A Faster Convex-Hull Algorithm via Bucketing

Ask Neve Gamby[1][0000−0002−2839−166X] and
Jyrki Katajainen[2,3][0000−0002−7714−5588]

[1] National Space Institute, Technical University of Denmark, Centrifugevej, 2800 Kongens Lyngby, Denmark aknvg@space.dtu.dk
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark jyrki@di.ku.dk
http://hjemmesider.diku.dk/~jyrki/
[3] Jyrki Katajainen and Company, 3390 Hundested, Denmark

Abstract. In the convex-hull problem, in two-dimensional space, the task is to find, for a given sequence S of n points, the smallest convex polygon for which each point of S is either in its interior or on its boundary. In this paper, we propose a variant of the classical bucketing algorithm that (1) solves the convex-hull problem for any multiset of points, (2) uses $O(\sqrt{n})$ words of extra space, (3) runs in $O(n)$ expected time on points drawn independently and uniformly from a rectangle, and (4) requires $O(n \lg n)$ time in the worst case. Also, we perform experiments to compare BUCKETING to other alternatives that are known to work in linear expected time. In our tests, in the integer-coordinate setting, BUCKETING was a clear winner compared to the considered competitors (PLANE-SWEEP, DIVIDE & CONQUER, QUICKHULL, and THROW-AWAY).

Keywords: Computational geometry · Convex hull · Algorithm · Bucketing · Linear expected time · Experimental evaluation

1 Introduction

Bucketing is a practical method to improve the efficiency of algorithms. As an example, consider the sorting problem under the assumption that the elements being sorted are integers. In BUCKETSORT [1], the elements are sorted as follows: (1) Find the minimum and maximum of the elements. (2) Divide the closed interval between the two extrema into equal-sized subintervals (*buckets*). (3) Distribute the points into these buckets. (4) Sort the elements in each bucket. (5) Concatenate the sorted buckets to form the final output. A data structure is needed to keep track of the elements inside the buckets, so this is not an in-place sorting method. The key is to use a worst-case optimal sorting algorithm when processing the buckets. This way the worst-case performance remains unchanged since the bucketing overhead is linear.

In our exploratory experiments, we could confirm that for integer sorting BUCKETSORT, which used the C++ standard-library `std::sort` to sort the buckets, was often faster than `std::sort` itself. In our implementation we followed the guidelines given by Nevalainen and Raita [27]. Their advice can be summed up as follows:

1. Do not make the distribution table too large! According to the theory, the number of buckets should be proportional to the number of elements. Often this is too large and may lead to bad cache behaviour. According to our experience, the memory footprint of the distribution table should not be much larger than the size of the second-level cache in the underlying computer.
2. Do not use any extra space for the buckets when distributing the elements into the buckets! Permute the elements inside the input sequence instead.

In geometric applications, in two-dimensional space, bucketing can be used in a similar manner: (1) Find the smallest rectangle covering the input points. (2) Divide this rectangle into equal-sized rectangles (*cells*). (3) Solve the problem by moving from cell to cell while performing some local computations. The details depend on the application in question.

In the 1980s, bucketing was a popular technique to speed up geometric algorithms (for two surveys, see [5,13]). In the 1990s, the technique was declared dead, because of the change in computer architectures: caching effects started to dominate the computational costs. Today, bucketing can again be used, provided that the distribution table is kept small. In this work, we only study bucketing algorithms that use a small distribution table and that avoid explicit linking by permuting the elements inside the sequence.

More specifically, we consider the problem of finding the convex hull for a sequence S of n points in the plane. Each point p is specified by its Cartesian coordinates (p_x, p_y). The *convex hull* $\mathcal{H}(S)$ of S is the boundary of the smallest convex set enclosing all the points of S. The goal is to find the smallest description of the convex hull, i.e. the output is the boundary of a convex polygon, the vertices of which—so-called *extreme points*—are from S.

Throughout the paper, we use n to denote the size of the input, h the size of the output, and $\lg x$ as a shorthand for $\log_2(\max\{2, x\})$. Our specific goal is to develop a convex-hull algorithm that (1) requires $O(n \lg n)$ time in the worst case, (2) runs in $O(n)$ time in the average case, (3) uses as little extra space as possible, and (4) is efficient in practice. In the integer domain, good average-case performance may be due to various reasons [7]:

1. The points are chosen uniformly and independently at random from a bounded domain.
2. The distribution of the points is *sparse-hulled* [11, Exercise 33-5], meaning that in a sample of n points the expected number of extreme points is $O(n^{1-\varepsilon})$ for some constant $\varepsilon > 0$.

In the book by Devroye [14, Section 4.4], a bucketing algorithm for finding convex hulls was described and analysed. It could improve the efficiency of any worst-case-efficient algorithm such that it runs in linear expected time without sacrificing the worst-case behaviour. According to Devroye, this algorithm is due to Shamos. This algorithm needs a distribution table of size $\Theta(n)$ which may lead to bad cache behaviour. We describe a variant of this algorithm that reduces the consumption of extra memory to $O(\sqrt{n})$, which was low enough to not contribute significantly to cache misses in our computing environment.

