A note on the implementation quality of a convex-hull algorithm

Ask Neve Gamby   Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. This note is a critical commentary of the recent paper entitled "A total order heuristic-based convex hull algorithm for points in the plane" published in *Computer-Aided Design* **70** (2016), 153–160. In the abstract of that paper it is concluded that the presented algorithm is (1) "much faster" than its competitors (2) "without penalties in memory space". In our opinion, the paper is well written, but in this note we raise a few objections to these conclusions.

Keywords. Computational geometry; convex hulls; robustness; performance.

1. Introduction

In physics, the term *observer effect* is used to refer to the fact that simply observing a phenomenon necessarily changes that phenomenon. A similar effect comes up in computer science when comparing different algorithms—one has to implement them and make measurements about their performance. In an ideal situation, one would eliminate the bias caused by quality of implementations and limit the uncertainty caused by inaccurate measurements [34, Chapter 14]. In practice, this is difficult. This note is part of our research program "quality of implementations" where we put some recent algorithm-engineering studies under close scrutiny. In particular, we want to remind our colleagues of the fact that some experiments do not tell the whole truth and that the quality of the implementation does matter.

In this note, we investigate the efficiency of algorithms that solve the convex-hull problem in the two-dimensional Euclidean space. As our starting point, we used the recent article by Gomes [21]. In this article, a variation of the PLANE-SWEEP algorithm by Andrew [2] was presented, analysed, and experimentally compared to several other well-known algorithms. The algorithm was named TORCH (total-order convex-hull) and—what was delightful—its source code was made publicly available [20].

In the remaining part of this note, we briefly describe the algorithm and the competitors used in the experiments in [21]. In particular, we go a bit deeper into the implementation details that are only revealed by a careful code analysis. Then we present the results of some alternative experiments and the totally new set of conclusions. All the programs used in this study

CPH STL Report 2017-3, December 2017. Revised January 2018, February 2018.

are publicly available [18]. We will let the readers make their own judgement on the actual truth in this matter. Perhaps, even further experimentation may be necessary.

2. Preliminary definitions

Recall that in the two-dimensional convex-hull problem we are given a sequence $S = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ of points, where each point p is specified by its Cartesian coordinates $(p.x, p.y)$, and the task is to compute the *convex hull* $\mathcal{H}(S)$ of S which is the boundary of the smallest convex set containing all the points of S. More precisely, the goal is to find the smallest set—in cardinality—of vertices describing the convex hull, i.e. the output is a convex polygon P containing all the points of S. In particular, the set of vertices in P is a subset of the points in S. A normal requirement is that in the output the vertices are sorted according to their polar angles around one arbitrary interior point. Throughout the paper, we use n to denote the number of points in the input sequence S and h the number of vertices in the output polygon P.

A point q of a convex set X is said to be an *extreme point* if the following condition is satisfied:

$$\nexists \{p, r\} \subset X \setminus \{q\} : a \cdot p + (1 - a) \cdot r = q \text{ for a real number } a, 0 < a < 1.$$

In other words, no two other points p and r exist such that q lies on the line segment $\overline{pr}$. By this definition, the vertices of P describing the convex hull $\mathcal{H}(S)$ are precisely the extreme points of the smallest convex completion containing the points of S.

Let S be a sequence of points. Of these points, a point p *dominates* another point q in the north-east direction if $p.x > q.x$ and $p.y > q.y$. A point p is *dominant* in this particular direction if it is not dominated by any other point in S. That is, the first quadrant centred at p has no other point in it. Analogous definitions can be made for the south-east, south-west, and north-west directions. A point is said to be *maximal* among the points in S if it is dominant in one of the four directions. Clearly, every extreme point is maximal (for a proof, see [6]), but the opposite is not necessarily true.

An algorithm operates *in-place* if it only uses $O(1)$ words of memory in addition to the input sequence. An in-place convex-hull algorithm (see, for example, [8]) partitions the input sequence into two parts: (1) the first part contains all the extreme points in clockwise or counterclockwise order of their appearance on P and (2) the second part contains all the remaining points that are inside P or on the edges of P. Furthermore, an algorithm is said to operate *in-situ* if, for an input sequence of size n, it uses $O(\lg n)$ words of additional memory. For example, QUICKSORT is an in-situ algorithm if it is implemented recursively such that the smaller subproblem is always solved first.

The *orientation test* is the basic geometric primitive that is used in many of the algorithms to be discussed. It takes three points p, q, r as its arguments and tells whether there is a left turn, right turn, or neither of these at q when moving from p to r via q. This primitive can be seen as a special cross-product computation, and it is a known improvement used to avoid trigonometric functions and division (mentioned, for example, in [1]).

3. Torch algorithm

The plane-sweep algorithm, proposed by Andrew [2], computes the convex hull for a sequence of points as follows:

(1) Sort the input points into non-decreasing order according to their x-coordinates.
(2) Determine the leftmost point (*west pole*) and the rightmost point (*east pole*). If there are several pole candidates with the same x-coordinate, let the bottommost point be the west pole; and on the other end, let the topmost point be the east pole.
(3) Partition the sorted sequence into two parts, the first containing those points that are above or on the line segment determined by the west pole and the east pole, and the second containing the points below that line segment.
(4) Scan the two sorted chains of points separately by eliminating all points that are not vertices of the convex hull. This step is similar to that used in Graham's rotational-sweep algorithm [22] and it involves about $2n$ orientation tests in the worst case.
(5) Concatenate the computed partial hulls to form the final answer.

Andrew [2] was careful to point out that in Step (3) it is only necessary to perform the orientation test for points whose y-coordinate is between the y-coordinates of the west and east poles. Here, the number of orientation tests performed is often much smaller than n.

The main algorithmic idea in the torch algorithm is to compute the maximal points before performing any orientation tests. Compared to the plane-sweep algorithm, Steps (3) and (4) are now replaced with the following three steps:

(2') Find the topmost point (*north pole*) and the bottommost point (*south pole*) by one additional scan and resolve the ties in a similar fashion as in Step (2).
(3') Partition the sorted sequence into four (overlapping) parts: these form the candidates for the north-east, south-east, south-west, and north-west hulls.
(4') In each of the candidate collections, compute the maximal points before eliminating the points that are not in the output polygon.

In Step (3'), to do the partitioning, it is only necessary to compare the coordinates, the orientation test is not needed. In the original description [21], the chains of maximal points were concatenated before the final elimination scan. However, doing so may cause the algorithm to malfunction—we will get back to the correctness of this approach in Section 7.

The computational complexity of this algorithm is dominated by the sorting step which requires $\Theta(n \lg n)$ time in the worst case. For many input distributions, the number of maximal points is small (see, for example, [6, Theorem 2]). If this is the case, after this improvement, the cost incurred by the evaluation of the orientation tests will be insignificant. Compared to other alternatives, this algorithm is important for two reasons:

(1) Only the x-coordinates are compared when sorting the points.

(2) The orientation test is executed at most twice per maximal point.

There is a catch that is somehow hidden in the original article [21]: In Step (3'), the partitioning procedure must retain the sorted order of the points, i.e. it must be *stable*. It is possible to do stable partitioning in-place in linear time (see [26]), but all linear-time algorithms developed for this purpose are known to be galactic—that is, they are not useful in this galaxy. Therefore, it is understandable that Gomes' implementation [20] uses four extra arrays of indices for this partitioning process.

