants to build the wall around the whole castle using the least amount of stone and labor, but demanded that the wall should not come closer to the castle than a certain distance. The King demanded Architect to give the exact amount of resources that are needed to build the wall. The task is to find the minimum possible length of the wall that the Architect could build around the castle to satisfy King's requirements. The task is somewhat simplified by the fact, that the King's castle has a polygonal shape and is situated on a flat ground. The Architect has already established a Cartesian coordinate system and has precisely measured the coordinates of all castle's vertices in feet.

The input consists of N pairs that represents the coordinates of the vertices in the King's castle and an integer which is the minimal number of feet that King allows for the wall to come close to the castle. All vertices are different and the sides of the castle do not intersect anywhere except for vertices.

Problem solution

The castle is represented by a set of points, where each point is a vertex of the castle. In other words, this set of points is a polygon which can be convex or maybe not. It is easy to see that the optimal solution is the convex polygon that encloses the castle polygon. The reason is that the convex polygon in some way generalizes the shape of the castle with possibly less vertexes, walls, corners, etc. Computing it is, of course, computing the convex hull of the set of the castle's vertices and can be done with any algorithm.

Then, the remaining step is to "enlarge" the convex hull of the castle by the minimal amount desired by the King. Iterating through all the convex hull points and computing the sum of the distances between consecutive points is the optimal solution of the problem.

4.2 Fence Orthogonality (Hard)

This competitive programming problem is from the CTU Open 2013 contest and it is available online at open.kattis.com [Kattis, 2013].

Problem description

Evil bunnies are eating Freddy's vegetables. In order to stop them, he decided to build a fence enclosing all vegetables in his garden. Freddy wants the fence to be as cheap (i.e., short) as possible, but for technical reasons, he can only build rectangular fences. For simplicity, we will assume the vegetables are negligibly small and can be represented by points in a two-dimensional plane.

The input consists of a first line with one integer N giving the number of vegetables in the garden. Each of the following N lines contains two integers, giving the coordinates of one vegetable to be protected. No two vegetables have the same coordinates. You may also assume the vegetables are not all on the same straight line.

Problem solution

Given the vegetables in the garden as 2D points, the shortest fence enclosing all of them is of course their convex hull. But, if the fence must be a rectangle then it is the bounding box of the vegetables. A naïve approach is to simply take the smallest and largest x-coordinates e y-coordinates of the vegetables. This will certainly produce a rectangular bounding box. But, this approach computes a bounding box orthogonal to the axes and it is not always the minimum perimeter bounding box. By rotating the bounding box to any arbitrary angle, it is possible to find a rectangle that has the smaller perimeter.

It has been shown [H.Freeman, 1975] that the minimum area bounding box of a set of points is collinear with one of the edges of the set's convex hull polygon.

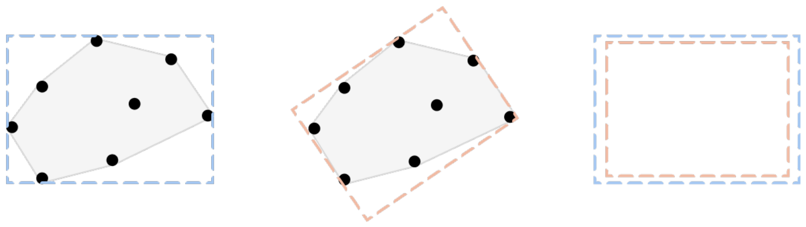


Figure 22: Both rectangles are bounding boxes of the given points. The blue bounding box is orthogonal to the axes, but it has higher perimeter and area compared to the orange one. The minimum perimeter/area bounding box is also collinear with an edge of the convex hull.

The Rotating Calipers technique is a versatile method for solving a number of problems from the field of computational geometry and it can be used to solve this competitive programming problem. It is used to compute the diameter of convex polygons, minimum bounding box, minimum and maximum distance between two convex polygons, the intersection of convex polygons and many things more.

The first thing to do is to compute the convex hull of the vegetables and to find the bounding box orthogonal to the axes by computing the smallest and largest x-coordinates and y-coordinates. This also gives the initial minimum perimeter P_{min}. More formally, this bounding box is made by four supporting lines, each for each side. Then, rotate the supporting lines until one coincides with an edge of the convex hull, compute its perimeter and update the minimum perimeter P_{min} and store the current rectangle if its perimeter is smaller than P_{min}. Repeat until any convex hull edge coincided at least one time with one of the supporting lines.

