# Multi-Pass Geometric Algorithms*

Timothy M. Chan[†] Eric Y. Chen[‡]

February 20, 2006

**Abstract**

We propose the study of exact geometric algorithms that require limited storage and make only a small number of passes over the input. Fundamental problems such as low-dimensional linear programming and convex hulls are considered.

## 1 Introduction

**The multi-pass model.** *Streaming* algorithms that make a single pass over the input and work with a small amount of space have grown in popularity [31], because of the ability of such algorithms to handle massive data sets. Since only one pass over the input is required, data elements may arrive one at a time and the entire data set never needs to be physically stored. Study of geometric algorithms in the data-stream model has already begun to take place in several recent papers (e.g., [3, 10, 37]).

In this paper, we examine a more powerful *multi-pass* model, where algorithms are allowed to make multiple passes over the input. The input remains unchanged after each pass, and depending on the problem, the answer may be sent to a write-only output stream. The goal is to minimize the amount of working space (measured in words, in this paper), while keeping the number of passes small. As usual, we would like to bound the total running time as well (which includes the cost of scanning and is at least the input size times the number of passes); for this purpose, we assume unit-cost random access for the working space (but not the input, of course).

Unlike in the one-pass streaming model, the motivation here is on applications where the data set is explicitly stored somewhere (for example, in tapes or disks). In light of recent interest in algorithmics for large data sets, the restriction that input elements are read sequentially in few passes is attractive because of the lower I/O overhead. In this paper, we investigate such multi-pass algorithms in computational geometry. What we discover is that although in the one-pass model one cannot usually obtain exact algorithms and must turn towards approximation algorithms, in the multi-pass model there are interesting (and important) geometric problems that can be solved *exactly*.

Multi-pass algorithms have only been briefly touched upon in some of the previous streaming papers in geometry (e.g., [2, 37]), but have gained more attention recently in other areas (e.g.,

*A preliminary version of this work has appeared in *Proc. 21st ACM Sympos. Comput. Geom.*, pages 180–189, 2005.
[†]School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, tmchan@uwaterloo.ca.
Research of this author has been supported by an NSERC Research Grant and a Premier's Research Excellence Award.
[‡]School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, y28chen@uwaterloo.ca.

[4, 16, 19, 20, 23]). The history of multi-pass algorithms can be traced back to much earlier times (in compiler design and automata theory). Munro and Paterson's seminal paper [30] in 1980 defined the model and studied the classical sorting and selection problems. In particular, they gave a selection algorithm that requires only $\lceil 1/\delta \rceil$ passes and $O(n^\delta \log^{2-2\delta} n)$ space for an arbitrary fixed constant $\delta > 0$; they also provided an almost matching lower bound. The present paper can be seen as a (belated) continuation of Munro and Paterson's work for sorting and searching problems in dimension beyond one.

**Our main results.** We naturally begin our study with some of the most fundamental problems in computational geometry. We obtain a large number of results on multi-pass algorithms, the highlights of which include:

- an $O(n)$-time randomized algorithm for linear programming (or any LP-type problem) in fixed dimensions, using a constant number of passes and $O(n^\delta)$ space for any fixed constant $\delta > 0$;
- an $O(n)$-time algorithm for computing the convex hull of $n$ sorted points in 2-d, using a constant number of passes and $O(n^{1/2+\delta})$ space;
- an $O(n \log n)$-time algorithm for computing the convex hull of $n$ arbitrary points in 2-d, using a constant number of passes and $O(hn^\delta)$ space, where $h$ denotes the output size.

We also provide nearly matching lower bounds to the above. (A more detailed table of results is given near the end of the paper.) Along the way, we identify a number of useful techniques for the multi-pass model (for example, using larger branching factors, running a multi-pass algorithm on a subset of the input, running several algorithms in series or in parallel, proving multi-pass lower bounds by adversary and information-theoretic arguments, etc.)

As an example, consider the problem of finding a line that minimizes the largest vertical distance to a collection of $n$ data points in the plane [34]. Results in the data-stream model [10] imply a one-pass algorithm that can approximate the minimum to within a $1+\varepsilon$ factor for any fixed constant $\varepsilon > 0$ with constant space. In contrast, our result on linear programming implies an $O(1)$-pass algorithm that can find the exact minimum with $O(n^\delta)$ space.

**A related read-only model.** Our interest in multi-pass algorithms actually stems from our earlier interest in *in-place* (or nearly in-place) algorithms [7], which are space-efficient algorithms for input that resides in an array. Such algorithms are allowed to permute entries of the array (and sometimes overwrite entries). Multi-pass algorithms are more restrictive (and thus in some ways more desirable) than in-place algorithms: not only do we lack random access to the input, but the input space can only be read and not written (so the original order of the data is retained during and after the execution of the algorithm). In-place algorithms for many geometric problems [7] also tend to be complicated, lack locality of reference, and are thus less practical. In contrast, multi-pass algorithms are automatically I/O-efficient (and even cache-oblivious), if the amount of working space is sufficiently small.

For a class of algorithms that is in "between" multi-pass and in-place algorithms, we also consider the *read-only* model, where we are allowed read-only random access to the input, and we are concerned no longer with the number of passes, just running time (as measured in the standard RAM) and the amount of extra working space.

To contrast with our multi-pass results, we give:

- a read-only $O(n)$-time randomized algorithm for linear programming in fixed dimensions, using $O(\log n)$ space;
- a read-only $O(n)$-time randomized algorithm for finding the convex hull of $n$ sorted points in $\mathbb{R}^2$, using $O(n^\delta)$ space for any fixed $\delta > 0$.

As an example, our linear programming result implies that there is a nearly in-place algorithm for testing whether a simple polygon is star-shaped (and if so, reporting a point in the kernel) [34], where the vertices are stored in an array in order. Previous in-place algorithms for linear programming [6] cannot be used, because permuting vertices would destroy the polygon.

Read-only algorithms have been previously considered for the sorting problem [5, 33] but are apparently less often studied. The main advantage is that if the user needs the input to be left in its original state, we do not need to duplicate the input first before sending it to the algorithm. (Duplicating the input would of course defeat the purpose of space-efficiency.) Another potential advantage is that the input does not need to reside in one place, as long as we can answer queries to access any individual element (in a way, this is similar to models proposed for *sublinear* algorithms [12], except that we are happy with linear running times). The read-only restriction can also be combined with external-memory models.

## 2 2-d Convex Hulls

We warm up with the most basic geometric problem, 2-d convex hulls. As Munro and Paterson [30] observed, there is a simple algorithm for sorting $n$ numbers using $O(s)$ space, $O(n/s)$ passes, and $O(n(\log n + n/s))$ total number of comparisons: the algorithm simply finds the next $s$ smallest elements in each pass. (Throughout this paper, space is measured in terms of the number of words, not bits.) These bounds were shown to be optimal asymptotically. Since 1-d sorting reduces to 2-d convex hulls, we cannot hope for a better result for 2-d convex hulls. Still, we show that the same result can be attained in 2-d by a similarly simple algorithm. It suffices to consider just the upper hull:

**Theorem 2.1** *Given $n$ points in $\mathbb{R}^2$ and a parameter $s$, we can generate the vertices of the upper hull from left to right (and print to an output stream), by an $O(n/s)$-pass algorithm that uses $O(s)$ space and runs in $O(n(\log n + n/s))$ time.*

**Proof:** We first present the pseudocode as a normal algorithm and afterwards consider its efficiency as a multi-pass algorithm.

