# In-Place Planar Convex Hull Algorithms*

Hervé Brönnimann[1], John Iacono[1], Jyrki Katajainen[2], Pat Morin[3], Jason Morrison[4], and Godfried Toussaint[3]

[1] CIS, Polytechnic University, Six Metrotech, Brooklyn, New York, 11201. {hbr,jiacono}@poly.edu

[2] Department of Computing, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark. jyrki@diku.dk

[3] SOCS, McGill University, 3480 University St., Suite 318, Montréal, Québec, CANADA, H3A 2A7. {morin,godfried}@cgm.cs.mcgill.ca

[4] School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, CANADA, K1S 5B6. morrison@cs.carleton.ca

**Abstract.** An in-place algorithm is one in which the output is given in the same location as the input and only a small amount of additional memory is used by the algorithm. In this paper we describe three in-place algorithms for computing the convex hull of a planar point set. All three algorithms are optimal, some more so than others...

## 1 Introduction

Let $S = \{S[0], \ldots, S[n-1]\}$ be a set of $n$ distinct points in the Euclidean plane. The *convex hull* of $S$ is the minimal convex region that contains every point of $S$. From this definition, it follows that the convex hull of $S$ is a convex polygon whose vertices are points of $S$. For convenience, we say that a point $p$ is "on the convex hull of $S$" if $p$ is a vertex of the convex hull of $S$.

As early as 1973, Graham [13] gave a convex hull algorithm with $O(n \log n)$ worst-case running time. Shamos [32] later showed that, in any model of computation where sorting has an $\Omega(n \log n)$ lower bound, every convex hull algorithm must require $\Omega(n \log n)$ time for some inputs. Despite these matching upper and lower bounds, and probably because of the many applications of convex hulls, a number of other planar convex hull algorithms have been published since Graham's algorithm [1,2,4,6,11,17,27,28,21,35].

Of particular note is the "Ultimate(?)" algorithm of Kirkpatrick and Seidel [21] that computes the convex hull of a set of $n$ points in the plane in $O(n \log h)$ time, where $h$ is the number of vertices of the convex hull. The same authors show that, on algebraic decision trees of any fixed order, $\Omega(n \log h)$ is a lower bound for computing convex hulls of sets of $n$ points having convex hulls with $h$ vertices.

* This research was partly funded by the National Science Foundation, the Natural Sciences and Engineering Research Council of Canada and the Danish Natural Science Research Council under contract 9801749 (project Performance Engineering).

Because of the importance of planar convex hulls, it is natural to try and improve the running time and storage requirements of planar convex hull algorithms. In this paper, we focus on reducing the intermediate storage used in the computation of planar convex hulls. In particular, we describe in-place and *in situ* algorithms for computing convex hulls. These are algorithms in which the input points are given as an array and the output, namely the vertices of the convex hull sorted in order of appearance on the hull, is returned in the same array. During the execution of the algorithm, additional working storage is kept to a minimum. In the case of in-place algorithms, the extra storage is kept in $O(1)$ while *in situ* algorithms allow an extra memory of size $O(\log n)$. After execution of the algorithm, the array contains exactly the same points, but in a different order.

In-place and *in situ* algorithms have several practical advantages over traditional algorithms. Primarily, because they use only a small additional amount of storage, they allow for the processing of larger data sets. Any algorithm that uses separate input and output arrays will, by necessity, require enough memory to store $2n$ points. In contrast, an *in situ* or in-place algorithm needs only enough memory to store $n$ points plus $O(\log n)$ or $O(1)$ working space, respectively. Related to this is the fact that *in situ* and in-place algorithms usually exhibit greater locality of reference, which makes them very practical for implementation on modern computer architectures with memory hierarchies. A final advantage of *in situ* algorithms, especially in mission critical applications, is that they are less prone to failure since they do not require the allocation of large amounts of memory that may not be available at run time.

We describe three planar convex hull algorithms. The first is in-place, uses Graham's scan in combination with an in-place sorting algorithm, and runs in $O(n \log n)$ time. The second algorithm runs in $O(n \log h)$ time, is *in situ* and is based on an algorithm of Chan *et al.* [4]. The third ("More Ultimate?") algorithm is based on an algorithm of Chan [3], runs in $O(n \log h)$ time and is in-place. The first two algorithms are simple, implementable, and efficient in practice. To justify this claim, we have implemented both algorithms and made the source code freely available [25].

To the best of our knowledge, this paper is the first to study the problem of computing convex hulls using *in situ* and in-place algorithms. This seems surprising, given the close relation between planar convex hulls and sorting, and the large body of literature on *in situ* sorting and merging algorithms [7,8,9,10,12,15,16,19,20,18,23,26,30,33,34,36].

The remainder of the paper is organized as follows: Sections 2, 3 and 4 describe our first, second and third algorithms, respectively. Section 5 summarizes and concludes with open problems.