Many convex-hull algorithms are known to run in linear expected time. Therefore, we also compared the practical performance of the new bucketing algorithm to that of other algorithms. The competitors considered were PLANE-SWEEP [4] (using BUCKETSORT as proposed in [3]; the average-case analysis of BUCKETSORT can be found, e.g., in [14, Chapter 1]); DIVIDE & CONQUER [29] (for the analysis, see [7]); QUICKHULL [10,16,20] (for the analysis, see [28]); and THROW-AWAY [2,4,12] (for the analysis, see [12]).

The contributions of this paper can be summarized as follows:

- We describe a space-efficient bucketing algorithm that solves the convex-hull problem for any multiset of points in the plane (Section 2).
- We perform micro-benchmarks to show which operations are expensive and which are not (Section 3). The results of these micro-benchmarks give an indication of how different algorithms should be implemented.
- We provide a few enhancements to most algorithms to speed up their straightforward implementations (Section 4). It turns out that, with careful programming, most known algorithms can be made fast.
- We perform experiments, in the integer-coordinate setting, to find out what is the state of the art when computing the convex hulls in the plane (Section 5).
- We report the lessons learned while doing this study (Section 6). Many of the guidelines are common-sense rules that can be found from the texts discussing experimentation (see, e.g. [8, Chapter 8]).

2 Bucketing

In one-dimensional bucketing, when the values come from the interval $[\mathtt{min}, \mathtt{max}]$ and the distribution table has $\mathtt{m}$ entries, the bucket index i, $0 \leq i \leq \mathtt{m} - 1$, of value v is computed using the formula

$$i = \left\lfloor \frac{(v - \mathtt{min}) * (\mathtt{m} - 1)}{\mathtt{max} - \mathtt{min}} \right\rfloor .$$

In two-dimensional bucketing, such a formula is needed for both x- and y-coordinates. Normally, geometric primitives only use addition, subtraction, and multiplication, but here we also need whole-number division.

Old version. The BUCKETING algorithm described in [14, Section 4.4] for solving the convex-hull problem works as follows: (1) Determine a bounding rectangle of the n input points. (2) Divide this rectangle into rectangular cells using a grid of size $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ and distribute the points into these cells. (3) Mark all outer-layer cells that may contain extreme points. (4) Collect all points in the marked cells as the extreme-point candidates. (5) Finally, use any known algorithm to compute the convex hull of the candidates.

In a naive implementation, the data at each cell are stored in a linked list and a two-dimensional array is used to store the headers to these lists. In the book [14, Section 4.4], the computation of the outer-layer cells was described as follows. (1) Find the leftmost non-empty column of cells and mark all the occupied cells in this column. Recall the row index j of the northernmost occupied cell. (2) After processing column i, mark one or more cells in column $i+1$ as follows: (a) Mark the cell at row number j. (b) Mark all cells between this cell and the northernmost occupied cell on that column provided that its row number is at least $j+1$. (c) Update j if we moved upwards. This way we get a staircase of at most $2\lceil\sqrt{n}\,\rceil$ marked cells. As to the correctness, all extreme points in the north-west quadrant must be in a marked cell. The other three quadrants are processed in a similar manner. Eventually at most $8\lceil\sqrt{n}\rceil$ cells are marked.

Devroye proved [14] that this algorithm runs in linear expected time when the points are independently and uniformly distributed in a rectangle. The worst-case running time depends on the algorithm used in the last step; with the PLANE-SWEEP algorithm this is $O(n \lg n)$ [4].

New version. We call the columns *slabs*. There are $\lceil\sqrt{n}\,\rceil$ slabs, each storing two y-values; we call them `min` and `max`, but their meaning depends on in which step of the algorithm we are. We only use bucketing for the x-coordinates when determining in which slab a point is; we do not materialize the cells. In the $\pm y$ directions, we maintain a staircase of y-values instead of a staircase of cells.

In detail, the algorithm works as follows:

(1) Find the minimum and maximum x-coordinate values of the points. These values are needed in the formula determining the slab of a point.
(2) If all points are on a vertical line, solve this special case by finding the points with the extreme y-coordinates, move these two points (or one point) to the beginning of the input sequence, and report them as the answer.
(3) Allocate space for the slab structure and initialize it so that in each slab the `max` value is $-\infty$ and the `min` value $+\infty$.
(4) Determine the extreme y-coordinates in each slab by scanning the points, calculating their slab index, and updating the stored `min` and `max` values within the slabs whenever necessary (random access needed).
(5) Determine the indices of the slabs where the topmost and bottommost points lie. This information can be extracted by examining the slab structure. After this the quadrants are uniquely determined. For example, when processing the west-north quadrant, the slabs are visited from the leftmost slab to the one that contains the topmost point and the `max` values are reset.
(6) In the west-north quadrant, form a staircase of y-values that specifies in each slab where the extreme points can lie. Initially, the `roof`, keeping track of the highest y-value seen so far, is set to the `max` value of the first slab and the `max` value at that slab is reset to $-\infty$. When visiting the following slabs, both the `max` value and the `roof` are updated such that the `roof` becomes the maximum of itself and `max` of the current slab, while `max` takes the value of the `roof` at the previous slab.
(7) The other quadrants are treated in the same way to form the staircases there.
(8) Partition the input by moving all the points outside the region determined by the staircases (above or below) to the beginning of the input sequence, and the points that were inside and could be eliminated to the end of the sequence. For each point, its slab index must be computed and the slab structure must be consulted (which involves random access).

Table 1. Hardware and software in use.