$v =$ the leftmost point
while $v \neq$ the rightmost point:
  find a vertical slab $\sigma$ containing $s$ points (if possible) with its left wall through $v$
  let $\langle q_0, \ldots, q_j \rangle$ be the upper hull of the points inside $\sigma$
  for each point $p$ to the right of $\sigma$:
    while $p$ is above $\overleftrightarrow{q_{j-1}q_j}$, $j = j - 1$
    set $j = j + 1$ and $q_j = p$
  print out $q_0, \ldots, q_j$ and set $v = q_j$

We prove correctness of the pseudocode. In each iteration of the main loop (lines 1–7), we start with a known hull vertex $v$ and compute the portion of the upper hull from $v$ up to and including the bridge (hull edge) at the right wall of $\sigma$. This is accomplished by imitating Graham's scan [34] (lines 4–6), except that points are processed not necessarily in sorted order (although points inside $\sigma$ are examined in line 3 before points to the right of $\sigma$). It is straightforward to verify (for example, by induction) that at the end of the for loop, $q_{j-1}q_j$ is indeed the bridge at the right wall of $\sigma$. Thus, after at most $\lceil n/s \rceil$ iterations, the entire upper hull is computed.

We now analyze the pseudocode as a multi-pass algorithm. Line 2 reduces to finding the $s$ leftmost points among those points to the right of $v$, and can be carried out in one pass, since we can easily maintain the $s$ smallest elements of a stream with $O(s)$ space. The running time is linear if we maintain the $s$ smallest elements by the following method: read in the next $s$ elements, insert them to the current buffer (but don't sort the buffer), select the median in the buffer in $O(s)$ time, remove the $s$ largest elements from the buffer, and repeat. Line 3 takes $O(s \log s)$ time and $O(s)$ space in main memory. Lines 4–6 require an additional pass and takes $O(n)$ time. The whole algorithm thus can be implemented with at most $2\lceil n/s \rceil$ passes, $O(s)$ space, and $O((n/s) \cdot (n + s \log s))$ running time. □

Even in the more relaxed read-only model, no algorithm with substantially better time-space product is possible, due to known lower bounds for sorting [5]. The situation is unfortunate, as many geometric problems (e.g., closest pair and diameter) are at least as hard as sorting-like problems (e.g., element uniqueness) and probably do not admit space-efficient algorithms with few passes. Still, we will identify some geometric problems not threatened by the sorting lower bound in the next sections.

# 3 Fixed-Dimensional Linear Programming

We consider linear programming in fixed dimensions, where a number of (deterministic and randomized) linear-time algorithms were known in the traditional model of computation. We show that many of these algorithms can be modified to work with little space ($O(n^\delta)$) using a constant number of passes, although some are more time-efficient than others. Some of the modifications are not too difficult, while some require new ideas. We also prove a nontrivial lower bound and contrast it with a read-only algorithm. (In the bounds below, we suppress dependence on $d$ but not $\delta$.)

## 3.1 Prune-and-search in 2-d

We begin in 2-d with the first published linear-time algorithm by Megiddo [26] and Dyer [17], based on prune-and-search. There are two difficulties in turning this algorithm into a multi-pass algorithm: First, since the original algorithm removes a constant fraction of the input at each iteration and consequently requires at least a logarithmic number of passes, we need to remove a larger fraction. This is accomplished by changing some parameters (instead of pairing, we use grouping of a larger size). Second, since we cannot explicitly remove any element from the input in the multi-pass model, we need to encode the current subset of surviving elements using a small amount of information and be able to retrieve this subset in a single pass whenever required. This task is accomplished by describing the current subset as the outcome of a series of processes ("filters") and retrieving the subset using a nontrivial "pipelining" technique.

**Theorem 3.1** *Given $n$ halfplanes in $\mathbb{R}^2$, we can compute the lowest point in their intersection by an $O(1/\delta)$-pass deterministic algorithm that uses $O((1/\delta^2)n^\delta)$ space and runs in $O((1/\delta)n^{1+\delta})$ time.*

**Proof:** We first define two subroutines: given a stream of halfplanes $H$, a parameter $r$, and a vertical slab $\sigma$, $\textsc{List}_{r,\sigma}(H)$ outputs a stream of vertical lines, and $\textsc{Filter}_{r,\sigma}(H)$ outputs a subset of halfplanes. (Throughout this paper, "slabs" are implicitly vertical slabs.)

$\textsc{List}_{r,\sigma}(H)$:
0. while there are remaining halfplanes in $H$:
1. read in the next $r$ halfplanes $h_1, \ldots, h_r$
2. compute the intersection $I = h_1 \cap \cdots \cap h_r$
3. print out the vertical lines through the vertices of $I$ inside $\sigma$

$\textsc{Filter}_{r,\sigma}(H)$:
0. while there are remaining halfplanes in $H$:
1. read in the next $r$ halfplanes $h_1, \ldots, h_r$
2. compute the intersection $I = h_1 \cap \cdots \cap h_r$
3. print out the halfplanes that are involved in the boundary of $I \cap \sigma$

For an input stream of size $n$, both subroutines are one-pass algorithms and require $O(r)$ space and $O((n/r) \cdot r \log r) = O(n \log r)$ time.

Given a set $H$ of $n$ halfplanes, our algorithm follows the outline below, where $r_0, r_1, \ldots$ is a sequence to be determined later. Here, we maintain the invariant that at each iteration $i$, the solution lies in a vertical slab $\sigma_i$ and is defined only by halfplanes in a subset $H_i$.

0. let $\sigma_0$ be the whole plane and $H_0 = H$
1. for $i = 0, 1, \ldots$:
2. if $|H_i|$ is below a constant then return solution for $H_i$ directly
3. divide the slab $\sigma_i$ into $r_i$ subslabs so that each subslab contains $O(1/r_i)$-th of the lines from $\textsc{List}_{r_i,\sigma_i}(H_i)$
4. decide which subslab contains the solution, and let this subslab be $\sigma_{i+1}$
5. $H_{i+1} = \textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$

We first prove correctness. Assume that the invariant holds at iteration $i$. After iteration $i$, since we know that the solution lies in $\sigma_{i+1}$, by design of $\textsc{Filter}$, only halfplanes in $H_{i+1} = \textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$ can indeed affect the solution.

Let $n_i = |H_i|$. We now show that $n_i$ decreases rapidly. Since $\textsc{List}_{r_i,\sigma_i}(H_i)$ generates at most $n_i$ lines, at most $O(n_i/r_i)$ of these lines are inside $\sigma_{i+1}$. Observe that if $h_1 \cap \cdots \cap h_{r_i}$ has $j$ vertices inside $\sigma_{i+1}$, then at most $j+2$ halfplanes are involved in the boundary of $h_1 \cap \cdots \cap h_{r_i} \cap \sigma_{i+1}$. Thus, $\textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$ generates at most $J + 2\lceil n_i/r_i \rceil$ halfplanes, where $J$ is the total number of vertices inside $\sigma_{i+1}$. As $J = O(n_i/r_i)$, we have $n_{i+1} = O(n_i/r_i)$.