## 2 An $O(n \log n)$ Time Algorithm

In this section, we present a simple in-place implementation of Graham's convex hull algorithm [13] or, more precisely, Andrew's modification of Graham's algorithm [1]. The algorithm requires the use of an in-place sorting algorithm. This can be any efficient in-place sorting algorithm (see, for example, [18,36]), so we refer to this algorithm simply as InPlace-Sort.

Because this algorithm is probably the most practically relevant algorithm given in this paper, we begin by describing the most conceptually simple version of the algorithm, and then describe a slightly more involved version that improves the constants in the running time.

### 2.1 The Basic Algorithm

The *upper convex hull* of a point set $S$ is the convex hull of $S \cup \{(0, -\infty)\}$. The *lower convex hull* of a point set $S$ is the convex hull of $S \cup \{(0, \infty)\}$. It is well-known that the convex hull of a point set is the intersection of its upper and lower convex hulls [29]. Graham's scan computes the upper (or lower) hull of an $x$-monotone chain incrementally, storing the partially computed hull on a stack. The addition of each new point involves removing zero or more points from the top of the stack and then pushing the new point onto the top of the stack.

The following pseudo-code uses the InPlace-Sort algorithm and Graham's scan to compute the upper or lower hull of the point set $S$. The parameter $d$ is used to determine whether the upper or lower hull is being computed. If $d = 1$, then InPlace-Sort sorts the points by increasing order of lexicographic $(x, y)$-values and the upper hull is computed. If $d = -1$, then InPlace-Sort sorts the points by decreasing order and the lower hull is computed. The value of $h$ corresponds to the number of elements on the stack.

In the following, and in all remaining pseudo-code, $S = S[0], \dots, S[n-1]$ is an array containing the input points. We use the C pointer notation $S + i$ to denote the array $S[i], \dots, S[n-1]$.

Graham-InPlace-Scan($S, n, d$)

1: Graham-InPlace-Sort($S, n, d$)
2: $h \leftarrow 1$
3: **for** $i \leftarrow 1 \dots n-1$ **do**
4:     **while** $h \geq 2$ **and not** right_turn($S[h-2], S[h-1], S[i]$) **do**
5:         $h \leftarrow h - 1$ { pop top element from the stack }
6:     **end while**
7:     **swap** $S[i] \leftrightarrow S[h]$
8:     $h \leftarrow h + 1$
9: **end for**
10: **return** $h$

It is not hard to verify that when the algorithm returns in Line 10, the elements of $S$ that appear on the upper (or lower) convex hull are stored in $S[0], \dots, S[h-1]$. In the case of an upper hull computation ($d = 1$), the hull vertices are sorted left-to-right (clockwise), while in the case of a lower hull computation ($d = -1$), the hull vertices are sorted right-to-left (also clockwise).

To compute the convex hull of the point set $S$, we proceed as follows (refer to Fig. 1): First we make a call to Graham-InPlace-Scan to compute the vertices of the upper hull of $S$ and store them in clockwise order at positions $S[0], \ldots, S[h-1]$. It follows that $S[0]$ is the bottommost-leftmost point of $S$ and that $S[h-1]$ is the topmost-rightmost point of $S$. We then use $h-1$ swaps to bring $S[0]$ to position $S[h-1]$ while keeping the relative ordering of $S[1], \ldots S[h-1]$ unchanged. Finally, we make a call to GRAHAM-INPLACE-SCAN to compute the lower convex hull of $S[h-2], \ldots, S[n-1]$ (which is also the lower convex hull of $S$). This stores the vertices of the lower convex hull in $S[h-2], \ldots, S[h+h'-2]$ in clockwise order. The end result is that the convex hull of $S$ is stored in $S[0], \ldots, S[h+h'-2]$ in clockwise order.

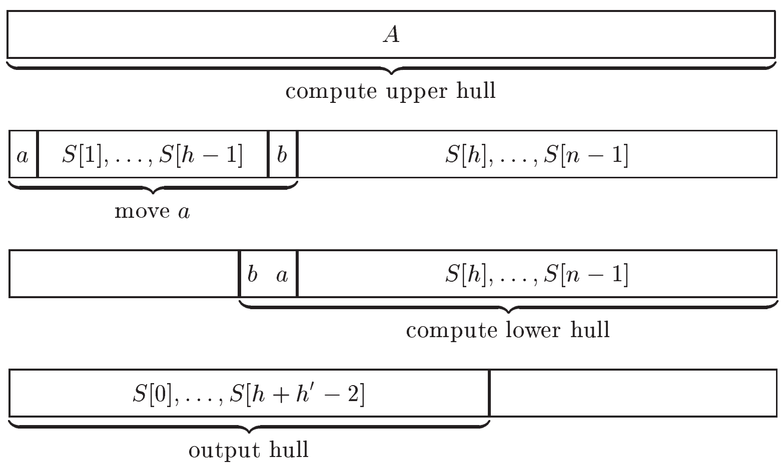


**Fig. 1.** The execution of the GRAHAM-INPLACE-HULL algorithm.

The following pseudo-code gives a more precise description of the algorithm.