Processor.	Intel® Core™ i7-6600U CPU @ 2.6 GHz (turbo-boost up to 3.6 GHz) × 4
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 18.04.1 LTS
Kernel.	Linux 4.15.0-43-generic
Compiler.	`g++` 8.2.0
Compiler options.	`-O3 -std=c++2a -Wall -Wextra -fconcepts -DNDEBUG`

(9) Release the space allocated for the slab structure.
(10) Apply any space-efficient convex-hull algorithm for the remaining points and report the convex hull first in the sequence.

Since the region that is outside the staircases computed by the new algorithm is smaller than the region covered by the marked cells in the original algorithm, the runtime analysis derived for the old version also applies for the new version. The critical region covers at most $8\lceil\sqrt{n}\,\rceil$ cells (that are never materialized) and the expected value for the maximum number of points in a cell is $O(\lg n/\lg\lg n)$. Hence, the work done in the last step is asymptotically insignificant. The amount of space required by the slab structure is $O(\sqrt{n})$. All the other computations can be carried out using a workspace of constant size (*in place*) or logarithmic size (*in situ*); for a space-efficient variant of PLANE-SWEEP, see [17].

3 Micro-benchmarking

When tuning our implementations, we based our design decisions on micro-benchmarks. These benchmarks should be understood as sanity checks. We encourage the reader to redo some of the tests to see whether the same conditions are valid in his or her computer system.

Test environment. All the experiments were run on a Linux computer. The programs were written in C++ and the code was compiled using the `g++` compiler. The hardware and software specifications of the test computer are summarized in Table 1. In the micro-benchmarks the same data set was used:

Square data set. The coordinates of the points were integers drawn randomly from the range $[\![-2^{31} \mathinner{..} 2^{31})\!)$ (i.e. random **int**s).

The points were stored in a C array. We report the test results for five values of n: 2^{10}, 2^{15}, 2^{20}, 2^{25}, and 2^{30}. All the reported numbers are scaled: For every performance indicator, 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 array.

Orientation tests. As an example of a geometric computation, where correctness is important, let us consider Graham's scan [19] as it appears in the PLANE-SWEEP algorithm [4]: We are given n points *sorted* according to their x-coordinates. The task is to perform a scan over the sequence by repeatedly considering triples (p, q, r) of points and eliminate q if there is not a right turn at q when moving from p to r via q. After this computation the points on the upper hull are gathered together at the beginning of the input. The scan is carried out in place. Typically, about $2n$ orientation tests are done in such a scan.

For three points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$, the orientation test boils down to the question of determining the sign of a 3×3 determinant:

$$\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}$$

If the sign of this determinant is positive, then r is on left of the oriented line determined by p and q. If the sign of the determinant is negative, then r is on right of the oriented line. And if the sign is zero, then r is on that line.

Formulated in another way, one can calculate $\mathit{lhs} = (q_x - p_x) * (r_y - p_y)$ and $\mathit{rhs} = (r_x - p_x) * (q_y - p_y)$, and then determine whether $\mathit{lhs} > \mathit{rhs}$, $\mathit{lhs} < \mathit{rhs}$, or $\mathit{lhs} == \mathit{rhs}$. When doing these calculations, we considered three alternatives:

Multiple-precision arithmetic. It is clear that (1) a construction of a signed value from an unsigned one may increase the length of the representation by one bit, (2) a subtraction may increase the needed precision by one bit, and (3) a multiplication may double the needed precision. Hence, if the coordinates of the points are `w` bits wide, it is sufficient to use $(2\mathtt{w} + 4)$-bit integers to get the correct value of *lhs* and *rhs*. For this purpose, we used the multiple-precision integers available at the CPH STL [24].

Double-precision arithmetic. By converting the coordinates to double-length integers and by handling the possible overflows in an ad-hoc manner, the calculations can be done with 2`w`-bit numbers and some **if** statements.

Floating-point filter. Many computers have additional hardware to accelerate computations on floating-point numbers. Therefore, it might be advantageous to perform the calculations on floating-point numbers and, only if the result is not known to be correct due to accumulation of rounding errors, use one of the above-mentioned exact methods to redo the calculations. In the present case, we employed Shewchuk's filter coded in [31].

In this micro-benchmark the coordinates were 32-bit integers of type **int**. Hence, in the multiple-precision solution the numbers were 68 bits wide. In the double-precision solution we could rely on 64-bit built-in integers of types **long long** and **unsigned long long**. For randomly generated data, the floating-point filter worked with 100 % accuracy. Table 2 shows how Graham's scan performed for different right-turn predicates. Generally, this linear-time computa-

Table 2. Performance of Graham's scan for different right-turn predicates for the x-sorted square data set [ns per point]. The width `w` of the coordinates was 32 bits. In the multiple-precision solution, intermediate calculations were done with numbers of type `cphstl`::$\mathbb{Z}$<2 ∗ `w` + 4>.

n	Multiple precision	Double precision	Floating-point filter
2^{10}	30.8	17.7	15.7
2^{15}	30.8	17.8	15.3
2^{20}	30.7	17.7	15.3
2^{25}	30.9	17.9	15.4
2^{30}	55.8	39.8	48.8

tion is by no means critical compared to the cost of sorting. Since the floating-point filter gave the best results, we used it in all subsequent experiments.