One issue needs to be addressed: in the multi-pass setting, we do not have space to store the subset $H_i$ explicitly. Instead, during each iteration, we need to re-generate $H_i$ from the original input $H$, by re-running the $\textsc{Filter}$ operations from scratch. In other words, we work with the following modified pseudocode:

0′. let $\sigma_0$ be the whole plane
1′. for $i = 0, 1, \ldots$:
2′. if the size of $\text{Filter}_{r_{i-1},\sigma_i}(\cdots(\text{Filter}_{r_0,\sigma_1}(H))\cdots))$ is below a constant then return solution for $\text{Filter}_{r_{i-1},\sigma_i}(\cdots(\text{Filter}_{r_0,\sigma_1}(H))\cdots))$ directly
3′. divide the slab $\sigma_i$ into $r_i$ subslabs so that each subslab contains $O(1/r_i)$-th of the lines from $\text{List}_{r_i,\sigma_i}(\text{Filter}_{r_{i-1},\sigma_i}(\cdots(\text{Filter}_{r_0,\sigma_1}(H))\cdots))$
4′. decide which subslab contains the solution, and let this subslab be $\sigma_{i+1}$

We briefly observe how in general one may combine a series of one-pass processes $P_1, \ldots, P_i$ (a *pipeline*) into a single one-pass algorithm in a space-efficient manner. Each process $P_i$ $(i > 1)$ has a buffer that $P_i$ reads from (and that $P_{i-1}$ writes into); for theoretical purposes, we may take the buffer size to be 1. During each cycle, if none of the buffers are full, we can execute the next step of $P_1$. Otherwise, we identify the process $P_j$ with the largest $j$ whose buffer is full and execute the next step of $P_j$. (In this step, $P_j$ may read from its buffer or write to $P_{j+1}$'s buffer, but we know that $P_j$'s buffer is not empty and $P_{j+1}$'s buffer is not full.) The space required for the pipeline is thus bounded by the sum of the space required for the individual processes.

We can now analyze the cost of iteration $i$ of our algorithm. The Filter pipeline requires $O(r_0 + \cdots + r_{i-1})$ space and $O(n_0 \log r_0 + \cdots + n_{i-1} \log r_{i-1})$ time. For line 3′, we can pipe this output stream through List and then through an algorithm for finding $r_i$ *approximate quantiles* in a stream of $n_i$ elements—i.e., finding elements of ranks within an additive error $O(n_i/r_i)$ from $n_i/r_i$, $2n_i/r_i$, $3n_i/r_i$, ... Munro and Paterson [30] provided a simple tree-based algorithm for this task that requires one pass and $O(r_i \log^2 n_i)$ space; it can be checked that the running time is $O(n_i \log(r_i \log n_i))$. (Munro and Paterson did not explicitly state the approximate quantiles problem in their paper but this subroutine was used as part of their exact, multi-pass selection algorithm; see [21] for another algorithm.)

For line 4′, recall [17, 26] that deciding whether the solution is to the left or right of a line $\ell$ reduces to computing the intersection of the halfplanes at $\ell$, which reduces to computing the minimum or maximum of a 1-d set. Thus, we can decide which of the $O(r_i)$ subslabs contains the solution in one pass, by maintaining $O(r_i)$ minima/maxima simultaneously, with $O(r_i)$ space and $O(nr_i)$ time.

The simplest choice of parameters is $r_0 = r_1 = \cdots = r$, with $O(\log_r n)$ iterations. The algorithm thus makes $O(\log_r n)$ passes, uses $O((1/\delta)r + r \log^2 n)$ space, and runs in $O(nr \log_r n)$ time. The theorem follows, for example, by setting $r = n^{\delta/2}$ (and noting that $\log n = O((1/\delta)n^{\delta/4})$). □

We can speed up the algorithm to run in *almost* linear time: this theorem gives our best deterministic result. (A faster randomized algorithm will be given later.)

**Theorem 3.2** *The running time in Theorem 3.1 can be improved to $O((1/\delta)n \log^{(c)} n)$, where $c$ is any fixed integer constant and $\log^{(c)}$ denotes logarithm iterated $c$ times.*

**Proof:** The algorithm is the same, but line 3′ (approximate quantiles) and line 4′ (the decision step) need to be done more efficiently, and the choice of parameters is different.

For line 3′, we use a variant of Munro and Paterson's algorithm where the tree has degree $b$ instead of 2. It is straightforward to check (see [30]) that the space bound becomes $O(br_i \log_b^2 n_i)$ and the time bound becomes $O(n_i \log(r_i \log_b n_i))$. We set $b = n^{\delta/2}$.

For line 4′, recall that this step reduces to computing the intersection of $n$ halfplanes at each of $O(r_i)$ vertical lines. We speed up the naive $O(nr_i)$-time approach by the following method: read in the next $r_i$ halfplanes, compute their intersection $I$ in $O(r_i \log r_i)$ time [34], compute the intersection of $I$ with the $O(r_i)$ vertical lines in an additional $O(r_i \log r_i)$ time (by binary searches), update the current answers at the $O(r_i)$ vertical lines, and repeat. The space bound is still $O(r_i)$, but the running time is improved to $O((n/r_i) \cdot r_i \log r_i) = O(n \log r_i)$.

This time bound can be further improved by applying the decision step only to halfplanes in $H_i = \textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots)$. Thus, line 4′ takes only $O(n_i \log r_i)$ time, in addition to the cost of re-executing the FILTER pipeline in a new pass.

We choose $r_0 = \log^{(c-1)} n$, $r_1 = \log^{(c-2)} n$, $\ldots$, $r_{c-2} = \log n$, and $r_{c-1} = r_c = \cdots = n^{\delta/2}$, with at most $c - 1 + \lceil 2/\delta \rceil$ iterations. Since $n_{j+1} = O(n_j/r_j)$ for all $j$, the running time of iteration $i$ is now

$$O(n_0 \log r_0 + \cdots + n_i \log r_i) \;=\; O\left(n \log r_0 + \sum_{i=1}^{c-1} \frac{n}{r_{i-1}} \log r_i\right) \;=\; O(n \log^{(c)} n).$$

The upper bounds on the number of passes and space are unchanged. □

We will need the following generalization for later applications:

**Theorem 3.3** *Given $n$ halfplanes in $\mathbb{R}^2$ and $q$ directions, we can compute the $q$ extreme points along the given directions in the intersection of the halfplanes by an $O(1/\delta)$-pass deterministic algorithm that uses $O((1/\delta^2)qn^\delta)$ space and runs in $O((1/\delta)(n \log^{(c)} n + n \log q))$ time.*

**Proof:** We modify the algorithm so that $\sigma_i$ is not a single slab but the union of up to $q$ (vertical) slabs. In line 3′, $\sigma_i$ is divided into up to $q + r_i$ subslabs. In line 4′, $\sigma_{i+1}$ is set to be the union of the subslabs containing the $q$ solutions. In the analysis, we now have $n_{i+1} = O(qn_i/r_i)$, so we need to increase all the $r_i$'s by a factor of $q$ to keep the same number of iterations. □

### 3.2 Prune-and-search in higher dimensions

Megiddo [27] extended his algorithm to higher dimensions, but the original algorithm seems difficult to work with in the multi-pass setting (among other things, it requires pairing of *nonadjacent* input elements). Nevertheless, with more modern tools, namely, *cuttings*, we can obtain a multi-pass variant of the prune-and-search algorithm. In fact, this version is more straightforward, because the current subset of surviving elements can be encoded by just a simplex and can be retrieved easily without any pipelining tricks. One helpful technique, of running several multi-pass algorithms "simultaneously", is illustrated.