Consider now a special input having four poles $(-1, 0)$, $(0, 1)$, $(1, 0)$, $(0, -1)$ and $n - 4$ points at the origin. In this case, none of the points at the origin are dominated by any of the poles. Therefore, all these duplicates should be considered as candidates when constructing the lateral hulls at all of the four directions. Hence, each of the index arrays will be of size $n - 4$. Furthermore, two other arrays of points were used: one to store the approximate convex hull and another (stack) to store the output. Since all the points are maximal, none of them will be eliminated. Hence, the approximate hull may contain about $4n$ points. Since all extra arrays were dynamic (C++ standard-library vectors or deques), the total amount of allocated space can be two times larger than what is actually needed (see, for example, the benchmarks reported in [24]). In the worst-case scenario, the amount of space allocated by Gomes' implementation may sum up to about $8n$ indices, $\frac{1}{256}h$ pointers (the page-table index of the deque), and $8n + h$ points. Hence, we strongly disagree with the claim "without penalties in memory space".

It should be emphasized that the maxima-first heuristics only reduces the number of orientation tests executed in the the average case. There is no asymptotic improvement in the overall running time—in the worst case or in the average case. The PLANE-SWEEP algorithm may execute the orientation test up to about $3n$ times. For the special input described above, Gomes' implementation may execute the orientation test at least $4n - O(1)$ times, so in the worst case there is no improvement in this respect either. Hence, we also disagree with the claim "much faster".

```
template<typename T>
bool left_turn(point<T> const& p, point<T> const& q, point<T>
  ↪ const& r) {
  using N = decltype(sizeof(T));
  constexpr N w = cphmpl::width<T>;
  using Z = cphstl::ℤ<2 * w + 2>;

  Z mid_x = Z(q.x);
  Z mid_y = Z(q.y);
  Z δ_x = mid_x − Z(p.x);
  Z δ_y = mid_y − Z(p.y);
  Z Δ_x = Z(r.x) − mid_x;
  Z Δ_y = Z(r.y) − mid_y;
  Z lhs = δ_x * Δ_y;
  Z rhs = Δ_x * δ_y;
  return lhs > rhs;
}
```

Figure 1. The left-turn predicate written in C++. The compile-time function `cphmpl::width<T>` returns the width of the coordinates of type `T` in bits. The class template `cphstl::ℤ<b>` provides fixed-width integers; the width of the numbers is `b` bits and, for numbers of this type, the same operations are supported as for built-in integers.

4. Robustness

In the experiments carried out by Gomes [21], the coordinates of the points were floating-point numbers. It is well-known that the use of the floating-point arithmetic in the implementation of the basic geometric primitives may lead to disasters (for some classroom examples—one being the computation of the convex hulls in the plane, we refer to [27]). Therefore, it was somewhat surprising that the robustness issue was not discussed at all.

In our experiments we rely on the exact-arithmetic paradigm. The coordinates of the points are assumed to be integers of fixed width, and all intermediate calculations are carried out with integers of higher precision so that all overflows and underflows can be avoided. Luckily, when computing the convex hulls in the plane, it is quite easy to predict the maximum precision needed in the intermediate calculations. Every addition or subtraction increases the needed precision by one bit and every multiplication doubles the needed precision.

As an example, let us consider the left-turn orientation predicate. Using the off-the-shelf tools available at the CPH STL [13] and the CPH MPL [25], the C++ code for computing the left-turn predicate is shown in Figure 1. If the width of the original coordinates is `w` bits, with $(2\mathtt{w} + 2)$-bit intermediate precision we can be sure that no overflows or underflows happen—and the computed answer is correct. There are many other—more advanced—methods to make the computation robust (for a survey, we refer to [31]).

5. Baselines

A normal advice given in most textbooks on algorithm engineering is that "it is essential to identify an appropriate *baseline*" for the experiments (the quotation is from [34, Chapter 14]). In the following subsections we will comment the competitors selected by Gomes [21] for his experiments.

5.1 Rotational-sweep

This historically important algorithm due to Graham [22] is a natural competitor. It solves the convex-hull problem as follows: (1) Find a point o that is on the convex hull or inside it. (2) Sort all the points around o lexicographically by polar angle and distance from o. (3) Scan the formed star-shaped polygon in clockwise order by repeatedly considering triples (p, q, r) of consecutive points on this polygon and eliminate q if there is not a right turn at q when moving from p to r via q.

Gomes [21] referred to the implementation described in [19] which closely follows the description given in the textbook by Cormen et al. [12, Section 33.3]. It was a pleasure to read this code, but it had some issues that we would implement differently:

– The routine is not fully in-place since the output is produced on a separate stack.
– The routine has the same robustness issues as Gomes' program: the arithmetic operations in the orientation test and the distance calculation can overflow—which makes its unclear whether the produced output is correct or not.
– The C library qsort is used in sorting; this program is known to be much slower than the C++ standard-library introsort [29]. Gomes used introsort in his own program; if it was not used here, the comparison would be unfair.

The rotational-sweep algorithm is known to call the orientation predicate $\Theta(n \lg n)$ times. Just based on this observation, it has no chance against the torch algorithm. We decided to leave this algorithm outside our experiments since the code used by Gomes was not available for us.

5.2 Plane-sweep

It is known that the plane-sweep algorithm can be implemented in-place (see [8]). Unfortunately, the description given there does not lead to an efficient implementation; in that article the goal was just to show that an in-place implementation is possible.

The problem with stable partitioning can be avoided by performing the computations in a different order: find the west and east poles before sorting. This way, when creating the two candidate collections, we can use unstable in-place partitioning employed in quicksort—for example, the variant proposed by Lomuto [12, Section 7.1]—and then sort the candidate collections

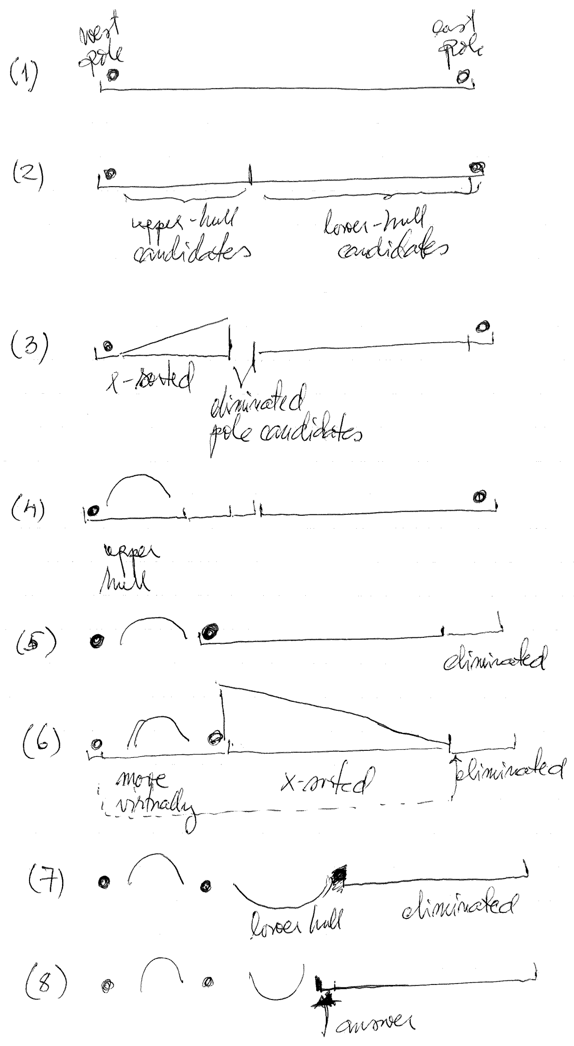

Figure 2. In-place PLANE-SWEEP algorithm illustrated.

separately. Still, we want to be careful to include all the implementation enhancements proposed by Andrew [2].