GRAHAM-INPLACE-HULL$(S, n)$

1: $h \leftarrow$ GRAHAM-INPLACE-SCAN$(S, n, 1)$
2: **for** $i \leftarrow 0 \ldots h-2$ **do**
3:    **swap** $S[i] \leftrightarrow S[i+1]$
4: **end for**
5: $h' \leftarrow$ GRAHAM-INPLACE-SCAN$(S+h-2, n-h+2, -1)$
6: **return** $h+h'-2$

Each call to GRAHAM-INPLACE-SCAN executes in $O(n \log n)$ time, and the loop in lines 2–4 takes $O(h)$ time. Therefore, the total running time of the algorithm is $O(n \log n)$. The amount of extra storage used by INPLACE-SORT is $O(1)$, as is the storage used by both our procedures.

**Theorem 1** *Algorithm* GRAHAM-INPLACE-HULL *computes the convex hull of a set of $n$ points in $O(n \log n)$ time using $O(1)$ additional memory.*

### 2.2 The Optimized Algorithm

The constants in the running time of Graham-InPlace-Hull can be improved by first finding the extreme points $a$ and $b$ and using these points to partition the array into two parts, one that contains vertices that can only appear on the upper hull and one that contains vertices that can only appear on the lower hull. Fig. 2 gives a graphical description of this. In this way, each point (except $a$ and $b$) takes part in only one call to Graham-InPlace-Scan.

To further reduce the constants in the algorithm, one can implement InPlace-Sort with the in-place merge-sort algorithm of Katajainen *et al.* [18]. This algorithm requires only $n \log_2 n + O(n)$ comparisons and $\frac{3}{2} n \log_2 n + O(n)$ swaps to sort $n$ elements. Since Graham's scan performs only $2n - h$ right-turn tests when computing the upper hull of $n$ points having $h$ points on the upper hull, the resulting algorithm performs at most $3n - h$ right-turn tests (the extra $n$ comes from the initial partitioning step). We call this algorithm Opt-Graham-InPlace-Hull.

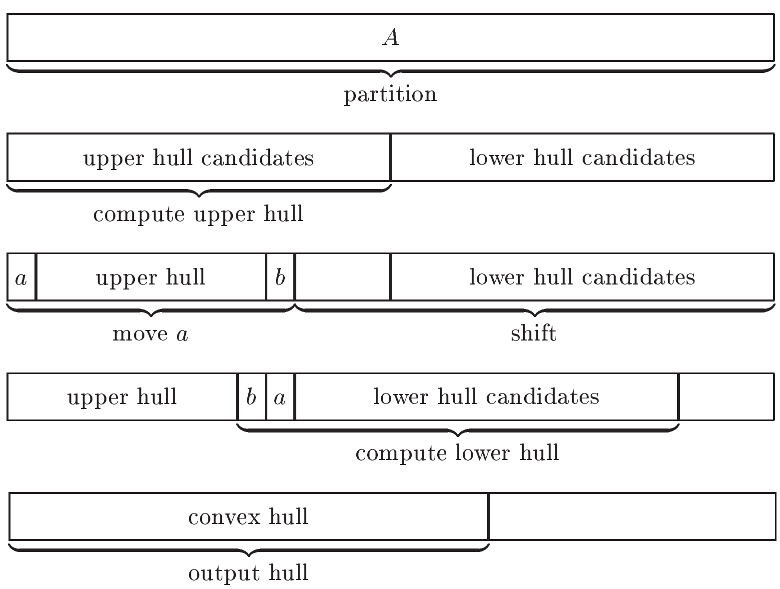


**Fig. 2.** A faster implementation of Graham-InPlace-Hull.

**Theorem 2** Opt-Graham-InPlace-Hull *computes the convex hull of $n$ points in $O(n \log n)$ time using at most $3n - h$ right turn tests, $\frac{3}{2} n \log_2 n + O(n)$ swaps, $n \log_2 n + O(n)$ lexicographic comparisons and $O(1)$ additional memory, where $h$ is the number of vertices of the convex hull.*

Finally, we note that if the array $A$ is already sorted in lexicographic order then no lexicographic comparisons are necessary. One can use an in-place stable partitioning algorithm to partition $A$ into the set of upper hull candidates and the set of lower hull candidates while preserving the sorted order within each set. There exists such an algorithm that runs in $O(n)$ time and perform $O(n)$ comparisons [19]. We call this algorithm Sorted-Graham-InPlace-Hull

**Theorem 3** Sorted-Graham-InPlace-Hull *computes the convex hull of* $n$ *points given in lexicographic order in* $O(n)$ *time using* $O(n)$ *right turn tests,* $O(n)$ *swaps, no lexicographic comparisons and* $O(1)$ *additional memory.*

## 3 An $O(n \log h)$ Time Recursive Algorithm

In this section, we show how to compute the upper (and symmetrically, lower) hull of $S$ in $O(n \log h)$ time using an *in situ* algorithm, where $h$ is the number of points of $S$ that on the upper (respectively, lower) hull of $S$. We begin with a review of the $O(n \log h)$ time algorithm of Chan *et al.* [4].