**Theorem 3.4** *Given $n$ halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an $O(1/\delta^{d-1})$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O((1/\delta^{O(1)})n^{1+\delta})$ time* w.h.p.*—i.e., with probability at least $1 - 1/n^c$ for any fixed constant $c$.*

**Proof:** Let $H$ be the set of $n$ bounding hyperplanes. The outline of the algorithm is simple:

0. let $\Delta$ be the whole space
1. repeat:
2.     if $|\{h \in H \mid h \text{ intersects } \Delta\}| \le r \log n$ then
        return the solution for $\{h \in H \mid h \text{ intersects } \Delta\}$ directly
3.     take a random sample $R$ of expected size $r \log n$ from $\{h \in H \mid h \text{ intersects } \Delta\}$
4.     compute a triangulation $T$ of the arrangement of $R$ restricted inside $\Delta$
5.     decide which simplex in $T$ contains the solution, and set $\Delta$ to be this simplex

Correctness is self-evident, since the solution is always contained in $\Delta$ and is defined only by hyperplanes that intersect $\Delta$.

Let $\Delta_i$ be the simplex $\Delta$ at the beginning of the $i$-th iteration, let $H_i = \{h \in H \mid h \text{ intersects } \Delta_i\}$, and let $n_i = |H_i|$. It is well-known [13, 22, 29] that for a random sample $R \subseteq H_i$ of size $r \log n$, w.h.p. a triangulation of the arrangement of $R$ forms an $O(1/r)$-*cutting* of $H_i$—i.e., a partition of $\mathbb{R}^d$ into simplices such that each simplex intersects at most $O(n_i/r)$ hyperplanes of $H_i$. Since all hyperplanes intersecting $\Delta_{i+1}$ must intersect $\Delta_i$, we have $n_{i+1} = O(n_i/r)$ w.h.p. The number of iterations is thus $O(\log_r n)$ w.h.p.

We can compute $n_i$ (line 2) easily in one pass. Line 3 can also be easily done in another pass by Bernoulli sampling (for each hyperplane, if it intersects $\Delta_i$, put it into $R$ with probability $(r \log n)/n_i$). In line 4, a triangulation of size $O((r \log n)^d)$ can be found in $O((r \log n)^d)$ time and space in main memory.

For line 5, recall [27] that deciding which side of a given hyperplane $h$ contains a solution reduces to solving a linear program restricted in $h$. Thus, line 5 can be done by solving $O((r \log n)^d)$ subproblems in $d-1$ dimensions. We handle these subproblems by making $O((r \log n)^d)$ calls to a $(d-1)$-dimensional algorithm not in series but in parallel. In other words, in every pass, we execute one pass of all invocations of the algorithms simultaneously; the space usage is multiplied by $(r \log n)^{O(1)}$ but not the number of passes.

Let $P_d(n)$, $S_d(n)$, and $T_d(n)$ be the number of passes, the space requirement, and the running time of the $d$-dimensional algorithm. Then

$$\begin{array}{lll}
P_d(n) = O((\log_r n) P_{d-1}(n)), & P_1(n) = 1 & \Longrightarrow \; P_d(n) = O(\log_r^{d-1} n) \\
S_d(n) = O((r \log n)^{O(1)} S_{d-1}(n)), & S_1(n) = O(1) & \Longrightarrow \; S_d(n) = O((r \log n)^{O(1)}) \\
T_d(n) = O(((r \log n)^{O(1)} (\log_r n) T_{d-1}(n)), & T_1(n) = O(n) & \Longrightarrow \; T_d(n) = O(n r^{O(1)} \log^{O(1)} n).
\end{array}$$

The theorem follows by setting $r = n^{\Theta(\delta)}$ (and readjusting $\delta$ by a constant factor). □

We remark that some of the polylogarithmic factors may be improved, although these are hidden under the $n^\delta$ factors. For example, the size of the triangulation $T$ can be reduced to $O((r \log n)^{\lfloor d/2 \rfloor})$, since only one cell of the arrangement needs to be triangulated. Also, the extra $\log n$ factor in the sample size can be avoided via known techniques on cuttings [11, 29], if we abandon high-probability bounds for expected bounds. More significantly, as one referee pointed out, the number of calls to the $(d-1)$-dimensional algorithm in line 5 could be greatly reduced (we think, to $O((r \log n)^2)$) while increasing the number of passes by only a constant factor, by following an approach from the prune-and-search linear programming algorithm by Dyer and Frieze [18].

The above algorithm can be derandomized, luckily, because of a recent result on derandomization for data streams. The running time is not as good as our earlier 2-d result, however:

**Theorem 3.5** *The algorithm in Theorem 3.4 can be made deterministic with the same performance.*

**Proof:** We replace the random sample $R$ (line 3) with a $(1/r)$-*approximation* of $H_i$ in a suitable range space [25]. Bagchi *et al.* [3] have recently described a tree-based algorithm for computing a $(1/r)$-approximation of size $O(r^2 \log r)$ that requires one pass, uses $O(r^{O(1)} \log^{O(1)} n)$ space, and runs in $O(n r^{O(1)} \log^{O(1)} n)$ time.

The triangulation $T$ (line 4) still forms an $O(1/r)$-cutting of $H_i$, although its size is now $O((r^2 \log r)^d)$. (Alternatively, we can replace $T$ with an $O(1/r)$-cutting of $R$, which has size $O(r^d)$ and can be computed by an internal-memory algorithm.) Again we can set $r = n^{\Theta(\delta)}$. □

The same approach can be applied to the *ham-sandwich cut* problem in the 2-d separable case [28]. (The decision step here requires an exact multi-pass algorithm for selection, which was provided by Munro and Paterson [30].)

**Theorem 3.6** *Given two $n$-point sets $A$ and $B$ that are separable by a line in $\mathbb{R}^2$, we can find a line $\ell$ such that each side of $\ell$ contains $\lfloor n/2 \rfloor$ points of $A$ and $\lfloor n/2 \rfloor$ points of $B$, by an $O(1/\delta^2)$-pass algorithm using $O((1/\delta)^{O(1)} n^\delta)$ space.*

## 3.3 Clarkson's algorithm

Another linear programming algorithm that can be made to work in the multi-pass model is Clarkson's randomized algorithm [14]. His approach, without any major modification, already yields a result with few passes and sublinear space.