To summarize, our implementation of the PLANE-SWEEP algorithm functions as follows (for an illustration, see Figure 2):

(1) Find the west and east poles by scanning the input sequence once.
(2) Partition the input sequence into two parts (a) the upper-hull candidates, i.e. the points that are above or on the line segment determined by the west pole and the east pole, and (b) the lower-hull candidates, i.e. the points below that line segment.
(3) Sort the upper-hull candidates into non-decreasing order according to their x-coordinates. Observe that, in the output, the points with equal

x-coordinates can be reported in any order—it is just required that the west pole must be the first point and the east pole must be the last point and, if there are points with the same x-coordinate as the poles, all other except the topmost point above the west pole and the bottom-most below the east pole are eliminated before any further processing.

(4) Scan the upper-hull candidates from left to right and eliminate all points that are not the vertices of the convex hull. In principle, this is just Lomuto's partitioning. This routine is semi-stable, meaning that it retains the order of the elements in the left part of partitioning. The eliminated points can be reported in any order.

(5) Move the eliminated points to the end of the sequence so that the lower-hull candidates become just after the computed upper hull. This is a standard in-place block-swap computation.

(6) Sort the lower-hull candidates into non-increasing order according to their x-coordinates. Temporarily, make the east pole the first point of the lower-hull candidate collection and the west pole its last point.

(7) Scan the lower-hull candidates from left to right and eliminate all points that are not the vertices of the convex hull.

(8) Report the computed partial hulls as the final answer. The return value is just an iterator to the first eliminated point.

If HEAPSORT [33] was used in Step (3) and Step (6), the algorithm would operate in-place and run in $O(n \lg n)$ worst-case time. However, the C++ standard-library INTROSORT [29], which is a combination of QUICKSORT and HEAPSORT, is known to be more efficient in practice. Still, it guarantees the $O(n \lg n)$ worst-case running time and, because of careful programming, it operates in-situ since the recursion stack is never allowed to become deeper than $2 \lg n$.

5.3 QUICKHULL

The QUICKHULL algorithm, which mimics QUICKSORT, has been reinvented several times (see, e.g. [9, 16]). As QUICKSORT, this algorithm is recursive (for a visualization, see Figure 3). In its general recursive step, we are given a line segment determined by two extreme points p and r, and a collection C of points on one side of this line segment (for the sake of concreteness, say above). If the cardinality of C is small, we can use any brute-force method to compute the convex chain from p to r, return that chain, and stop. Otherwise, of the points in the candidate collection C, we find an extreme point q in the direction orthogonal to the given line segment—in the case of ties, we select the leftmost extreme point. Then, we eliminate all the points inside or on the triangle (p, q, r) from further consideration by moving them away to the end of the zone occupied by C. Finally, we invoke the recursive step once for the points of C that are above the line segment determined by p and q, and once for the points of C that are above the line segment determined by q and r. The key feature is that the interior points

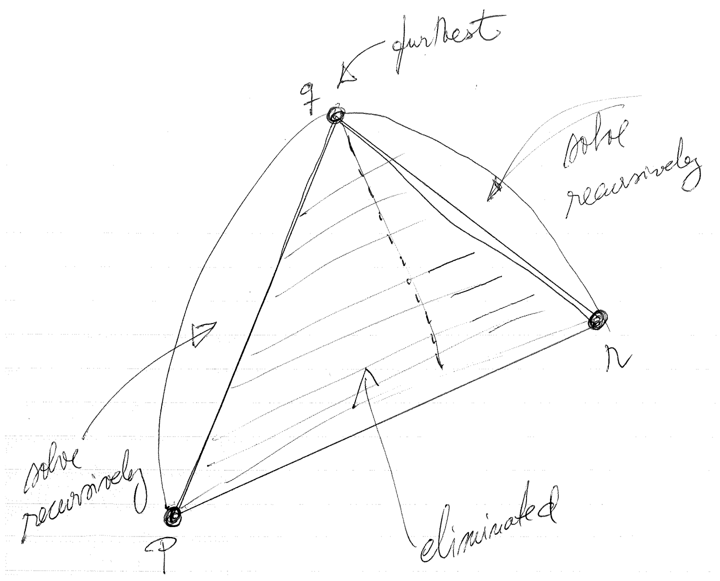

Figure 3. A visualization of the general recursive step in the QUICKHULL algorithm.

are eliminated during sorting—full sorting may not be necessary. Therefore, we expect this algorithm to be fast.

Gomes used the QHULL program [4] in his experiments. However, he did not give any details how he used this library routine so, unfortunately, we have not been able to repeat his experiments. From the QHULL manual [5], one can read that the program is designed to

- solve many geometric problems,
- work in the d-dimensional space for any $d \geq 2$,
- be output sensitive,
- reduce space requirements, and
- handle most of the round-off errors caused by floating-point arithmetic.

Based on this description, we got a feeling that this is like catching a mouse with a canon. Moreover, if this program avoids precision problems and Gomes' program does not, a comparison would be unfair.

Simply, specialized code designed for solving the two-dimensional convex-hull problem—and only that—should be faster. After doing the implementation work, the resulting code was less than 200 lines long. As an extra bonus,

- we had full control over any further tuning and
- we could be sure that the computed output was correct as, also here, we relied on exact integer arithmetic.

To summarize, our implementation of the QUICKHULL algorithm works as follows: (1) As in the PLANE-SWEEP algorithm, find the two poles, the west pole p and the east pole r. (2) Put p first and r last in the sequence. (3) Partition the remaining points between p and r into upper-hull and lower-hull candidates: C_1 contains those points that lie above or on the line segment $\overline{pr}$ and C_2 those that lie below it. (4) Call the elimination routine described above for the upper-hull candidates with the parameters C_1, p, r, and orientation *above*. (5) Move r just after the upper chain computed. (6) Move the eliminated points after the lower-hull candidates. (7) Apply the elimination routine for the lower-hull candidates with the parameters C_2, p, r, and orientation *below*. (8) Now the point sequence is rearranged such that (a) p comes first, (b) the computed upper chain follows it, (c) r comes next, (d) then comes the computed lower chain, and (e) all the eliminated points are at the end. Thereafter, return an iterator to the first eliminated point (or the past-the-end iterator if no such point exists) as the final answer.

Eddy [16] proved that in the worst case the QUICKHULL algorithm runs in $O(nh)$ time. Due to the recursion stack maintained by the system, the amount of extra space used by this implementation is $O(h)$ words in the worst case. Hence, the worst-case performance can be poor if h is large. However, Overmars & van Leeuwen [30] proved that in the average case the algorithm runs in $O(n)$ expected time.