To compute the upper hull of a point set $S$, we begin by arbitrarily grouping the elements of $S$ into $\lfloor n/2 \rfloor$ pairs. From these pairs, the pair with median slope $s$ is found using a linear time median finding algorithm.[1] We then find a point $p \in S$ such that the line through $p$ with slope $s$ has all points of $S$ below it.

Let $q.x$ denote the $x$ coordinate of the point $q$ and let $\bar{i}$ denote the index of the element that is paired with $S[i]$. We now use $p$, and our grouping to partition the elements of $S$ into three groups $S^0$, $S^1$, and $S^2$ as follows (see Fig. 3):

$$S[i] \in \begin{cases} S^0 \text{ if } S[i].x \leq p.x \text{ and } (S[\bar{i}], p) \text{ is not above } S[i] \\ S^1 \text{ if } S[i].x > p.x \text{ and } (S[\bar{i}], p) \text{ is not above } S[i] \\ S^2 \text{ otherwise} \end{cases}$$

The algorithm then recursively computes the upper hull of $S^0 \cup \{p\}$ and $S^1 \cup \{p\}$ and outputs the concatenation of the two. For a discussion of correctness and a proof that this algorithm runs in $O(n \log h)$ time, see the original paper [4].

Now we turn to the problem of making this an *in situ* algorithm. The choice of median slope $s$ ensures that $S^0 \leq 3n/4$ and $S^1 \leq 3n/4$, so the algorithm uses only $O(\log n)$ levels of recursion. Our strategy is to implement each level using $O(1)$ local variables and one call to a median-finding routine that uses $O(\log n)$ additional memory.

For simplicity, assume $n$ is even. The case when $n$ is odd is easily handled by processing an extra unpaired element after all the paired elements have been processed. To pair off elements, we pair $S[i]$ with $S[i+1]$ if $i$ is even and with $S[i-1]$ if $i$ is odd. Several *in situ* linear time median finding algorithms exist (see, for example, Horowitz *et al.* [14, Section 3.6] or Lai and Wood [22]), so

[1] Bhattacharya and Sen [2] and Wenger [35] have both noted that median finding can be replaced by choosing a random pair of elements. The expected running time of the resulting algorithm is $O(n \log h)$.

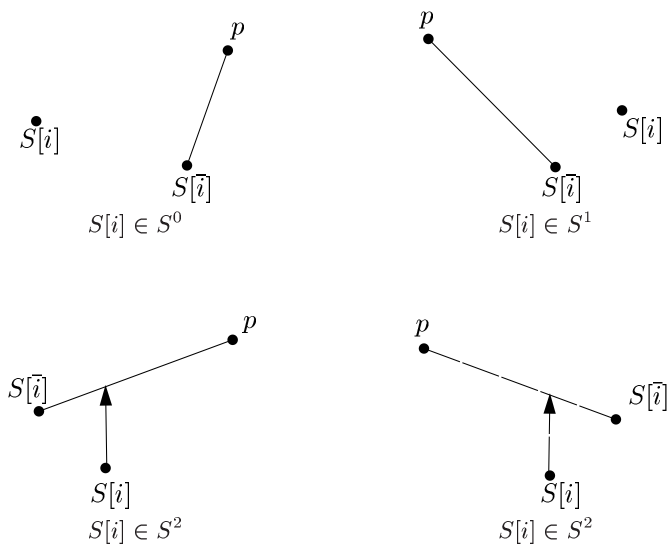


**Fig. 3.** Partitioning $S$ into $S^0$, $S^1$ and $S^2$.

finding the pair $(S[i], S[i+1])$ with median slope can be done with one of these. The tricky part of the implementation is the partitioning into sets $S^0$, $S^1$ and $S^2$. The difficulty lies in the fact that the elements are grouped into pairs, but the two elements of the same pair may belong to different sets $S^i$ and $S^j$.