**Theorem 3.7** *Given $n$ halfspaces in $\mathbb{R}^d$ whose intersection is nonempty, we can compute the lowest point in the intersection by a $(d+1)$-pass Las Vegas algorithm that uses $O(\sqrt{n \log n})$ space and runs in $O(n)$ time w.h.p.*

**Proof:** Let $H$ be the given set of halfspaces. The nonrecursive version of Clarkson's algorithm can be paraphrased as follows:

0. take a random sample $R$ of expected size $r \log n$
1. for $i = 0, \ldots, d$:
2.     let $v_i$ be the solution for $\{h \in H \mid h \in R \text{ or } \exists j < i,\ v_j \text{ violates } h\}$
3. return $v_d$

Let $H_i = \{h \in H \mid h \in R \text{ or } \exists j < i,\ v_j \text{ violates } h\}$. It can be shown [14] that at least $i$ of the halfspaces defining the optimal solution is in $H_i$, and thus $v_d$ is indeed the optimal solution.

It is well-known [29] that w.h.p., a random sample $R$ of size $r \log n$ is an $O(1/r)$-*net*—a subset $R \subseteq H$ with the property that any point violating no halfspaces in $R$ violates at most $O(n/r)$ halfspaces in $H$. Thus, $|H_i| = r \log n + O(n/r)$ w.h.p.

Line 0 can be easily done in one pass. In each iteration of line 1, the subset $H_i$ can be found in one pass, and $v_i$ can be computed by an internal-memory linear programming algorithm in $O(|H_i|)$ time and space. Note that iteration 0 does not require a new pass (since $H_0 = R$). The theorem follows by setting $r = \sqrt{n/\log n}$. (Note that if the problem could be infeasible, an additional pass is required to verify the solution.) □

We can combine Clarkson's algorithm with our previous prune-and-search method and thereby improve the expected running time of Theorem 3.4 to linear:

**Theorem 3.8** *Given $n$ halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an $O(1/\delta^{d-1})$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O((1/\delta^{O(1)})n)$ time w.h.p.*

**Proof:** We use the same algorithm from the proof of Theorem 3.7, except that line 2 is now done by feeding $H_i$ into the algorithm in Theorem 3.4. In other words, in every pass, we only read in a halfspace if it is in $R$ or is violated by $v_j$ for some $j < i$. (Note that $H_i$ cannot be explicitly stored.) Compared to Theorem 3.4, the number of passes is increased by a $d+1$ factor, and the space requirement is increased by an $O(r \log n)$ term. The expected running time becomes $O((1/\delta^{d-1})n + (1/\delta^{O(1)})(r \log n + n/r)^{1+\delta})$. The theorem follows by setting $r = n^\delta$. $\square$

We can also obtain a multi-pass algorithm purely from Clarkson's recursive algorithm. Here, the intermediate subsets are a little harder to describe but still do not require pipelining.

**Theorem 3.9** *Given $n$ halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by a $2^{O(1/\delta)}$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O(2^{O(1/\delta)}n)$ time w.h.p.*

**Proof:** The recursive version of Clarkson's algorithm can be rewritten as follows, where the argument $\mathcal{C}$ is a small collection of pairs of the form $(R, V)$ with a subset $R \subseteq H$ and a set $V$ of at most $d+1$ points. (In the initial call, $\mathcal{C} = \emptyset$.)

Clarkson($\mathcal{C}$):
0. if $|\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}| \leq 2r \log n$ then
   return the solution for $\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$ directly
1. take a random sample $R'$ of expected size $r \log n$ from
   $\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$
2. for $i = 0, \ldots, d$:
3. $\quad v_i = \text{Clarkson}(\mathcal{C} \cup \{(R', \{v_0, \ldots, v_{i-1}\})\})$
4. return $v_d$

Let $H_\mathcal{C} = \{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$. The same analysis shows that $|H_{\mathcal{C} \cup \{(R', \{v_0, \ldots, v_{i-1}\})\}}| \leq r \log n + O(|H_\mathcal{C}|/r)$ w.h.p. Thus, the depth of the recursion is $O(\log_r n)$ and the number of recursive calls is $(d+1)^{O(\log_r n)}$ w.h.p., since $(d+1)^{O(\log_r n)}$ is polynomial in $n$. In particular, $|\mathcal{C}| = O(\log_r n)$.

We can compute $|H_\mathcal{C}|$ (line 0) easily in one pass in $O(n|\mathcal{C}|)$ time. Line 1 can also be done in another pass within the same time bound. The algorithm thus takes $(d+1)^{O(\log_r n)}$ passes, requires $O(r \log n \log_r n)$ space, and runs in $O((d+1)^{O(\log_r n)} n \log_r n)$ time w.h.p. Setting $r = n^{\Theta(\delta)}$ (and readjusting $\delta$) yields the theorem. $\square$

The algorithm can be derandomized with $O(2^{O(1/\delta)} n^{1+\delta})$ running time, like in Theorem 3.5 (because $(1/r)$-approximations are special cases of $(1/r)$-nets).

The above theorem appears weaker than Theorem 3.8 in terms of the dependence on $\delta$, but Clarkson's recursive algorithm has a few advantages. First, the number of passes is polynomial in $d$ for a fixed $\delta$. Second, the randomized version can be applied to the entire class of *LP-type* problems [36] (including convex programming [1]).

## 3.4 A lower bound

We now establish a lower bound for linear programming in 2-d (and consequently in higher dimensions). We show that the algorithm in Theorem 3.1 is nearly optimal (up to a constant factor in the number of passes, and ignoring the $1/\delta^{O(1)}$ factor in the space). Our proof is based on an adversary argument by Munro and Paterson [30] that established a similar lower bound for selection. Our proof is not a straightforward adaptation, however, because (i) linear programming is different from selection (needless to say), and (ii) Munro and Paterson's proof assumes a comparison-based model of computation where the only allowable operations on the input elements are comparisons of two elements. This model is not sufficient to solve geometric problems.

Our proof works under a very general decision-tree computation model: input coefficients are real numbers of unlimited precision, and the only allowable operations on the input halfplanes are testing the sign of a function evaluated at the coefficients of a subset of halfplanes currently in memory. The test function can be any continuous function (typically but not necessarily multi-variate polynomials).