5.4 GIFT-WRAPPING

The GIFT-WRAPPING algorithm [23] can be visualized by wrapping paper around the input points—i.e. determining the boundary edges one by one. This means that the input is scanned h times and, for each considered point, it is necessary to perform the orientation test. Thus, the worst-case running time of this algorithm is $O(nh)$ and the constant factor in the leading term is quite high. Our preliminary experiments showed that this algorithm is slow, unless h is a small constant. Therefore, we did not consider this algorithm as a worthy competitor for TORCH and we left it outside this study.

5.5 Output sensitivity

Gomes implemented Chan's output-sensitive convex-hull algorithm [10] which is known to run in $O(n \lg h)$ time in the worst case. There are several other algorithms having the same running time [7, 11, 28, 32]. In our preliminary experiments, we were not able to make any of the known output-sensitive convex-hull algorithms competitive in practice. In this respect, our results are fully aligned with those reported by Gomes. Hence, we will not consider any of the output-sensitive algorithms in this study.

6. Pruning before sorting

The QUICKHULL algorithm brings into the surface the important issue of eliminating interior points—if there are any—before sorting. When computing the convex hull for a collection of points, one has to find the extreme points at all directions. A rough approximation of the convex hull is obtained by considering only a few predetermined directions. As discussed in several papers (see, for example, [1, 2, 15]), by eliminating the points falling inside this approximative hull, the problem size can often be reduced considerably. After this kind of preprocessing, any of the above-mentioned algorithms could be used to process the remaining points.

Akl & Toussaint [1] used four equispaced directions—those determined by the coordinate axes. Also, Andrew [2] mentioned this as an option to achieve further saving in the average-case performance. Devroye & Toussaint [15] used eight equispaced directions. Akl & Toussaint [1] demonstrated the usefulness of this idea experimentally and Devroye & Toussaint [15] verified theoretically that for certain random distributions the number of points left after pruning will be small. Unfortunately, the result depends heavily on the shape of the region, from where the points are drawn randomly. A rectangle is fine, but a circle is not.

More serious criticism of Gomes' work [21] is that the main algorithmic idea—to compute the maximal points before finding the extreme points—is not new. This idea was proposed, for example, in the paper by Bentley et al. [6] in 1978 and in a follow-up of that paper by Devroye [14] in 1980. Both of these papers took the idea one step further: Compute the maximal points before sorting as a preprocessing step. For a wide range of distributions, this approach will lead to an $O(n)$ expected-time algorithm, and the worst-case running time is still $O(n \lg n)$ [6, 14].

7. Testing

When writing the programs for this note, we tried to follow the best industrial practices. First, we made frequent reviews of each other's code. Second, we built a test suite that could be used to check the answers computed. Only after the programs passed the tests, they were benchmarked. One more time we experienced that programming is difficult. Even if the test suite had only a dozen, or so, test cases, it took time before each of the programs passed the tests. And, when we asked our students to write some more test cases, new bugs were found.

Naturally, testing was automated. Three functions formed the kernel of our test framework: `same_multiset`, `convex_polygon`, and `all_inside`. The first function `same_multiset` takes fours iterators `p`, `q`, `r`, and `s` as its arguments and checks whether the multiset of n points in the range $[\![\texttt{p}\,..\,\texttt{q})\!)$ and that in the range $[\![\texttt{r}\,..\,\texttt{s})\!)$ are the same. To carry out this task, we copied the given multisets, sorted the copies lexicographically, and verified that the sorted sequences were equal. The second function `convex_polygon` takes two iterators `p` and `q` as its arguments, and checks whether the h points in the range $[\![\texttt{p}\,..\,\texttt{q})\!)$ form the consecutive vertices of a convex polygon. The third function `all_inside` takes fours iterators `p`, `q`, `r`, and `s` as its arguments and checks whether all the n points in the range $[\![\texttt{p}\,..\,\texttt{q})\!)$ are on the boundary or interior of the convex polygon of size h specified in the range $[\![\texttt{r}\,..\,\texttt{s})\!)$. The first function is easily seen to require $O(n \lg n)$ time in the worst case. To implement the second function in $O(n)$ worst-case time, we took inspiration from [12, Exercise 33.1-5]. The third function runs in $O(n \lg h)$ worst-case time since it performs two binary searches over the convex polygon of size h for each of the n points.

When writing this note, the test suite had—among others—the following functions—the names of the test cases were selected to indicate what is being tested:

- `empty_set`,
- `one_point`,
- `two_points`,
- `three_points_on_a_vertical_line`,
- `three_points_on_a_horizontal_line`,
- `ten_equal_points`,
- `four_poles_with_duplicates`,
- `random_points_on_a_parabola`,
- `random_points_in_a_square` (10 000 test cases),
- `positive_overflow`,
- `negative_overflow`.

As one can see from this list, we expect that the programs are able to handle degenerate cases and multiset data.

Having this test suite available, it was natural to expose Gomes' program for these test cases. The first problem encountered was that the interface used by Gomes was totally different from that used by us. To be able to handle the diversity in interfaces, it was necessary to refactor our test framework. After refactoring, it is assumed that every algorithm is implemented in its own **namespace**, and that this **namespace** provides a function `solve` to be used to compute convex hulls and a function `check` to be used to check the validity of the answer computed by the solver. That is, the programs can be made self-testing if wanted—but the space overhead for doing this is quite high since extra copies of both the input and output are taken.

Let I be the iterator type used by the input sequence. For our programs the involved function signatures are the following:

```
template<typename I>
using solver = I (*)(I, I);

template<typename I>
using checker = bool (*)(I, I);
```

In this basic form, the signature of an in-place solver is similar to that used by the generic function `std::partition` in the C++ standard library. A solver takes two iterators specifying the first and the past-the-end positions of the input sequence and, after reordering the points, the return value specifies the end of the computed convex hull and the beginning of all interior points in the very same sequence. A checker can take a copy of the input, solve the problem with a solver, and use the tools in our test framework to check that we still have the same points in the output, that the computed hull is actually a convex polygon, and that all the input points are inside or on the boundary of the computed convex hull.

In Gomes' program [20], the input sequence was a private member of a class and the answer was put into a stack that was a global variable. Neither the input format nor the output format were very convenient. For example, consider a use case where we want to compute convex hulls for two specific sequences. The user cannot give the input sequence for the convex-hull class, but it must be generated by one of the routines available there. Since the output stack is shared, after computing the convex hull for the second sequence, the output for the first sequence is lost.

As the first manoeuvre, we made it possible for the user to give his or her own input and output sequences for the convex-hull class. This was done by adding a new constructor for this class that took a pointer to the input sequence and another pointer to the output sequence as its parameters. Still, after this, the caller was the owner of these sequences and took care of their memory management. The convex-hull class was only responsible for the temporary data structures needed during the computation. This way the generation of the points could be separated from the computation of the convex hull. Hereafter, we could remove all the point-generation functions from the convex-hull class. We also removed all the drawing aids since these were not relevant for us.

To compute the convex hull for a given sequence, we provided a generic function that utilized the facilities available in the convex-hull class. This code was taken from the existing main function. The signature of this generic function was as follows:

```
template<typename S, typename T>
using solver = void (*)(S&, T&);
```

That is, the input sequence is of type `S`, the output sequence is of type `T`, and both are passed by reference so that the modifications made to them will be visible for the caller. Now the output sequence can be any sequence that supports the operations `back`, `push_back`, `pop_back`, and `size`. Our test framework requires that it also provides random-access iterators.