Distribution. To explore different options when implementing distributive methods, consider the task of producing a histogram of bucket sizes along the x-axis when we are given n points and a distribution table of size `m`. This involves a scan over the input and, for each point. the calculation of its bucket index and an increment of the counter at that bucket. In the micro-benchmark we varied the table size and the data type used in arithmetic operations. For each of the considered types, the x-coordinates were first cast to this type, the calculations were done with them, and the result was cast back to an index type.

The results of these experiments are reported in Table 3 for several different table sizes and data types. For multiple-precision integer arithmetic, the package from the CPH STL was used [24]. These results confirm two things:

1. The distribution table should not be large; otherwise, caching effects will become visible.
2. It is preferable to do the bucket calculations using floating-point numbers.

Scanning. Next, let us consider what is the cost of sequential scanning. Three tasks reappear in several algorithms: (1) Find the minimum and maximum of the points according to their lexicographic order. For n points, this task can be accomplished with about $(3/2)n$ point comparisons using the standard-library function `std::minmax_element`. (2) Use two points p and r to partition a sequence of n points into two parts so that the points above and on the line determined by p and r come on the left and those below the line on the right. For this task, the standard-library function `std::partition` can be employed. The orientation predicate is needed to determine on which side of the line a point lies. (3) Copy a sequence of size n to an array. The function `std::copy` is designed for this task.

Table 4 shows the performance of these functions in our test environment: `std::minmax_element` is very fast, whereas `std::partition` is slower since it involves point moves and orientation tests. Especially, for the largest instance, when the size of the input is close to the maximum capacity of main memory, the slowdown is noticeable. The performance is linear up to a certain point, but after that point the memory operations became more expensive. In the test computer, the

Table 3. Performance of the histogram creation along the x-axis for the square data set [ns per point]. In all runs, n was fixed to 2^{20}. The coordinates were of type **int**; their width `w` was 32 bits. With integers of width $2\texttt{w}+3$, all arithmetic overflows could be avoided. Integers of width $2\texttt{w}$ were also safe since $\lg \texttt{m} \leq \texttt{w} - 3$.

m	**double**	cphstl::$\mathbb{Z}$<2 * w + 3>	cphstl::$\mathbb{Z}$<2 * w>
2^{10}	3.90	37.91	10.96
2^{11}	3.85	39.04	11.05
2^{12}	3.90	38.99	10.87
2^{13}	3.92	38.97	10.76
2^{14}	4.19	39.14	11.07
2^{15}	4.07	38.90	10.96
2^{16}	4.74	39.10	11.81
2^{17}	5.03	39.27	11.98
2^{18}	5.21	38.97	12.09
2^{19}	7.77	48.81	18.86
2^{20}	13.31	75.94	35.82

Table 4. Performance of scanning for the square data set [ns per point].

n	`std::minmax_element`	`std::partition`	`std::copy`
2^{10}	0.22	11.2	0.73
2^{15}	0.20	10.3	0.80
2^{20}	0.20	10.4	1.41
2^{25}	0.20	11.6	3.13
2^{30}	0.20	34.5	out of memory

saturation point was reached for $n = 2^{28}$; for $n = 2^{27}$, the cost per point was still about the same as that for $n = 2^{25}$. For the largest instance, copying failed since there was not space for two point arrays of size 2^{30} in main memory.

Sorting. Most industry-strength sorting algorithms are hybrids. The C++ standard-library INTROSORT [26] is a typical example: It uses MEDIAN-OF-THREE QUICKSORT [32] (see also [23]) for rough sorting and it finishes its job by a final INSERTIONSORT scan. If the recursion stack used by QUICKSORT becomes too deep, the whole input will be processed by HEAPSORT [33]. Now, small inputs are processed fast due to INSERTIONSORT, the worst case is $O(n \lg n)$ for an input of size n due to HEAPSORT, and the performance is good due to QUICKSORT.

We wanted to test whether a combination, where the input elements are distributed into buckets and the buckets sorted by INTROSORT, can improve the performance even further. In our implementation we followed closely the guidelines given in [27]. We name the resulting algorithm ONE-PHASE BUCKETSORT. This algorithm has two drawbacks: (1) Its interface is not the same as that of the library sort. Namely, it has one additional functor as a parameter that is used to map every element to a numerical value. In our application this is not a problem since the coordinates of the points are integers. (2) For an input of size n, the

Table 5. Performance of sorting for the square data set [ns per point].

n	INTROSORT	ONE-PHASE BUCKETSORT	TWO-PHASE BUCKETSORT
2^{10}	36.5	14.2	33.4
2^{15}	53.5	19.9	29.4
2^{20}	70.7	33.1	43.7
2^{25}	89.1	58.9	63.0
2^{30}	173	out of memory	324

algorithm—as implemented in [27]—requires an extra array for n elements and a header array for $\texttt{m} = \min\{n/5, \texttt{max\_m}\}$ integers. The bucket headers are used as counters to keep track of the size of the buckets and as cursors when placing the elements into the buckets. We selected the constant `max_m` such that the whole header array could be stored in the second-level cache.

To make the distributive approach competitive with respect to space usage, we also implemented TWO-PHASE BUCKETSORT that distributes the input elements into $O(\sqrt{n}\,)$ buckets and sorts each of them using ONE-PHASE BUCKETSORT. This version permutes the elements inside the input sequence before sorting the buckets. For the two-phase version the linear running time is valid as long as $\sqrt{n}$ is not significantly larger than `max_m`.