**Theorem 3.10** *Any $\lfloor 1/\delta \rfloor$-pass algorithm that can find the lowest point in the intersection of $n$ upper halfplanes in $\mathbb{R}^2$ must require a storage of $\Omega(n^\delta)$ points.*

For the proof of the above theorem, it is more convenient to work in dual space, where the problem is as follows: given two $n$-point sets $P_1$ and $P_2$ separated by the $y$-axis, find the *bridge*, i.e., the edge of the upper hull of $P_1 \cup P_2$ at the $y$-axis. We need the lemma below, which roughly states that after one pass, a problem on $2n$ points ($P_1 \cup P_2$) remains as difficult as a problem of about $2n/s$ points ($X_1 \cup X_2$):

**Lemma 3.11** *Given two open disks $D_1, D_2 \subset \mathbb{R}^2$ separated by the $y$-axis, and an algorithm that can store less than $s$ points, there exists a sequence $P_j$ of $n$ points inside $D_j$, a subset $X_j \subseteq P_j$, and an open disk $D''_j \subset D_j$, for each $j \in \{1, 2\}$, such that after we run the first pass of the algorithm on the concatenation of $P_1$ and $P_2$,*

(i) *no point of $X_1 \cup X_2$ is in memory;*

(ii) *the result of the pass would be identical if we move the points of $X_1$ to arbitrary points in $D''_1$ and the points of $X_2$ to arbitrary points in $D''_2$;*

(iii) *the bridge for $P_1 \cup P_2$ is equal to the bridge for $X_1 \cup X_2$ at the $y$-axis, even if we move the points of $X_1$ and $X_2$ as in* (ii)*;*

(iv) *$|X_1| = |X_2| = \lceil n/s \rceil - 1$.*

**Proof:** The adversary builds the first half of the input $P_1$ entirely using copies of $s$ points $p_1, \ldots, p_s$, whose coordinates $(x_1, y_1, \ldots, x_s, y_s)$ are chosen from an open set $U \subset \mathbb{R}^{2s}$ to be specified later. At every step, the adversary identifies a point $p_i$ that is currently not in memory and chooses as the next input point a new copy of $p_i$. When a point $p_k$ is about to be chosen for the $\lceil n/s \rceil$-th time, the adversary stops this process, chooses copies of any point other than $p_k$ to fill in the rest of $P_1$, and sets $X_1$ to contain all $\lceil n/s \rceil - 1$ existing copies of $p_k$. Observe that no two copies of $p_k$ can reside in memory at any time, and at the end of the pass over $P_1$, no copy of $p_k$ is in memory.

Initially, we set $U$ to contain all tuples $(x_1, y_1, \ldots, x_s, y_s)$ such that the points $p_1 = (x_1, y_1), \ldots, p_s = (x_s, y_s)$ form a strictly concave chain inside $D_1$, and for each $i$, there is a tangent line $\ell_i$ that touches the chain only at $p_i$ and intersects $D_2$. This set $U$ is indeed nonempty and open. Whenever the algorithm performs a test, we consider the sign of the test function for each choice $(x_1, y_1, \ldots, x_s, y_s) \in U$; if not all choices yield the same sign, we refine $U$ to a smaller open subset in which they do. At the end of the pass over $P_1$, since $U$ is open, we can fix the coordinates of $p_1, \ldots, p_s$ and find a neighborhood $\hat{D}_1$ of $p_k$ so that moving $p_k$ to any point in $\hat{D}_1$ would produce the same outcome. In fact, each copy of $p_k$ can be moved to a different point in $\hat{D}_1$, since no two copies of $p_k$ can participate in the same test.

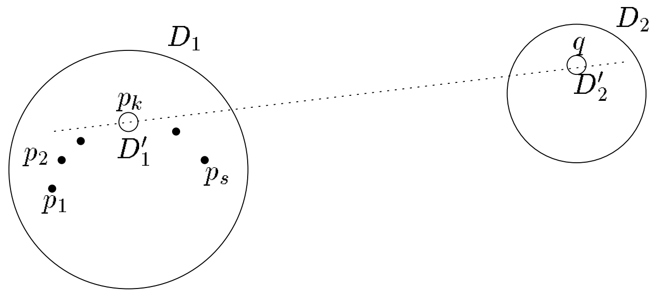


Figure 1: Proof of Lemma 3.11.

We now take a point $q \in D_2$ on a tangent line $\ell_k$ at $p_k$. We can find a sufficiently small neighborhood $D'_1 \subset \hat{D}_1$ at $p_k$ and a sufficiently small neighborhood $D'_2 \subset D_2$ at $q$, such that any line intersecting $D'_1$ and $D'_2$ is above every $p_i$ $(i \neq k)$. (See Figure 1.) Thus, the bridge between $P_1$ and any point set inside $D'_2$ is defined by a point in $X_1$, even if each copy of $p_k$ is moved to an arbitrary point in $D'_1$.

In a similar fashion, the adversary proceeds to build the second half of the input $P_2$ and the subset $X_2$ during the remainder of the first pass, using copies of $s$ points that form a concave chain inside $D'_2$, with tangents intersecting $D'_1$. The disks $D'_1$ and $D'_2$ are similarly refined to $D''_1$ and $D''_2$, so that the bridge between $P_1$ and $P_2$ is defined by a point in $X_1$ and a point in $X_2$, even if the points in $X_1$ and $X_2$ are moved to arbitrary points in $D''_1$ and $D''_2$ respectively. The construction is complete. □

**Proof of Theorem 3.10:** We apply the lemma to obtain the sequences $P_1$ and $P_2$, together with subsets $X_1$ and $X_2$ (of size about $n/s$). The points of $P_1 - X_1$ and $P_2 - X_2$ are considered fixed, while the points of $X_1$ and $X_2$ will later be perturbed; such a perturbation will not change the execution of the first pass by (ii). By (iii), the answer is not yet determined after the first pass, if $|X_1| = |X_2| \geq 2$. In the second pass, we apply the lemma again, to obtain perturbations of the subsequences of the points in $X_1$ and $X_2$ only (skipping over all other points); this process yields its own versions of the subsets $X_1$ and $X_2$ (of size about $n/s^2$) and fixes more points. We can repeat the same process for about $\log_s n$ passes, until $|X_1| = |X_2|$ drops below a constant and the answer is determined. The theorem follows by setting $s \approx n^\delta$. □

### 3.5 A read-only algorithm

One other well-known linear programming algorithm that we have not examined yet is Seidel's randomized incremental algorithm [35]. Unfortunately, this algorithm requires a random permutation of the input, which we cannot afford to store in the multi-pass setting; even if the input is given in random order, the algorithm requires at least a logarithmic expected number of passes. Still, Seidel's algorithm can be helpful in the less restrictive read-only model, if the algorithm is applied in a recursive fashion, as we demonstrate below. The same recursive idea appeared in another paper [9] (in the context of solving "implicit" linear programs). The space bound here beats our multi-pass lower bound.

**Theorem 3.12** *Given a read-only array of $n$ halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an algorithm that runs in $O(n)$ expected time and uses $O(\log n)$ space.*

**Proof:** We describe a recursive algorithm, assuming that the input resides in the union of at most $d$ subarrays each of size at most $n$: First divide the input into at most $dr$ blocks of size $n/r$. The idea is to treat each block as a single constraint. More precisely, each block represents a convex object (the intersection of the halfspaces in the block), and the problem is to find the lowest point inside the intersection of these $r$ convex objects—a convex programming problem. It is important to note that these objects are not explicitly constructed but are rather viewed abstractly. Sharir and Welzl [36] showed that a variation of Seidel's randomized incremental algorithm [35] for linear programming can be used for convex programming (or more generally, LP-type problems). For $O(r)$ constraints, the algorithm requires an expected $O(r)$ number of "violation tests"—deciding whether a point is outside a given object—and an expected $O(\log^d r)$ number of "basis evaluations"—computing the optimal solution for a given subset of $d$ objects. For our convex objects, each violation test can be done in $O(n/r)$ time and $O(1)$ space by scanning the $n/r$ halfspaces in a block; a basis evaluation can be done by making a recursive call for the union of $d$ subarrays. We therefore obtain the following recurrence for the running time for some constant $c$:

$$T(n) \leq (c \log^d r) T(n/r) + O(r \cdot n/r).$$

Setting $r$ to be a sufficiently large constant (depending only on $d$) yields $T(n) = O(n)$. The space requirement is $O(r)$ times the depth of the recursion $O(\log_r n)$. $\square$

The above recursive algorithm also works well in the cache-oblivious model.