Before running any tests, already a quick code inspection revealed that Gomes' program was not correct. (1) It must fail for $n = 0$, $n = 1$, $n = 2$ because the code refers to non-existing array entries. (2) The program had memory leaks since the allocated arrays were never released. This may hinder the repetition of many tests in a row since the memory will be released first when the `main` program is terminated. (3) It was quite clear that the orientation predicate could not work since the coordinates were floating-point numbers and the result of the computation was converted to an **int**. We fixed these obvious errors by handling the small input instances in a brute-force manner, by actually releasing the allocated memory in the destructor of the convex-hull class, and by reverting the program to rely on exact integer arithmetic, as our other programs.

After these measures, interesting bugs started to pop up. For a test case with four points, we observed the special situation that the constructed approximative hull could be much larger than n. That is, when the input was divided into four candidate collections, a point on the border between two quadrants should be copied to both quadrants. For the correctness, this may not be a problem if the following scanning will again eliminate these duplicates, but this opens up a possibility that the memory requirement for the temporary arrays is quadrupled. The problem was that the duplicates were not processed correctly by the following scan. This special case forced us to look at the code more carefully.

Andrew [2] was very careful to mention how to select the poles if there are several candidates with the same x-coordinate. Gomes had not implemented this precaution: he sorted the points according to their x-coordinates, and after sorting made the first point and the last point the west and east poles, respectively. Therefore, the program failed, for example, for the following input: (0, 0), (0, 2), and (0, 1). The output was (0, 0), (0, 1), and (0, 0). The program was supposed to compute the convex hull in counterclockwise order. The (wrong) west pole was there twice because the scan following the maxima finding did not use the west pole to eliminate the points at the end of the upper hull.

To get forward, we revised (1) the function that found the poles and (2) the function that inflated the approximate hull. After this, we found two more errors. (3) The scanning routine added the three first points uncritically to the output, but among these points there could be duplicates that should be removed. (4) The approximate hull could have many points with the same x-coordinate as the (topmost) west pole. As pointed out by Andrew [2], these points must be processed differently from the remaining points. So we needed a separate loop to find the bottommost point with the same x-coordinate as the west pole; all the other points between the topmost west pole and the bottommost west pole (if any) could be ignored.

Now the program passed the test case for three vertical points, but failed for three horizontal points: (0, 0), (1, 0), (2, 0). For this set, the approximate hull contained five points: (0, 0), (1, 0), (2, 0), (1, 0), (0, 0). Hence, the following scan visited the points (0, 0), (0, 1), removed (0, 1) from the stack, added (0, 2) to the stack, and then incorrectly removed (0, 2) from the stack—when the scan was back from (0, 2) to (0, 0) along the same route. To fix this, we ignore the current point on the approximative hull if it is strictly on the line segment between the west pole and the east pole. Otherwise, the normal elimination procedure for the points in the stack is executed. Unfortunately, this check increased the number of orientation tests executed which was the figure Gomes tried to optimize.

Next the program failed for random points on a parabola. For instance, for four points (3, 9), (1, 1), (2, 4), (0, 0), the approximative hull contained seven points: (0, 0), (1, 1), (2, 4), (3, 9), (2, 4), (1, 1), (0, 0); and the (incorrect) output contained two points: (0, 0) and (1, 1). That is, the approximate hull started from the origin, went up below the parabola to the top, and became down the same way above the parabola. How are we supposed to eliminate these points that we have seen earlier? Graham's scan [22] is known to work for a star-shaped polygon, but special care must be taken if there are duplicates or if there are several points with the same polar angle when the vertices are sorted around the centre of the polygon. It also works for a monotonic polygonal chain [2], but—as the examples given here show—it does not work for a polygonal chain where the vertices overlap.

At this point we gave up and decided to implement our own version of TORCH. After finding the poles, the algorithm processes the points in four phases (for an illustration, see Figure 4):

(1a) Partition the input sequence such that the points in the north-west quadrant (Q_1 in the figure) determined by the west and north poles are moved to the beginning and the remaining points to the end (normal two-way partitioning).

(1b) Sort the points in Q_1 according to their x-coordinates.

(1c) Find the maximal points in Q_1 and move the eliminated points at the end of the zone occupied by the points of Q_1.

(1d) Scan the remaining candidates to determine the convex chain from the west pole to the north pole. Again the eliminated points are put at the end of the corresponding zone.

(2a) Consider all the remaining points (also those eliminated earlier) and partition the zone occupied by them into two so that the points in Q_2 become just after the west-north chain ending with the north pole.

(2b) Compute the north-east chain of the convex hull for the points in Q_2 as in Steps (1b)–(1d).

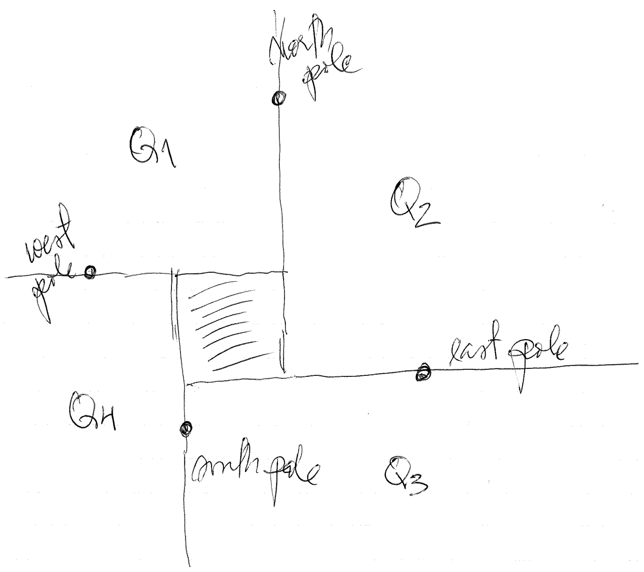

Figure 4. In-place TORCH algorithm illustrated.

(3) Compute the south-east chain of the convex hull for the points in Q_3 as in Step (2).
(4) Compute the south-west chain of the convex hull for the points in Q_4 as in Step (2).

It is important to observe that any point in the middle of Figure 4 is dominated by one of the poles in each of the four directions. Therefore, these points are never moved to a candidate collection. However, if a point lies in two (or more) quadrants, it will take part in several sorting computations. (In Figure 4, move the east pole up and see what happens.) To recover from this, it would be advantageous to use an adaptive sorting algorithm. (Actually, INTROSORT [29] should be able to handle this situation satisfactorily.)

This version was much easier to debug since we could rely on the standard-library components and our own tested components. It is no secret that, for each of our programs, serious debugging—as experienced with Gomes' program—has been necessary. However, it was easier to talk about someone else's mistakes than our own mistakes. It took about four months for the second author to develop an implementation of the PLANE-SWEEP algorithm that included all Andrew's enhancements and that passed all the test cases. The first author worked nine months for his master's thesis entitled "Expected linear-time convex-hull algorithms" [17] and the version of QUICKHULL published there was faulty. First the revised version developed for this note passed all our test cases. Seeing the development history of the QHULL program (`http://www.qhull.org/`), we do not dare to claim that our programs are correct, but we promise a reward of 2.56 brownie points for every person that is the first to report an error in one of our convex-hull programs (IN-SITU PLANE-SWEEP, QUICKHULL, or IN-SITU TORCH). You should send your bug reports to jyrki@di.ku.dk with the subject "Bug report".