In the benchmark we sorted an array of n points according to their x-coordinates using the above-mentioned sorting algorithms. As seen from Table 5, both versions of BUCKETSORT worked reasonably well compared to INTROSORT until the size of the input reached that of main memory. In this situation, most of the other memory intensive programs had troubles as well—either they failed due to excessive memory usage or became slow.

4 Competitors

Any algorithm solving the convex-hull problem should read the whole input and report the extreme points in the output in sorted angular order. Thus, if $scan(n)$ denotes the cost of scanning a sequence of size n sequentially and $sort(h)$ the cost of sorting a sequence of size h, $\Omega(scan(n) + sort(h))$ is a lower bound for the running time of any convex-hull algorithm.

Many algorithms have been devised for the convex-hull problem, but none of them is known to match the above-mentioned lower bound on the word RAM [21]—when the coordinates of the points are integers that fit in one word each. The best deterministic algorithms are known to run in $O(n \lg\lg n)$ worst-case time (for example, the PLANE-SWEEP algorithm [4] combined with fast integer sorting [22]) or in $O(n \lg h)$ worst-case time (the MARRIAGE-BEFORE-CONQUEST algorithm [25]). On the other hand, when the input points are drawn according to some random distribution, which is a prerequisite for the analysis, there are algorithms that can solve the convex-hull problem in linear expected time. This is not in conflict with the $\Omega(scan(n) + sort(h))$ bound, since integers drawn independently at random from a uniform distribution in a bounded interval can be sorted in linear expected time by BUCKETSORT.

The most noteworthy alternatives for a practical implementation are:

Plane sweep. The PLANE-SWEEP algorithm [4] is a variation of ROTATIONAL-SWEEP [19] where the points are sorted according to their x-coordinates. The problem is solved by computing the upper-hull and lower-hull chains separately. To start with, two extreme points are found—one on the left and another on the right. Then the line segment determined by these two is used to partition the input into upper-hull candidates and lower-hull candidates. Finally, the upper-hull candidates are scanned from left to right and the lower-hull candidates from the right to left as in Graham's algorithm [19]. To work in linear expected time, BUCKETSORT (see, for example, [1]) could be used when sorting the candidate collections. If INTROSORT was used to sort the buckets, the worst-case running time would be $O(n \lg n)$.

Divide and conquer. As is standard in the DIVIDE & CONQUER scheme [29], if the number of given points is less than some constant, the problem is solved directly using some straightforward method. Otherwise, the problem is divided into two subproblems of about equal size, these subproblems are solved recursively, and the resulting convex hulls of size h_1 and h_2 are merged in $O(h_1 + h_2)$ worst-case time. An efficient merging procedure guarantees that the worst-case running time of the algorithm is $O(n \lg n)$ and, for sparse-hulled distributions, the average-case running time is $O(n)$ [7]. For the theoretical analysis to hold, two properties are important: (1) the division step must be accomplished in $O(1)$ time and (2) the points in the subproblems must obey the same distribution as the original points. This can be achieved by storing the points in an array and shuffling the input randomly at the beginning of the computation.

Quickhull. This algorithm, which mimics QUICKERSORT [30], has been reinvented several times (see, e.g. [10,16,20]). It also starts by finding two extreme points p and r, one on the left and another on the right, and computes the upper-hull chain from p to r and the lower-hull chain from r to p separately. For concreteness, consider the computation of the upper chain from p to r. In the general step, when the problem is still large enough, the following is done: (1) Find the extreme point q with the largest distance from the line segment $\overline{pr}$. (2) Eliminate the points inside the triangle pqr from further consideration. (3) Compute the chain from p to q recursively by considering the points above the line segment $\overline{pq}$ and (4) the chain from q to r by considering the points above the line segment $\overline{qr}$. (5) Concatenate the chains produced by the recursive calls and return that chain.

Eddy [16] proved that in the worst case QUICKHULL runs in $O(nh)$ time. However, Overmars and van Leeuwen [28] proved that in the average case the algorithm runs in $O(n)$ expected time. Furthermore, if the coordinates of the points are integers drawn from a bounded universe of size U, the worst-case running time of QUICKHULL is $O(n \lg U)$ [17].

Throw-away elimination. When computing the convex hull for a sequence 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, e.g. [2,4,12]), by eliminating the points falling inside such an approximative hull, the problem size can often be reduced considerably. After this preprocessing, any of the above-mentioned algorithms could be used to process the remaining points. Akl and Toussaint [2] used four equispaced directions—those determined by the coordinate axes; and Devroye and Toussaint [12] used eight equispaced directions. The first of these papers demonstrated the usefulness of this idea experimentally and the second paper verified theoretically that for certain random distributions the number of points left will be small. Unfortunately, the result depends heavily on the shape of the domain, from where the points are drawn at random. A rectangle is fine, but a circle is not.

Our implementations of these algorithms are available from the website of the CPH STL [`http://www.cphstl.dk`] in the form of a `pdf` file and a `tar` archive [18]. To ensure the reproducibility of the experimental results, the package also contains the driver programs used in the experiments.

5 Experiments

As the micro-benchmarks indicated, the performance of the memory system became an issue when the problem size reached the capacity of main memory. Therefore, it became a matter of honour for us to ensure that our programs can also handle large problem instances—an "out-of-memory" signal was not acceptable if the problem instance fitted in internal memory.