We first note that we can compute the sizes $n_0$, $n_1$ and $n_2$ of these sets in linear time without difficulty by scanning $S$. Conceptually, we partition $S$ into three *files*, $f_0$, $f_1$ and $f_2$ that contain pairs of points in $S$. The file $f_0$ contains the elements $S[0], \dots, S[2\lfloor n_0/2\rfloor - 1]$. The file $f_1$ contains the elements $S[2\lfloor n_0/2\rfloor], \dots, S[2\lfloor (n_0+n_1)/2\rfloor - 1]$. The file $f_2$ contains the elements $S[2\lfloor (n_0+n_1)/2\rfloor], \dots, S[n]$.

It is important to note that these files are only abstractions. Each file $f_i$ is implemented using two integer values $r_i$ and $\phi_i$. The value of $r_i$ is initialized to the index of the first record in the file. The value of $\phi_i$ is initialized to $r_i + k_i$ where $k_i$ is the number of elements in $f_i$. A READ operation on $f_i$ returns the pair $(S[r_i], S[r_{i+1}])$ and increases the value of $r_i$ by 2. We say that $f_i$ is *empty* if $r_i \geq \phi_i$.

These files are used in conjunction with a stack $A$ that stores pairs of points. The stack and files serve two purposes: (1) when there is no data on the stack we read a pair from one of the files and store it on the stack, and (2) when we are about to overwrite an element from a pair that has not yet been placed on the stack, we read the pair from the file and save it on the stack. In this way no element is ever overwritten without first being saved on the stack, and the initial pairing of elements is preserved.

These ideas are made more concrete by the following pseudo-code, which places the elements of $S^0$ into array locations $S[0], \ldots, S[n_0 - 1]$, the elements of $S^1$ into array locations $S[n_0], \ldots, S[n_0 + n_1 - 1]$, and the elements of $S^2$ into array locations $S[n_0 + n_1], \ldots, S[n-1]$. The algorithm repeatedly processes pairs $(a, b)$ of elements by determining which of the three sets $a$ and $b$ belong to and then placing $a$ and $b$ in their correct locations.

CSY-PARTITION($S, n, n_0, n_1$)

$i_0 \leftarrow 0$<br>
$i_1 \leftarrow n_0$<br>
$i_2 \leftarrow n_0 + n_1$<br>
$m \leftarrow 0$<br>
**while** $m > 0$ **or** one of $f_0$, $f_1$, $f_2$ is not empty **do**<br>
    **if** $m = 0$ **then**<br>
        $A[m] \leftarrow$ READFROMFILE()<br>
        $m \leftarrow m + 1$<br>
    **end if**<br>
    $m \leftarrow m - 1$<br>
    $P \leftarrow A[m]$ { process this pair }<br>
    **for** both $q \in P$ **do**<br>
        $S^j \leftarrow$ GROUP($q, P$)<br>
        PLACE($q, i_j$)<br>
        $i_j \leftarrow i_j + 1$<br>
    **end for**<br>
**end while**

The READFROMFILE function simply reads a pair from one of the non-empty files and returns it. The GROUP($q, P$) returns (a pointer to) the group of point $q$ in the pair $P$. The PLACE($q, k$) function places the point $q$ at index $k$ in $S$, after ensuring that the overwritten element has been read and placed on the stack.

PLACE($q, k$)

**for** $i \leftarrow 0, 1, 2$ **do**<br>
    **if** $k \geq r_i$ **and** $k < \phi_i$ **then**<br>
        { $S[k]$ belongs to $f_i$ and has not yet been read }<br>
        READ($a, b$) from $f_i$<br>
        $A[m] \leftarrow (a, b)$<br>
        $m \leftarrow m + 1$<br>
    **end if**<br>
**end for**<br>
$S[k] \leftarrow q$

To show that this partitioning step is correct, we make 2 observations. (1) Exactly $n/2$ pairs of elements are read and processed since the file abstraction ensures that no pair is read more than once and the algorithm does not terminate until all files are empty. (2) The code in PLACE ensures that any pair is read and placed on the stack $A$ before an element of the pair is overwritten. Therefore, all of the original $n/2$ pairs of elements are processed and each element is placed into one of $S^0$, $S^1$ or $S^2$.

Since the algorithm uses a stack $A$ that may grow without bound, it is not obvious that the partitioning algorithm's additional memory is of a constant size. To prove that $A$ does not grow without bound note that overwriting $k_i$ elements of $f_i$ causes at most $\lceil k_i/2 \rceil$ read operations. Each iteration of the outer loop places one pair of elements, and each read operation reads one pair of elements. Therefore, the total number of read operations performed after $k$ iterations is at most $k+3$. However, each iteration removes 1 pair of elements from the stack $A$, so the total number of pairs on the stack after $k$ iterations is not more than 3. Since this holds for any value of $k$, the stack $A$ never holds more than 3 pairs of elements.