8. Alternative experiments

Now we are ready to describe our experiments to obtain an alternative truth on the state of the art as to the algorithms for computing the convex hulls in the plane. We selected the following three competitors for this study:

IN-SITU TORCH**.** As explained earlier, Gomes' original implementation [20] used far too much memory (Section 3) and did not work correctly (Section 7), so we had to reimplement it before we could use TORCH in our benchmarking.

IN-SITU PLANE-SWEEP**.** We implemented this algorithm following the guidelines given in Section 5.2.

QUICKHULL**.** We implemented this algorithm as explained in Section 5.3.

To simplify the implementation task, we used the standard-library routines whenever possible, e.g. `std::sort`, `std::minmax_element`, `std::max_element`, `std::partitioning`, `std::reverse`.

As a point of reference, we also made measurements for the standard-library sorting routine:

x-SORT. Call `std::sort` to sort the points according to their x-coordinates.
Lastly, we considered the pruning heuristics proposed in [1, 2] and combined it with our PLANE-SWEEP implementation:

POLES-FIRST. Compute the west, north, east, and south poles; and prune the points inside the quadrilateral having these poles as its vertices.

This pruning heuristics can be implemented using $O(1)$ extra space and $O(n)$ time. In principle, the elimination scan is just a two-way partitioning.

In all our experiments, the coordinates of the points were integers of type **int**, each occupying 32 bits. The intermediate calculations required by the geometric predicates must use wider numbers: 64-bit integers would be sufficient, but the resulting code is inelegant. Even a fewer number of bits is known to be sufficient [3]. As shown in Section 4, with 66-bit integers, robustness can be achieved in a simple way, but this requires numbers that are wider than any of the built-in integers. Hence, the goal to reduce the number of times the geometric primitives are to be executed is well motivated.

All the programs were modified to use the same point class template that took the type of the coordinates as its template parameter. To ensure robustness, we used double-precision integers (**signed long long** and **unsigned long long**), kept track of all the possible overflows and underflows, and dealt with them accordingly. This alternative required smaller implementation effort, and it was better tested than the experimental library solution described in Section 4. This solution relies on the fact that the coordinates are of type **int**, so it is not generic. However, since we are not dependent on any multi-precision integer library, it should be easier for others to redo our experiments.

All the experiments were run on two computers: one running Linux and the other running Windows. Both test computers could be characterized as standard laptops that one can buy from any well-stocked computer store. A summary of the test environments is given in Figure 5. Although both computers had four cores, the experiments were run on a single core.

As the input for the programs, we used three types of data collections:

random points in a square. The coordinates of the points were integers drawn randomly from the range $[\![-2^{31} \,..\, 2^{31}\,)\!)$ (i.e. random **int**s). For this collection, the expected size of the output is $O(\lg n)$ [6].

random points in a disc. As above, the coordinates of the points were random **int**s, but only the points inside the circle centred at the origin $(0, 0)$ with the radius $2^{31} - 1$ were accepted to the input collection. In this case, the expected size of the output is $O(\sqrt{n})$.

random points in a saturated universe. One important issue raised by Gomes [21] was to consider what happens when there are many duplicates among the input. To simulate this situation, the coordinates were integers drawn randomly from a small universe $[\![-\lfloor\sqrt{n}\rfloor \,..\, \lfloor\sqrt{n}\rfloor]\!]$.

None of the considered algorithms use random-access capabilities to a large extend. Therefore, we did not expect to see any significant cache effects in these experiments. Based on the memory configuration of the test computers, we just wanted to check what happened when the problem instances

(a)

processor. Intel® Core™ i7-6600U CPU @ 2.6 GHz (turbo-boost up to 3.6 GHz)—4 cores
word size. 64 bits
first-level data cache. 8-way set-associative, 64 sets, 4×32 KB
first-level instruction cache. 8-way set-associative, 64 sets, 4×32 KB
second-level cache. 4-way set-associative, 1 024 sets, 256 KB
third-level cache. 16-way set-associative, 4 096 sets, 4.096 MB
main memory. 8.038 GB
operating system. Ubuntu 17.10
kernel. Linux 4.13.0-16-generic
compiler. `g++` 7.2.0
compiler options. `-O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts`

(b)

processor. Intel® Core™ i7-4700MQ CPU @ 2.40 GHz—4 cores and 8 logic processors
word size. 64 bits
first-level data cache. 8-way set-associative, 4×32 KB
first-level instruction cache. 8-way set-associative, 4×32 KB
second-level cache. 4-way set-associative, 4×256 KB
third-level cache. 12-way set-associative, 6 MB
main memory. DDR3, 8 GB, DRAM frequency 800 MHz
operating system. Microsoft Windows 10 Home 10.0.15063
compiler. Microsoft Visual Studio Express 2013
compiler options. build to release mode, run as `.exe`

Figure 5. Hardware and software in use: (a) Linux computer, (b) Windows computer.

became bigger than the capacity of different memory levels. Therefore, we measured the running time of the programs for five predetermined values of n: 2^{10}, 2^{15}, 2^{20}, 2^{25}, and 2^{30}. In particular, it seemed not to be necessary to plot more detailed curves of the running times.

In the earlier studies, the number of orientation tests and that of coordinate comparisons have been targets for optimization. Since the considered algorithms reorder the points in-place, the number of coordinate moves is also a relevant performance indicator. Hence, in addition to the execution time, we measured the number of orientation tests, coordinate comparisons, and coordinate moves. The number of orientation tests performed is proportional to n in the worst case for TORCH and PLANE-SWEEP, and in the average case for QUICKHULL. The number of coordinate comparisons and coordinate moves is proportional to $n \lg n$ in the worst and average cases for TORCH and PLANE-SWEEP, and to n in the average case for QUICKHULL.

Be aware that all the reported numbers are scaled: For any of the performance indicators, if X is the observed measurement result, we report X/n. That is, a constant means linear performance. To avoid the problems with inadequate clock precision, a test for $n = 2^i$ was repeated $\lceil 2^{27}/2^i \rceil$ times; each repetition was done with a new input sequence. Sadly the Windows computer could only handle up to 2^{26} points per test case.

In our first set of experiments, we measured the CPU time used by the competitors for the considered data sets. The obtained results are shown in Figure 6 (square), Figure 7 (disc), and Figure 8 (saturated universe). At first glance, the results are unambiguous: for randomly generated data, QUICKHULL is quick! We could force QUICKHULL to perform worse than the two competitors when the points were drawn from an arc of a parabola. However, we could only make it a constant factor slower; we could not provoke the $O(nh)$ behaviour because of the restricted precision of the coordinates.

Let us look at the results a little more closely. The first thing we notice is that in most cases the cost per point increases as the input instances get larger, while sometimes the cost may stagnate or even get smaller. A logarithmic increase indicates a dominating $O(n \lg n)$ cost, as the CPU times are divided by n. Sorting naturally follows this growth. As both TORCH and PLANE-SWEEP sort a fraction of points a few times, the cost of both is in direct relation to the sorting cost. Both QUICKHULL and POLES-FIRST are not directly bounded by this, since they perform some elimination before reverting to sorting. Especially, we note that QUICKHULL tends to become slightly faster (per point) on larger instances. There is a trade-off between the amount of work used for pruning and that used for sorting the remaining points. For randomly generated data, h tends to grow much slower than n, so the cost incurred by sorting will gradually vanish.