Briefly stated, the improvements made to the algorithms were as follows:

Plane sweep. As our starting point, we used the in-situ version described in [17]. To achieve the linear expected running time, TWO-PHASE BUCKETSORT was used when sorting the points according to their x-coordinates.

Divide and conquer. As in PLANE-SWEEP, the upper and lower hulls were computed separately. When merging two sorted chains of points, we used an in-place merging algorithm (`std::inplace_merge`). The library routine was adaptive: If there was free memory available, it was used; if not, an in-place merging routine was employed. It must be pointed out that the emergency routine did not run in linear worst-case time [15]—although it relied on sequential access. Therefore, when the memory limit was hit, the worst-case running time of the DIVIDE & CONQUER algorithm was $O(n(\lg n)^2)$.

Quickhull. We implemented this algorithm recursively by letting the runtime system handle the recursion stack. Otherwise, we relied on many of the same in-place routines as those used in the other algorithms. The extra space required by the recursion stack is $O(\lg U)$ words when the coordinates come from a bounded universe of size U [17].

Bucketing. The bucketing was done by maintaining information on the `min` and `max` y-values in each of the $\lceil \sqrt{n}\, \rceil$ slabs. After determining the staircases, the points outside them were moved to the beginning of the sequence and in-situ PLANE-SWEEP was used to finish the job. The elimination overhead was three scans: one min-max scan to find the extreme points in the $\pm x$ directions, one scan to determine the outermost points at each slab in the $\pm y$ directions, and one partitioning scan to eliminate the points that were inside the region determined by the staircases. The slab structure used $O(\sqrt{n}\,)$ words of space and, except the above-mentioned scans, its processing cost was $O(\sqrt{n}\,)$. Bucket indices were calculated using floating-point numbers.

Throw-away elimination. We found the extreme points in eight predetermined directions, eliminated all the points inside the convex polygon determined by them, and processed the remaining points with in-situ PLANE-SWEEP. Thus, the elimination overhead was two scans: one max-finding scan to find the extrema and another partitioning scan to do the elimination.

We wanted to test the performance of these heuristics in their home ground when the input points were drawn according to some random distribution. We run the experiments on the following data sets—the first one was already used in the micro-benchmarks:

Square data set. The coordinates of the points were integers drawn randomly from the range $[\![-2^{31} \mathinner{..} 2^{31})\!)$. The expected size of the output is $O(\lg n)$ [6].

Disc data set. As above, the coordinates of the points were random **int**s, but only the points inside the circle centred at the origin with the radius $2^{31} - 1$ were accepted to the input. Here the expected size of the output is $O(n^{1/3})$.

In earlier studies, the number of orientation tests and that of coordinate comparisons have been targets for optimization. Since the algorithms reorder the points in place, the number of coordinate moves is also a relevant performance indicator. In our first experiments, we measured the performance of the algorithms with respect to these indicators. The purpose was to confirm that the algorithms execute a linear number of basic operations. Due to hardware effects, it might be difficult to see the linearity from the CPU-time measurements.

The operation counts are shown in Table 6 (orientation tests), Table 7 (coordinate comparisons), and Table 8 (coordinate moves). The results are unambiguous: (1) There are no significant fluctuations in the results. For all competitors, the observed performance is linear—as the theory predicts. For the elimination strategies, the number of coordinate moves is sublinear; for BUCKETING this is even the case for orientation tests. (2) For BUCKETING some of the observed

Table 6. The number of orientation tests executed [per n] for the square data set.

n	PLANE-SWEEP	DIVIDE & CONQUER	QUICKHULL	BUCKETING	THROW-AWAY
2^{10}	2.31	2.76	4.95	0.16	2.04
2^{15}	2.34	2.79	4.86	0.02	2.01
2^{20}	2.34	2.80	4.82	0.00	2.00
2^{25}	2.39	2.85	4.70	0.00	2.00
2^{30}	2.09	2.56	5.37	0.00	2.00

Table 7. The number of coordinate comparisons done [per n] for the square data set.

n	PLANE-SWEEP	DIVIDE & CONQUER	QUICKHULL	BUCKETING	THROW-AWAY
2^{10}	11.0	8.81	2.26	6.31	15.0
2^{15}	9.45	8.84	2.25	5.65	15.0
2^{20}	9.01	8.88	2.25	5.53	15.0
2^{25}	8.91	8.80	2.25	5.51	15.1
2^{30}	8.88	8.88	2.25	5.50	15.2

Table 8. The number of coordinate moves performed [per n] for the square data set.

n	PLANE-SWEEP	DIVIDE & CONQUER	QUICKHULL	BUCKETING	THROW-AWAY
2^{10}	26.5	32.4	5.14	2.49	1.21
2^{15}	29.1	32.7	4.59	0.42	0.20
2^{20}	29.0	32.6	4.51	0.08	0.04
2^{25}	29.0	32.6	4.72	0.01	0.00
2^{30}	29.1	32.7	4.91	0.00	0.00

Table 9. The average fraction of points left after elimination [%] for the two data sets.

n	BUCKETING (square)	BUCKETING (disc)	THROW-AWAY (square)	THROW-AWAY (disc)
2^{10}	8.07	12.2	4.29	12.2
2^{15}	1.18	2.07	0.71	10.2
2^{20}	0.19	0.36	0.12	9.99
2^{25}	0.03	0.06	0.01	9.97
2^{30}	0.00	0.01	0.00	9.97

values are frighteningly small compared to others albeit casts to floating-point numbers and operations on them were not measured.