# 4 2-d Convex Hulls for Sorted Point Sets

We next examine special cases of the 2-d convex hull problem where more efficient multi-pass algorithms are possible. In this section, we consider the case when the input points are given in sorted order according to their $x$-coordinates. Here, the best traditional algorithm is Graham's scan, which takes linear time [34]; however, this algorithm requires $\Omega(n)$ space in the worst case. We give different strategies with sublinear space. (Similar strategies might work also for input that is preprocessed and stored in other "natural" orders.)

We first introduce some terminology: a *partial hull* $H$ of a point set refers to a subset of the edges of the upper hull; a *gap* of $H$ refers to a maximal closed (vertical) slab whose interior does not intersect any edges of $H$; the *gap size* of $H$ refers to the maximum number of points in a gap.

## 4.1 Two algorithms

Our first algorithm is simple but not time-optimal.

**Theorem 4.1** *Given $n$ points in $\mathbb{R}^2$ sorted from left to right, we can generate the vertices of the upper hull from left to right by an $O(1/\delta)$-pass algorithm that uses $O((1/\delta^2)n^{1/2+\delta})$ space and runs in $O((1/\delta)n \log n)$ time.*

**Proof:** We first build a partial hull $H$ with gap size at most $\sqrt{n}$, as follows: just divide the plane into $\sqrt{n}$ (vertical) slabs each with $\sqrt{n}$ points, and take the bridges at the walls of these slabs. Computing these $\sqrt{n}$ bridges reduces to solving $q = \sqrt{n}$ linear programs on a common set of constraints in 2-d, by duality, and can be done using Theorem 3.3.

Once $H$ is constructed, we can finish in a single pass by examining each gap $\sigma$ of $H$ from left to right and computing the portion of the hull within $\sigma$, by applying an internal-memory algorithm (Graham's scan) to the points within $\sigma$. With $O(\sqrt{n})$ space, the final pass takes linear time. □

We now present a more complicated linear-time method by adapting the standard "merge-hull" algorithm [34] based on bottom-up divide-and-conquer. To minimize the number of passes, we implement the divide-and-conquer in a breadth-first rather than depth-first manner, and we replace binary merges with $r$-way merges. We also need to generalize merging of convex hulls to merging of partial hulls, accomplished by the following lemma:

**Lemma 4.2** *Given two vertically separated point sets in $\mathbb{R}^2$, and given their partial hulls with gap size $g$ stored in main memory, we can find the bridge between the two point sets by a 2-pass algorithm that uses $O(g)$ extra space and runs in time linear in the number of points. Moreover, at the end of each pass, the extra space usage is reduced to $O(1)$.*

**Proof:** We apply the binary-search algorithm by Overmars and van Leeuwen [32, 34] to the two partial hulls $H_A$ and $H_B$ of the two point sets $A$ and $B$. In each iteration of this algorithm, we examine the median edges of $H_A$ and $H_B$, perform some constant-time operations, and throw away half of one of the two partial hulls. After $O(\log n)$ iterations, one of the partial hulls, say $H_A$, has no edge remaining. In this case, we have identified a gap $\sigma_A$ of $H_A$ that contains a vertex defining the solution. In one pass, we read in the $O(g)$ points of $A$ inside $\sigma_A$, compute the hull $H'_A$ of these points, and apply the binary-search algorithm to $H'_A$ and $H_B$. After $O(\log n)$ iterations, $H_B$ has no edge remaining and we have identified a gap $\sigma_B$ of $H_B$ that contains the other vertex defining the solution. We release $H'_A$ from memory. In a second pass, we read in the $O(g)$ points of $A$ in $\sigma_A$ and the $O(g)$ points of $B$ in $\sigma_B$ and return their bridge. □

**Theorem 4.3** *The running time in Theorem 4.1 can be improved to $O((1/\delta)n)$.*

**Proof:** We use a different method based on bottom-up merging, outlined below, to compute a partial hull $H$ of the input point set with gap size at most $\sqrt{n}$. As in the proof of Theorem 4.1, once $H$ has been computed, we can fill in the gaps in a final pass in linear time.

```
0. divide the plane into √n vertical slabs each with √n points
1. for each slab σ, create a partial hull of the points inside σ with 0 edges
2. while there is more than one partial hull in existence:
3.     divide the current partial hulls into groups of r members each, from left to right
4.     for each group G:
5.         compute the bridge between every pair of partial hulls in G
6.         merge the r partial hulls in G into a single partial hull
```

All partial hulls here have gap size at most $\sqrt{n}$ and they have total size $O(\sqrt{n})$ at every iteration. After $O(\log_r n)$ iterations, we arrive at one partial hull of the whole point set, as desired. Line 5 takes $O(r^2)$ calls per group to the subroutine in Lemma 4.2. The key idea is to perform these calls simultaneously over all groups: the number of passes is still two, the extra space usage is $O(r^2\sqrt{n})$, and the running time is $O(rn)$. (It should be noted that during a pass, as we go from one group to another, we only need to retain $O(r^2)$ amount of information per group due to the last part of

Lemma 4.2.) Once the bridges have been identified in line 5, line 6 can be done easily by using the highest bridge at each of the $r-1$ vertical separating lines. The algorithm thus makes a total of $O(\log_r n)$ passes, uses $O(r^2\sqrt{n})$ space, and runs in $O(rn\log_r n)$ time. Setting $r = n^{\delta/2}$ yields the theorem. □

### 4.2 A lower bound

We now show that the near-$\sqrt{n}$ space complexity is actually close to optimal. Our lower-bound proof this time is information-theoretic rather than adversary-based.

**Theorem 4.4** *Any $O(1)$-pass algorithm that can generate (and print out) the vertices of the upper hull (in any order), given $n$ points in $\mathbb{R}^2$ sorted from left to right, must require $\Omega(\sqrt{n/\log n})$ units of storage, where a "unit" refers to a point or $\log n$ bits of information. Here, it is assumed that to print a point to the output stream, the point must currently reside in memory.*

**Proof:** Set $r = 2\sqrt{n\log n}$. Create a point set $P$ as follows: take $r$ uniformly spaced arcs of length $\varepsilon\varepsilon'$ on the upper part of the unit circle and place a group $G_i$ of $n/r$ points on the $i$-th arc (indexed from left to right); also place an extra point at the leftmost point of the circle. Let $v_i$ be the rightmost point in $G_i$. Given a string $z \in \{0,1\}^r$, define a modified point set $P(z)$, obtained by moving $v_i$ upward by a distance of $\varepsilon$ whenever the $i$-th bit of $z$ is 1. By making $\varepsilon$ and $\varepsilon'$ sufficiently small, $P(z)$ obeys the following property: if the $i$-th bit of $z$ is 0, all points in $G_i$ are extreme; otherwise, no point in $G_i$ is extreme except for $v_i$.