Fig. 4 recaps the algorithm for computing the upper hull of $S$. First the algorithm partitions $S$ into the sets $S^0$, $S^1$ and $S^2$. It then recurses on the set $S^0$. After the recursive call, the convex hull of $S^0$ is stored at the beginning of the array $S$, and the last element of this hull is the point $p$ that was used for partitioning. The algorithm then shifts $S^1$ leftward so that it is adjacent to $p$ and recurses on $S^1 \cup \{p\}$. The end result is the upper hull of $S$ being stored consecutively and in clockwise order at the beginning of the array $S$.

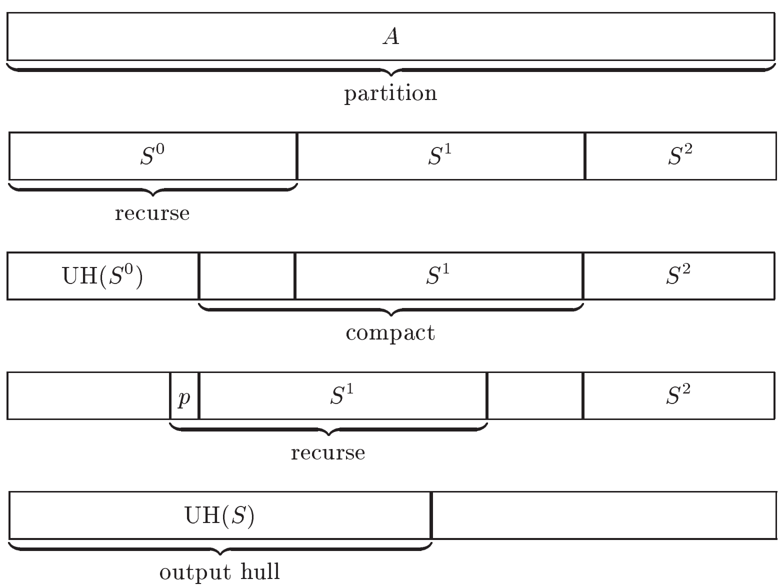


**Fig. 4.** Overview of the CSY-InSitu-Hull algorithm.

Using the technique from Section 2 (Figures 1 and 2), this upper hull algorithm can be made in to a convex hull algorithm with the same running time and memory requirements.

**Theorem 4** *Algorithm* CSY-InSitu-Hull *computes the convex hull of* $n$ *points in* $O(n \log h)$ *time using* $O(\log n)$ *additional storage, where* $h$ *is the number of vertices of the convex hull.*

## 4 An $O(n \log h)$ Time Iterative Algorithm

Next, we give an $O(n \log h)$ time in-place planar convex hull algorithm. We begin with a review of Chan's $O(n \log h)$ time algorithm [3], which is essentially a speedup of Jarvis' March [17].

Chan's algorithm runs in rounds. During $i$th round the algorithm finds the first $g_i = 2^{2^i}$ points on the convex hull. Once $g_i \geq h$ the rounds end as the algorithm detects that it has found all points on the convex hull. During round $i$, the algorithm partitions the input points into $n/g_i$ groups of size $g_i$ and computes the convex hull of each group. The vertices on the convex hull are output in clockwise order beginning with the leftmost vertex. Each successive vertex is obtained by finding tangents from the previous vertex to each of the $n/g_i$ convex hulls. The next vertex is determined, as in Jarvis' March, by choosing the vertex having largest polar angle with respect to the previously found vertex as origin. In the case where the largest polar angle is not unique, ties are broken by taking the farthest vertex from the previously found vertex.

Finding a tangent to an individual convex hull can be done in $O(\log g_i)$ time if the vertices of the convex hull are stored in an array in clockwise order [5,29]. There are $n/g_i$ tangent finding operations per iteration and $g_i$ iterations in round $i$. Therefore, round $i$ takes $O(n \log g_i) = O(n2^i)$ time. Since there are at most $\lceil \log \log h \rceil$ rounds, the total cost of Chan's algorithm is $\sum_{i=1}^{\lceil \log \log h \rceil} O(n2^i) = O(n \log h)$.

Next we show how to implement each round using only $O(1)$ additional storage. Assume for the sake of simplicity that $n$ is a multiple of $g_i$. For the grouping step, we build $n/g_i$ groups of size $g_i$ by taking groups of consecutive elements in $S$ and computing their convex hulls using Graham-InPlace-Hull. Two questions now arise: (1) Once we start the tangent-finding steps, where do we put the convex hull vertices as we find them? (2) In order to find a tangent from a point to a group in $O(\log g_i)$ time we need to know the size of the convex hull of the group. How can we keep track of all these sizes using only $O(1)$ extra memory?