(a)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	36.65	50.27	55.27	34.20	50.28
2^{15}	54.03	65.12	71.34	30.73	57.25
2^{20}	71.48	82.70	88.60	30.81	66.44
2^{25}	89.37	102.3	108.1	30.68	75.12
2^{30}	162.7	186.2	183.2	91.56	219.0

(b)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	41.18	53.42	62.08	51.49	60.80
2^{15}	61.30	72.29	82.96	47.98	69.95
2^{20}	83.49	93.59	104.9	48.52	80.85
2^{25}	106.6	122.7	130.7	48.68	97.92

Figure 6. The running time of the competitors [in nanoseconds per point] for the square data set: (a) Linux computer, (b) Windows computer.

(a)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	36.23	54.84	53.82	34.10	41.37
2^{15}	53.70	69.99	69.02	30.77	44.80
2^{20}	70.96	87.27	85.92	31.89	50.87
2^{25}	89.05	105.4	104.4	31.89	57.90
2^{30}	155.0	198.9	183.8	91.36	135.6

(b)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	39.18	52.76	61.47	28.31	54.72
2^{15}	60.70	72.79	80.62	25.81	65.70
2^{20}	66.15	97.78	91.59	22.41	105.5
2^{25}	65.28	89.83	86.75	23.99	108.8

Figure 7. The running time of the competitors [in nanoseconds per point] for the disc data set: (a) Linux computer, (b) Windows computer.

(a)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	29.35	53.52	47.98	16.75	60.57
2^{15}	37.89	64.02	53.74	14.18	69.28
2^{20}	47.18	82.64	62.05	14.50	77.73
2^{25}	57.92	106.6	72.73	15.42	88.20
2^{30}	115.5	422.5	143.0	73.47	223.1

(b)

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	29.42	56.58	53.76	25.68	70.98
2^{15}	36.52	57.43	59.02	22.46	78.93
2^{20}	47.83	68.50	68.60	22.62	89.58
2^{25}	59.97	83.33	80.63	23.13	102.9

Figure 8. The running time of the competitors [in nanoseconds per point] for the saturated-universe data set: (a) Linux computer, (b) Windows computer.

In terms of speed, TORCH has three success criteria: (1) It should be faster than PLANE-SWEEP that it attempts to improve upon; (2) it should perform almost as well as sorting; and (3) it should perform well when compared to other algorithms of industrial standard, as it would otherwise be outcompeted by those. For problem instances of low and medium sizes on a square (Figure 6) and on a disc (Figure 7), TORCH is placed well in between x-SORT and PLANE-SWEEP. It performs poorly in the Linux environment for the saturated universe (Figure 8), likely because of the earlier-mentioned problem with overlapping work.

We notice that TORCH becomes slower for large instances. This may be due to the fact that it tries to reduce the computational cost from $O(n)$ to $O(h)$. To achieve this, it may be necessary to process the same points several times. Thus, for large data collections, its advantage over PLANE-SWEEP decreases. There is a trade-off between the pruning cost and the obtained gain. The pruning cost grows with an extra factor of $c \lg n$ per point, albeit with a small constant c, while the saving is a constant per point. As for the third criteria, on the Linux computer, TORCH struggles hard with POLES-FIRST, which is often faster, since it prunes points before sorting, causing the sorting term to grow slower. On the Windows computer, TORCH often beats POLES-FIRST, but it still has trouble competing with QUICKHULL.

One should also notice that the programs behave differently on different computers. On the Linux machine, for all except large instances, both TORCH and POLES-FIRST perform better than PLANE-SWEEP, but not significantly better. On the Windows machine, TORCH lands between the two; with some exceptions, sometimes PLANE-SWEEP and sometimes POLES-FIRST is faster. This indicates that the orientation tests can be evaluated faster on the Linux machine, which coincide with indications from our micro-benchmarks. The differences can come from several sources, but we consider it likely that different types of CPUs, compilers, operating systems, and hardware architectures have contributed to this. These differences serve as a warning to those who wish to crown a specific algorithm as the champion, since even with reasonable statistical accuracy, there can be a significant difference in how well a given algorithm performs in different environments.

In our second set of experiments, we considered the behaviour of the competitors for the other performance indicators. The average-case scenario was considered, i.e. the coordinates were random **int**s (square). These experiments were carried out by using a coordinate type that increased a counter each time a comparison, a move, or an orientation test was executed. Hence, no changes to the programs computing the convex hulls were necessary.

The obtained counts are shown in Figure 9 (orientation tests), Figure 10 (coordinate comparisons), and Figure 11 (coordinate moves). From these numbers it is clear that, for random data, QUICKHULL runs in linear expected time. As to the orientation tests, TORCH is a clear winner; the expected number of orientation tests executed is $O(\lg n)$ which explains the zeros. As to the number of coordinate comparisons, TORCH comes last because interior points take part in four partitioning processes and some of them may also take part in more than one sorting process since the quadrants can overlap. As to the number of coordinate moves, despite repeated work in partitioning, TORCH performs better than PLANE-SWEEP since fewer moves are done in partitioning and in scanning, and the sorting tasks are smaller.

9. Concluding remarks

Let us summarize the main issues raised in this note:

n	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	0.02	2.31	4.95	2.85
2^{15}	0.00	2.34	4.86	2.87
2^{20}	0.00	2.34	4.82	2.91
2^{25}	0.00	2.39	4.70	2.80
2^{30}	0.00	2.09	5.37	2.80

Figure 9. The number of orientation tests executed by the competitors [divided by n] for the square data set.

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	12.0	21.7	14.8	2.37	14.0
2^{15}	18.1	27.9	20.9	2.26	17.2
2^{20}	24.2	34.2	27.1	2.25	20.4
2^{25}	30.2	40.6	33.2	2.25	23.0
2^{30}	36.5	45.8	39.1	2.25	25.9

Figure 10. The number of coordinate comparisons performed by the competitors [divided by n] for the square data set.

n	x-SORT	TORCH	PLANE-SWEEP	QUICKHULL	POLES-FIRST
2^{10}	19.1	21.3	31.4	15.2	23.5
2^{15}	26.2	28.0	38.6	14.7	27.0
2^{20}	33.3	35.3	45.7	14.5	30.9
2^{25}	40.5	42.3	53.0	14.5	33.6
2^{30}	47.4	49.2	59.7	16.4	38.0

Figure 11. The number of coordinate moves performed by the competitors [divided by n] for the square data set.

literature. It is disappointing to know that the key idea of TORCH appeared in open literature in 1978 [6, 14]. A normal Internet user does not have access to these sources; one has to work at a well-established university in order to access them in paper or digital form. Still, we can just agree on the advice given in [34, Chapter 3]: "you need to be confident that you have read and understood all of the scientific literature that has a significant connection to your work."

correctness. It gives little meaning to compare programs that do not work correctly. For random data, Gomes' implementation of TORCH produced reasonable results, but it could not handle special cases correctly. Again, we can agree on the advice given in [34, Chapter 14]: "Find an independent way of verifying that the output is correct".