The algorithm iterates through all the points and the computation for each iteration takes $O(1)$ time. Rotating can be done efficiently because at any step you know the next edge of the convex hull. Computing the perimeter is also trivial because at any step is known the base and height of the bounding box. So, the Rotating Calipers technique takes linear time to compute the minimal bounding box plus the time needed to compute the convex hull.

5 Conclusion

Not only in geometry and in competitive programming, but also in many other fields it is important to know about the convex hull and some of the strategies used by some algorithms can used to solve many other problems. The convex hull itself of a set of N-dimensional points gives a powerful tool to extrapolate many details about the set of points. Because of that, it has been studied for years by computer scientists and nowadays the convex hull is fundamental for many more computational geometry problems. The computation can be done with different approaches and there isn't one better than the others. For sure, it all depends of the intrinsic characteristics of the set of points and their dimensionality.

The implementation of each algorithm is more or less trivial but it is important to efficiently store the convex hull and be as much smart as possible to lower the number of comparisons between points. Each algorithm has also its peculiarities and it is not possible to implement any of them by just looking to the pseudocode. The reason is that, because the problem is a geometric problem, it is needed a deep understanding of exactly what is happening and how during the iteration of each algorithm. Because of that, the pseudocode is a good starting point to understand how the algorithm compute the convex hull, but looking at an example visualization of each iteration of the algorithm is very important and it gives a deep and more clear understaing of how the algorithm works.

In the competitive programming world, a problem that just asks to compute the convex hull is not common because is a trivial problem: to solve it, it is just needed to know at least one algorithm for computing the convex hull. But, in competitive programming there are tons of problems that are geometric problems. To solve many of them it is required to compute the convex hull as the first or second step. This is why the convex hull problem is important for the competitive programming community.

Together with this report comes also a GitHub repository with all the source code of each algorithm, written in C++. It is available at https://github.com/domferr/competitive-programming/tree/main/TheConvexHull. Almost the entire source code can be read in this report, but the GitHub repository also contains a testsuite. Any algorithm has been unit tested with the testsuite by using the GoogleTest framework.

This report, its figures and the source code can also be a starting point for many other future projects. For example a web visualization of the computation of each algorithm, iteration after iteration. But also, extending the source code to compute the convex hull of 3-dimensional or N-dimensional set of points.

References

[C. Bradford Barber, 1996] C. Bradford Barber, David P. Dobkin, H. H. (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22:469–483.

[Chan, 1996] Chan, T. M. (1996). Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(3):361–368.

[Chen, 2019] Chen, J. (2019). Prune and search. [Online; accessed 20-January-2022].

[David G. Kirkpatrick, 1986] David G. Kirkpatrick, R. S. (1986). The ultimate planar convex hull algorithm? *Siam J. Comput.*, 15(1):287–299.

[de Berg et al., 2008] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry. Algorithms and Applications.* Springer, 3 edition.

[Erickson, 2004] Erickson, J. (2004). Teaching notes about convex hulls. [Online; accessed 19-January-2022].

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

[Gang Mei, 2012] Gang Mei, John C.Tipper, N. X. (2012). An algorithm for finding convex hulls of planar point sets. *2nd International Conference*, pages 888–891.

[H.Freeman, 1975] H.Freeman, R. (1975). Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Communications of the ACM*, 18:409–413.

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

[Kattis, 2013] Kattis (2013). Fence orthogonality. [Online; accessed 30-January-2022].

[Kazhdan, 2016] Kazhdan, M. (2016). Convex hulls (3d). [Online; accessed 25-January-2022].

[Sergey Volchenkov, 2001] Sergey Volchenkov, R. E. (2001). Wall. [Online; accessed 30-January-2022].

[Souvaine, 2005] Souvaine, D. (2005). The kirkpatrick-seidel marriage before conquest algorithm. [Online; accessed 25-January-2022].

[Wikipedia, 2021] Wikipedia (2021). Output-sensitive algorithm — wikipedia, the free encyclopedia. [Online; accessed 19-January-2022].