To answer the first question, we store convex hull vertices at the beginning of the array $S$ in the order that we find them. That is, when we find the $k$th vertex on the convex hull, we swap it with $S[k-1]$. We say that a group $G$ is *dirty* if one of its members has been found to be on the convex hull of $S$. A group that is not dirty is *clean*. If a group $G$ is clean, then we can use binary search to find a tangent to $G$ in $O(\log g_i)$ time, otherwise we have to use linear search which takes $O(g_i)$ time. Since after $k$ iterations, the algorithm stores the first

$k$ hull vertices in locations $S[0], \ldots S[k-1]$, the first group consists of elements $S[k], \ldots, S[g_i - 1]$ and is always considered dirty.

To keep track of which other groups are dirty, we mark them by reordering the first two points of the group. In a clean group, the points are stored in lexicographic order. In a dirty group, we store them in reverse lexicographic order. This allows us to test in constant time whether a group is clean or dirty.

To keep track of the size of the convex hull of each clean group without storing the size explicity we use another reordering trick. Let $G[0], \ldots, G[g_i - 1]$ denote the elements of a clean group $G$ and let $<$ denote lexicographic comparison of $(x, y)$ values. We say that the *sign* of $G[j]$ is $+$ if $G[j] < G[j+1]$, and $-$ otherwise. If the convex hull of $G$ contains $h$ vertices, then it follows that the first elements $G[0], \ldots, G[h-2]$ have signs that form a sequence of 1 or more $+$'s followed by 0 or more $-$'s. Furthermore, the elements $G[h], \ldots, G[g_i - 1]$ can be reordered so that the remainder of the signs form an alternating sequence. When we do this, a group element $G[j]$, $0 < j < g_i - 1$, $j \neq h - 1$ is on the convex hull of $G$ if and only if $G[j-1]$, $G[j]$, $G[j+1]$ do not have signs that alternate.

As for $G[0]$, $G[g_i - 1]$ and $G[h-1]$ we know that $G[0]$ is always on the convex hull of $G$. The point $G[g_i - 1]$ is on the convex hull of $G$ if and only if $G[g_i - 2]$ is on the convex hull of $G$ and the triangle $G[g_i - 2], G[g_i - 1], G[0]$ is oriented clockwise.[2] The point $G[h-1]$ is on the convex hull of $G$ if and only if $G[h-2]$ is on the convex hull of $G$ and the triangle $G[h-2], G[h-1], G[0]$ is oriented clockwise. Therefore, for any index $0 \leq j < g_i$, we can test if $G[j]$ is on the convex hull of $G$ in constant time. Using this in conjunction with binary search, we can compute the number of vertices on the convex hull of $G$ in $O(\log g_i)$ time. Thus, we can compute the size of the convex hull of $G$ and find a tangent in $O(\log g_i)$ time, as required.

We have provided all the tools for an in-place implementation of Chan's algorithm. Except for the extra cost of finding tangents in dirty groups, the running time of this implementation is asymptotically the same as that of the original algorithm, so we need only bound this extra cost of finding tangents in dirty groups. During one step of round $i$, we find tangents of at most $g_i$ dirty groups, at a cost of $O(g_i)$ per group, and there are $g_i$ steps in round $i$. Therefore, the total cost of searching dirty groups during round $i$ is $O(g_i^3) \subseteq O(n)$ for all $g_i \leq n^{1/3}$. Therefore, the total cost of round $i$ is $O(g_i^3 + n \log g_i) \subseteq O(n \log g_i)$ for any $h < n^{1/3}$. Since we can abort the algorithm when $g_i \geq n^{1/3}$ and use Graham-InPlace-Hull, the overall running time of the algorithm is again $O(n \log h)$.

**Theorem 5** *The above algorithm,* Chan-InPlace-Hull, *computes the convex hull of $n$ points in $O(n \log h)$ time using $O(1)$ additional storage, where $h$ is the number of vertices of the convex hull.*

The constants in Chan-InPlace-Hull can be improved using the following trick that is mentioned by Chan [3]. When round $i$ terminates without finding the entire convex hull, the $g_i$ convex hull points that were computed should not

[2] We use the convention that three collinear points are *not* oriented clockwise.

be discarded. Instead, the grouping in round $i+1$ is done on the remaining $n-g_i$ points, thus eliminating the need to recompute the first $g_i$ convex hull vertices. This optimization works perfectly when applied to Chan-InPlace-Hull since the first $g_i$ convex hull points are already stored at locations $S[0], \ldots, S[g_i - 1]$.

## 5 Conclusions

We have given three algorithms for computing the convex hull of a planar point set. The first algorithm is in-place and runs in $O(n \log n)$ time. The second algorithm is *in situ* and runs in $O(n \log h)$ time. The third algorithm is in-place and and runs in $O(n \log h)$ time. The first two algorithms are reasonably simple and implementable. In order to facilitate comparisons with other convex hull implementations, our source code is available for download [25].