Both BUCKETING and THROW-AWAY eliminate some points before applying in-situ PLANE-SWEEP. To understand better how efficient these elimination strategies are, we measured the average fraction of points left after elimination. The figures are reported in Table 9. For the square data set, both methods show extremely good elimination efficiency. For the disc data set, BUCKETING will be better since the expected number of points in the outer layer is at most $O(\sqrt{n}\,(\lg n/\lg\lg n))$ [14, Section 4.4], whereas for THROW-AWAY the expected number of points left is $\Omega(n)$ [17, Fact 3]. Although the elimination efficiency can vary, the performance of these algorithms can never be much worse than that of PLANE-SWEEP since the elimination overhead is linear.

In our final experiments, we measured the CPU time used by the competitors for the considered data sets. The results are shown in Table 10 (square) and Table 11 (disc). We can say that (1) PLANE-SWEEP had mediocre performance; (2) DIVIDE & CONQUER worked at about the same efficiency; (3) THROW-AWAY improved the performance with its preprocessing; (4) QUICKHULL was not effec-

Table 10. The running time of the competitors for the square data set [ns per point].

n	PLANE-SWEEP	DIVIDE & CONQUER	QUICKHULL	BUCKETING	THROW-AWAY
2^{10}	60.5	73.2	49.5	14.3	20.4
2^{15}	56.9	70.0	46.0	6.98	16.4
2^{20}	62.6	69.7	45.7	5.88	16.4
2^{25}	85.9	69.8	48.4	5.50	15.2
2^{30}	189.9	114.2	124.5	75.4	45.5

Table 11. The running time of the competitors for the disc data set [ns per point].

n	PLANE-SWEEP	DIVIDE & CONQUER	QUICKHULL	BUCKETING	THROW-AWAY
2^{10}	57.7	71.0	60.4	16.8	29.2
2^{15}	53.7	68.4	57.8	7.64	27.4
2^{20}	56.9	67.0	57.6	5.86	28.5
2^{25}	80.2	66.7	57.8	5.53	30.1
2^{30}	183.9	116.9	128.3	75.9	77.2

tive because it performed many expensive orientation tests; and (5) BUCKETING showed outstanding performance compared to its competitors thanks to its effective preprocessing. However, THROW-AWAY that uses less space and has better memory-access locality behaved well when almost all memory was in use.

6 Reflections

When we started this work, we were afraid of that this would become a study on sorting in new clothes. But the essence turned out to be how to avoid sorting.

When writing the programs and performing the experiments, we made many mistakes. We have collected the following checklist of the most important issues to prevent ourselves and others from repeating these mistakes.

Compiler options. Switch all compiler warnings on when developing the programs and use full optimization when running the experiments.

Library facilities. Use the available library resources; do not reinvent them.

Techniques. Keep bucketing in your toolbox. While the other convex-hull algorithms are busily partitioning the input into upper-hull and lower-hull candidates, BUCKETING has already solved the problem. Partitioning is also needed in BUCKETING, but it is fast since the partitioning criterion is simpler and only a small fraction of the points will be moved.

Robustness. Implement geometric primitives in a robust manner. The simplest way to achieve this is to do intermediate calculations with multiple-precision numbers. We found it surprising that multiple-precision arithmetic is not provided by the C++ standard library.

Floating-point acceleration. Use floating-point numbers wisely. These can speed up things; we used them in bucket calculations and orientation tests.

Space efficiency. Do not waste space. As one of the micro-benchmarks showed, an innocent copying can generate an "out-of-memory" signal when the problem size reaches the capacity of main memory. It is known that the PLANE-SWEEP algorithm can be implemented in place (see [9,17]). On the other hand, a program can be useful even though its memory usage is not optimal; it is quite acceptable to use $O(\sqrt{n})$ words of extra space.

Correctness. Use an automated test framework. Our first checkers could verify the correctness of the output in $O(n^2)$ worst-case time. After refactoring, the checkers were improved to carry out this task in $O(n \lg n)$ worst-case time (for more details, see [17,18]). But the checkers do some copying which means that they cannot be used for the largest problem instances.

Quality assurance. Try several alternatives for the same task to be sure about the quality of the chosen alternative.