reproducibility. In experimental work, a "key principle is that the experiment should be verifiable and reproducible" (the quotation is from [34, Chapter 14]). Unfortunately, since Gomes' code for the ROTATIONAL-SWEEP and PLANE-SWEEP algorithms was not available for us, and since sufficient information was not provided how he used the QHULL library routine, we could not redo his experiments.

space requirements. There is a big difference if the amount of extra space used is $O(n)$ or $O(1)$ words. We explained how the PLANE-SWEEP and TORCH algorithms can be implemented in-place. The in-situ versions of these algorithms turned out to work well in practice as well.

speed. It can be advantageous to use less memory. One can avoid the overhead caused by memory management and one can incur less cache misses. Our experiments indicate that, for in-situ variants, the improvement provided by TORCH compared to PLANE-SWEEP is marginal.

fairness. Consider the warning [34, Chapter 14]: "A common error is to compare the resource requirements of two algorithms that perform subtly different tasks." We would take this one step further: It is not fair to compare the resource requirements of two algorithms if they perform the task in different means. More specifically, it is not fair to compare two convex-hull algorithms if the first is robust (QHULL) and the other is not (Gomes' implementation of TORCH). Additionally, it is not fair to ignore the costs involved in memory allocation and deallocation if a program uses temporary storage to accomplish its task (as Gomes did).

more aggressive elimination. The idea of eliminating interior points during the computation is good! However, in contemporary computers, sorting is fast. So an elimination heuristics—even if it runs in linear time—should not be complicated. Two simple ideas may make the programs faster for some special cases: (1) When some points are observed to be on the line segment between two extreme points, they could be eliminated. For example, in our PLANE-SWEEP implementation, such points are kept among the upper-hull candidates, in order to perform a two-way partitioning instead of a three-way partitioning (which is known to be slower). (2) When sorting the candidate collections according to their x-coordinates, if two points are seen to have the same x-coordinate, the point with the smaller (or larger) y-coordinate could be eliminated. More generally, it is possible to sort n points having at most d different x-values in $O(n \lg d)$ worst-case time. On the other hand, as our experiments also indicated, aggressive elimination may make a program (POLES-FIRST) slower since for some input instances not enough points will be eliminated to actually speed-up the program.

Acknowledgements

We thank the students of our algorithm-engineering course (autumn 2017) for the test cases that made our programs to fail, and for their comments that helped us to better understand the material discussed. We also thank Pawel Winter for bringing the authors together.

Software availability

The programs implemented and benchmarked are available via the home page of the CPH STL (`http://www.cphstl.dk`) in the form of a `pdf` file [18] and a `zip` archive.

References

[1] S. G. Akl and G. T. Toussaint, A fast convex hull algorithm, *Inform. Process. Lett.* **7**, 5 (1978), 219–222.

[2] A. M. Andrew, Another efficient algorithm for convex hulls in two dimensions, *Inform. Process. Lett.* **9**, 5 (1979), 216–219.

[3] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec, Evaluating signs of determinants using single-precision arithmetic, *Algorithmica* **17**, 2 (1997), 111–132.

[4] C. Barber, D. Dobkin, and H. Huhdanpaa, The Quickhull algorithm for convex hulls, *ACM Trans. Math. Software* **22**, 4 (1996), 469–483.

[5] C. B. Barber, Qhull manual, Worldwide web document (1995–2015). Retrieved from `http://www.qhull.org/`.

[6] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson, On the average number of maxima in a set of vectors and applications, *J. ACM* **25**, 4 (1978), 536–543.

[7] B. K. Bhattacharya and S. Sen, On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm, *J. Algorithms* **25**, 1 (1997), 177–193.

[8] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint, Space-efficient planar convex hull algorithms, *Theoret. Comput. Sci.* **321**, 1 (2004), 25–40.

[9] A. Bykat, Convex hull of a finite set of points in two dimensions, *Inform. Process. Lett.* **7**, 6 (1978), 296–298.

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

[11] T. M. Y. Chan, J. Snoeyink, and C.-K. Yap, Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three, *SODA 1995*, SIAM (1995), 282–291.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, The MIT Press (2009).

[13] Dept. Comput. Sci., Univ. Copenhagen, The CPH STL, Website `http://www.cphstl.dk/` (2000–2018).

[14] L. Devroye, A note on finding convex hulls via maximal vectors, *Inform. Process. Lett.* **11**, 1 (1980), 53–56.

[15] L. Devroye and G. T. Toussaint, A note on linear expected time algorithms for finding convex hulls, *Computing* **26**, 4 (1981), 361–366.

[16] W. F. Eddy, A new convex hull algorithm for planar sets, *ACM Trans. Math. Software* **3**, 4 (1977), 398–403.

[17] A. N. Gamby, Expected linear-time convex-hull algorithms: Investigation into possible improvements of the running times in the planar case, M. Sc. thesis, Dept. Comput. Sci., Univ. Copenhagen (2017).

[18] A. N. Gamby and J. Katajainen, Computing convex hulls in the plane: Collected algorithms in C++, CPH STL report **2018-1**, Dept. Comput. Sci., Univ. Copenhagen (2018).

[19] GeeksforGeeks, Convex hull | Set 2 (Graham scan), Worldwide web document (2017). Retrieved from `http://www.cdn.geeksforgeeks.org/convex-hull-set-2-graham-scan/`.

[20] A. J. P. Gomes, Git repository, Worldwide web document (2016). Retrieved from `https://github.com/mosqueteer/TORCH/`.

[21] A. J. P. Gomes, A total order heuristic-based convex hull algorithm for points in the plane, *Comput.-Aided Design* **70** (2016), 153–160.

[22] R. L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Inform. Process. Lett.* **1**, 4 (1972), 132–133.

[23] R. A. Jarvis, On the identification of the convex hull of a finite set of points in the plane, *Inform. Process. Lett.* **2**, 1 (1973), 18–21.

[24] J. Katajainen, Worst-case-efficient dynamic arrays in practice, *SEA 2016*, *LNCS* **9685**, Springer (2016), 167–183.

[25] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Dept. Comput. Sci., Univ. Copenhagen (2017).

[26] J. Katajainen and T. Pasanen, Stable minimum space partitioning in linear time, *BIT* **32**, 4 (1992), 580–585.

[27] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap, Classroom examples of robustness problems in geometric computations, *Comput. Geom.* **40**, 1 (2008), 61–78.

[28] D. G. Kirkpatrick and R. Seidel, The ultimate planar convex hull algorithm?, *SIAM J. Comput.* **15**, 1 (1986), 287–299.

[29] D. R. Musser, Introspective sorting and selection algorithms, *Software Pract. Exper.* **27**, 8 (1997), 983–993.

[30] M. H. Overmars and J. van Leeuwen, Further comments on Bykat's convex hull algorithm, *Inform. Process. Lett.* **10**, 4–5 (1980), 209–212.

[31] S. Schirra, Robustness and precision issues in geometric computation, *Handbook of Computational Geometry*, Elsevier (2000), 597–632.

[32] R. Wenger, Randomized quickhull, *Algorithmica* **17**, 3 (1997), 322–329.

[33] J. W. J. Williams, Algorithm 232: Heapsort, *Commun. ACM* **7**, 6 (1964), 347–348.

[34] J. Zobel, *Writing for Computer Science*, 3rd edition, Springer-Verlag London (2014).