This paper came to be when two separate groups of researchers (authors 1–3 and authors 4–6) discovered they were both working on in-place computational geometry algorithms and decided to merge their results. Some of these results have been omitted due to space constraints. These include in-place or *in situ* implementations of Eddy's algorithm (also known as quickhull) [11], Kirkpatrick and Seidel's algorithm [21], Seidel's randomized linear programming algorithm [31] and Megiddo's deterministic linear programming algorithm [24].

The ideas presented in this paper also apply to other problems. The *maximal elements* problem is that of determining all elements $S[i]$ such that $S[j] < S[i]$ for all $0 \leq j < n$. An algorithm almost identical to Graham's scan can be used to solve the maximal elements problems in $O(n \log n)$ time, and this can easily be implemented in-place. Furthermore, an in-place algorithm almost identical to that in Section 4 can be used to solve the maximal elements problem in $O(n \log h)$ time, where $h$ is the number of maximal elements.

The question of *in situ* and in-place algorithms for maximal elements and convex hulls in dimensions $d \geq 3$ is still open. In order for this question to make sense, we ask only that the algorithm identify which input points are maximal or on the convex hull. Testing whether a given point is maximal can be done in $O(dn)$ time using the definition of maximality. Testing whether a single point is on the convex hull is a $d-1$ dimensional linear programming problem that can be solved in-place in $O(d!n)$ expected time using Seidel's algorithm [31]. Thus, the maximal elements problem can be solved in $O(dn^2)$ time and the convex hull problem can be solved in $O(d!n^2)$ time using in-place algorithms. Are there algorithms with reduced dependence on $n$?

More generally, one might ask what other computational geometry problems admit in-place or *in situ* algorithms. Some problems that immediately come to mind are those of computing $k$-piercings of sets, finding maximum cliques in intersection graphs, computing largest empty disks, computing smallest enclosing disks, and finding ham-sandwich cuts.

## References


1. A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979. Corrigendum, *Information Processing Letters*, 10:168, 1980.
2. B. K. Bhattacharya and S. Sen. On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *Journal of Algorithms*, 25(1):177–193, 1997.
3. T. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.
4. T. Chan, J. Snoeyink, and C. K. Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional Voronoi diagrams. *Discrete & Computational Geometry*, 18:433–454, 1997.
5. B. Chazelle and D. P. Dobkin. Intersection of convex objects in 2 and 3 dimensions. *Journal of the ACM*, 34:1–27, 1987.
6. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1988.
7. E. W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3):223–233, 1982.
8. E. W. Dijkstra and A. J. M. van Gasteren. An introduction to three algorithms for sorting in situ. *Information Processing Letters*, 15(3):129–134, 1982.
9. S. Dvorak and B. Durian. Stable linear time sublinear space merging. *The Computer Journal*, 30(4):372–375, 1987.
10. S. Dvorak and B. Durian. Unstable linear time O(1) space merging. *The Computer Journal*, 31(3):279–282, 1988.
11. W. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, 3(4):398–403, 1977.
12. R. W. Floyd. Algorithm 245, Treesort 3. *Communications of the ACM*, 7:401, 1964.
13. R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.
14. E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms.* Computer Science Press, 1998.
15. B.-C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, 1988.
16. B.-C. Huang and M. A. Langston. Fast stable merging and sorting in constant extra space. *The Computer Journal*, 35(6):643–650, 1992.
17. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.
18. J. Katajainen, T. Pasanen, and J. Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1996.
19. J. Katajainen and T. A. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32(4):580–585, 1992.
20. J. Katajainen and T. A. Pasanen. In-place sorting with fewer moves. *Information Processing Letters*, 70(1):31–37, 1999.
21. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.
22. T. W. Lai and D. Wood. Implicit selection. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 14–23. Springer-Verlag, 1988.

23. H. Mannila and E. Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18(4):203–208, 1984.
24. N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31(1):114–127, 1984.
25. P. Morin. insitu.tgz. Available online at http://cgm.cs.mcgill.ca/~morin/, 2001.
26. J. I. Munro, V. Raman, and J. S. Salowe. Stable in situ sorting and minimum data movement. *BIT*, 30(2):220–234, 1990.
27. F. P. Preparata. An optimal real time algorithm for planar convex hulls. *Communications of the ACM*, 22:402–405, 1979.
28. F. P. Preparata and S. J. Hong. Convex hulls of finite point sets in two and three dimensions. *Communications of the ACM*, 2(20):87–93, 1977.
29. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.
30. J. Salowe and W. Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, 1987.
31. R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.
32. M. I. Shamos. *Computational Geometry*. PhD thesis, Yale University, 1978.
33. H. W. Six and L. Wegner. Sorting a random access file in situ. *The Computer Journal*, 27(3):270–275, 1984.
34. A. Symvonis. Optimal stable merging. *The Computer Journal*, 38(8):681–690, 1995.
35. R. Wenger. Randomized quick hull. *Algorithmica*, 17:322–329, 1997.
36. J. W. J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, 7:347–348, 1964.