References

1. Akl, S.G., Meijer, H.: On the average-case complexity of "bucketing" algorithms. J. Algorithms **3**(1), 9–13 (1982). https://doi.org/10.1016/0196-6774(82)90003-7
2. Akl, S.G., Toussaint, G.T.: A fast convex hull algorithm. Inf. Process. Lett. **7**(5), 219–222 (1978). https://doi.org/10.1016/0020-0190(78)90003-0
3. Allison, D.C.S., Noga, M.T.: Some performance tests of convex hull algorithms. BIT **24**(1), 2–13 (1984). https://doi.org/10.1007/BF01934510
4. Andrew, A.M.: Another efficient algorithm for convex hulls in two dimensions. Inf. Process. Lett. **9**(5), 216–219 (1979). https://doi.org/10.1016/0020-0190(79)90072-3
5. Asano, T., Edahiro, M., Imai, H., Iri, M., Murota, K.: Practical use of bucketing techniques in computational geometry. In: Toussaint, G.T. (ed.) Computational Geometry. North-Holland (1985)
6. Bentley, J.L., Kung, H.T., Schkolnick, M., Thompson, C.D.: On the average number of maxima in a set of vectors and applications. J. ACM **25**(4), 536–543 (1978). https://doi.org/10.1145/322092.322095
7. Bentley, J.L., Shamos, M.I.: Divide and conquer for linear expected time. Inf. Process. Lett. **7**(2), 87–91 (1978). https://doi.org/10.1016/0020-0190(78)90051-0
8. Berberich, E., Hagen, M., Hiller, B., Moser, H.: Experiments. In: Müller-Hannemann, M., Schirra, S. (eds.) Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice. Springer-Verlag (2010)
9. Brönnimann, H., Iacono, J., Katajainen, J., Morin, P., Morrison, J., Toussaint, G.: Space-efficient planar convex hull algorithms. Theoret. Comput. Sci. **321**(1), 25–40 (2004). https://doi.org/10.1016/j.tcs.2003.05.004
10. Bykat, A.: Convex hull of a finite set of points in two dimensions. Inf. Process. Lett. **7**(6), 296–298 (1978). https://doi.org/10.1016/0020-0190(78)90021-2
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
12. Devroye, L., Toussaint, G.T.: A note on linear expected time algorithms for finding convex hulls. Computing **26**(4), 361–366 (1981). https://doi.org/10.1007/BF02237955
13. Devroye, L.: Expected time analysis of algorithms in computational geometry. In: Toussaint, G.T. (ed.) Computational Geometry. North-Holland (1985)

14. Devroye, L.: Lecture Notes on Bucket Algorithms. Birkhäuser Boston, Inc. (1986)
15. Dvořák, S., Ďurian, B.: Stable linear time sublinear space merging. Comput. J. **30**(4), 372–375 (1987). https://doi.org/10.1093/comjnl/30.4.372
16. Eddy, W.F.: A new convex hull algorithm for planar sets. ACM Trans. Math. Software **3**(4), 398–403 (1977). https://doi.org/10.1145/355759.355766
17. Gamby, A.N., Katajainen, J.: Convex-hull algorithms: Implementation, testing, and experimentation. Algorithms **11**(12) (2018). https://doi.org/10.3390/a11120195
18. Gamby, A.N., Katajainen, J.: Convex-hull algorithms in C++. CPH STL report 2018-1, Dept. Comput. Sci., Univ. Copenhagen (2018–2019), `http://www.diku.dk/~jyrki/Myris/GK2018S.html`
19. Graham, R.L.: An efficient algorithm for determining the convex hull of a finite planar set. Inf. Process. Lett. **1**(4), 132–133 (1972). https://doi.org/10.1016/0020-0190(72)90045-2
20. Green, P.J., Silverman, B.W.: Constructing the convex hull of a set of points in the plane. Comput. J. **22**(3), 262–266 (1979). https://doi.org/10.1093/comjnl/22.3.262
21. Hagerup, T.: Sorting and searching on the word RAM. In: Morvan, M., Meinel, C., Krob, D. (eds.) STACS 1998. LNCS, vol. 1373, pp. 366–398. Springer (1998). https://doi.org/10.1007/BFb0028575
22. Han, Y.: Deterministic sorting in $O(n \log \log n)$ time and linear space. J. Algorithms **50**(1), 96–105 (2004). https://doi.org/10.1016/j.jalgor.2003.09.001
23. Hoare, C.A.R.: Quicksort. Comput. J. **5**(1), 10–16 (1962). https://doi.org/10.1093/comjnl/5.1.10
24. Katajainen, J.: Class templates `cphstl::`$\mathbb{N}$ and `cphstl::`$\mathbb{Z}$ for fixed-precision arithmetic. Work in progress (2017–2019)
25. Kirkpatrick, D.G., Seidel, R.: The ultimate planar convex hull algorithm? SIAM J. Comput. **15**(1), 287–299 (1986). https://doi.org/10.1137/0215021
26. Musser, D.R.: Introspective sorting and selection algorithms. Software Pract. Exper. **27**(8), 983–993 (1997). https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-%23
27. Nevalainen, O., Raita, T.: An internal hybrid sort algorithm revisited. Comput. J. **35**(2), 177–183 (1992). https://doi.org/10.1093/comjnl/35.2.177
28. Overmars, M.H., van Leeuwen, J.: Further comments on Bykat's convex hull algorithm. Inf. Process. Lett. **10**(4–5), 209–212 (1980). https://doi.org/10.1016/0020-0190(80)90142-8
29. Preparata, F.P., Hong, S.J.: Convex hulls of finite sets of points in two and three dimensions. Commun. ACM **20**(2), 87–93 (1977). https://doi.org/10.1145/359423.359430
30. Scowen, R.S.: Algorithm 271: Quickersort. Commun. ACM **8**(11), 669–670 (1965). https://doi.org/10.1145/365660.365678
31. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust predicates for computational geometry (1996), `http://www.cs.cmu.edu/~quake/robust.html`
32. Singleton, R.C.: Algorithm 347: An efficient algorithm for sorting with minimal storage [M1]. Commun. ACM **12**(3), 185–187 (1969). https://doi.org/10.1145/362875.362901
33. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM **7**(6), 347–348 (1964). https://doi.org/10.1145/512274.512284