Suppose the algorithm makes $p$ passes and at any time stores at most $s$ units of space. Assume that $ps < \sqrt{n/\log n}$. For a given input, define the *exit configuration* to be the concatenation of the memory content after each of the $p$ passes. (Here, for a point, its "content" comprises its index, together with an extra bit indicating whether it has been moved upward.) The number of different exit configurations is $2^{ps(\log n+O(1))} < 2^r$. By the pigeonhole principle, there exist two strings $z, z' \in \{0,1\}^r$ such that $P(z)$ and $P(z')$ have the same exit configuration.

Suppose that $z$ and $z'$ agree in the first $i-1$ bits, but the $i$-th bit of $z$ is 0 and the $i$-th bit of $z'$ is 1. Consider each pass made by the algorithm on the two inputs $P(z)$ and $P(z')$. Since the memory content at the end of the previous pass is identical and the inputs are identical up to $v_i$, the algorithm behaves identically before $v_i$ is read during the pass; so, no points in $G_i$ can be printed until $v_i$ is read. At most $s$ points are stored at this time; so at most $s$ points in $G_i$ can be printed in the rest of the pass. On input $P(z)$, the algorithm is supposed to print all $n/r$ points in $G_i$ in $p$ passes. Thus, $ps \geq n/r = \Omega(\sqrt{n/\log n})$. □

We comment that the assumption made in the above theorem seems reasonable, but a stronger lower bound without the assumption would be desirable. Our proof does not work, for example, if we can print just the indices (relative to the sorted input order) of the points on the hull, or if we just want to count the overall number of hull vertices.

### 4.3 A read-only algorithm

To contrast with the above lower bound result, we show that a more space-efficient algorithm is possible in the read-only setting.

**Theorem 4.5** *Given a read-only array of $n$ points in $\mathbb{R}^2$ sorted from left to right, we can generate the vertices of the upper hull from left to right by an algorithm that uses $O((1/\delta)n^\delta)$ extra space and runs in $O((1/\delta)n)$ expected time.*

**Proof:** We describe a recursive algorithm: First divide the input into $r$ blocks each containing $n/r$ points, from left to right. As in the proof of Theorem 3.12, it is helpful to view each block abstractly as a single convex object. Consider the following variant of Graham's scan to compute all tangents of the upper hull of $r$ vertically separated convex objects $\gamma_1, \ldots, \gamma_r$, ordered from left to right:

```
0. e_1 = tangent between γ_1 and γ_2, i_0 = 1, i_1 = 2, j = 1
1. for i = 3, ..., r:
2.    while j > 0 and γ_i is not entirely below the line through e_j, j = j − 1
3.    e_{j+1} = tangent between γ_{i_j} and γ_i, i_{j+1} = i
4.    j = j + 1
5. return ⟨e_1, ..., e_j⟩
```

The above method performs $O(r)$ "primitive operations"—testing whether an object is entirely below a line (line 2), and finding the tangent (bridge) between two objects (lines 0 and 3). In our case, each object is the upper hull of the points in a block. The first type of operation can be carried out in $O(n/r)$ time and $O(1)$ space by scanning the points in the block. The second type can be handled by Theorem 3.12 in $O(n/r)$ expected time and $O(\log n)$ space. The method then gives us a partial hull with gap size at most $n/r$.

After this process, we take each gap of this partial hull and recursively output the upper hull of the points within each gap. The expected running time of the whole algorithm satisfies the recurrence

$$T(n) = rT(n/r) + O(r \cdot n/r),$$

which solves to $T(n) = O(n \log_r n)$. The space bound is $O(r \log_r n)$. Setting $r = n^\delta$ yields the theorem. $\square$

We can avoid randomization if we do not mind a larger constant factor:

**Theorem 4.6** *The algorithm in Theorem 4.5 can be made deterministic if the running time is increased to $O(2^{O(1/\delta)}n)$.*

**Proof:** In the second type of operation, we want to find a bridge of the upper hull of $2n/r$ points. Instead of using Theorem 3.12, we handle this operation by a recursive call to our upper-hull algorithm. (From the output of the algorithm, we can skip over all edges but keep only the bridge we want.) As a result, we obtain a new recurrence:

$$T(n) = O(r)T(2n/r) + O(r \cdot n/r),$$

which solves to $T(n) = O(2^{O(\log_r n)}n)$. The space bound is the same. Setting $r = n^\delta$ yields the theorem. $\square$

## 5 2-d Output-Sensitive Convex Hulls

For unsorted point sets, results better than Theorem 2.1 are still possible if the output size $h$ is small. For example, for points distributed uniformly in a square, it is known that the expected value of $h$ is logarithmic [34] (although for this special case there is actually a simple one-pass algorithm).

We prove the following theorem by mimicking Kirkpatrick and Seidel's output-sensitive algorithm [24] based on top-down divide-and-conquer. Again, the divide-and-conquer is executed breadth-first and with a larger branching factor. (It is interesting to note that Kirkpatrick and Seidel's algorithm was originally designed to make the running time output-sensitive, not the space; other output-sensitive convex hull algorithms do not seem to work as well in the multi-pass model.)

**Theorem 5.1** *Given $n$ points in $\mathbb{R}^2$, we can generate the $h$ vertices of the upper hull from left to right, by an $O(1/\delta^2)$-pass algorithm that uses $O((1/\delta^2)hn^\delta)$ space and runs in $O((1/\delta^2)n\log n)$ time.*

**Proof:** The "unfolded" version of Kirkpatrick and Seidel's algorithm is outlined below. Here, we repeatedly insert edges to a partial hull $H$ until it becomes the complete hull:

0. $H = \emptyset$
1. for $i = 0, 1 \ldots$:
2.     if the gap size of $H$ is below a constant then
           compute and print the hull within each gap and return
3.     for each gap $\sigma$ of $H$:
4.         divide $\sigma$ into $r_i$ subslabs each containing $O(1/r_i)$-th of the points
5.         compute the bridges at the walls of these subslabs and insert them to $H$

The simplest choice of parameters is $r_0 = r_1 = \cdots = r$, with $O(\log_r n)$ iterations of the outer loop. Line 4 can be carried out by an approximate quantiles algorithm [21, 30]. Line 5 requires solving $r$ linear programs and can be carried out by Theorem 3.3 in $O(1/\delta)$ passes, $O((1/\delta)rm^\delta)$ space, and at most $O((1/\delta)m\log m)$ time, where $m$ is the number of points in the gap $\sigma$.

In each iteration of the outer loop, there are at most $h+1$ gaps. The key idea is to handle all $O(h)$ iterations of the inner loop (lines 3–5) simultaneously. Each point belongs to only one gap of $H$, which can be identified in $O(\log h)$ time. So, the number of passes for lines 3–5 remains $O(1/\delta)$, the space usage is $O((1/\delta)hrn^\delta)$, and the running time is at most $O((1/\delta)n\log n)$. The algorithm thus makes $O((1/\delta)\log_r n)$ passes, uses $O((1/\delta)hrn^\delta)$ space, and runs in $O((1/\delta)n\log n\log_r n)$ time. The theorem follows by setting $r = n^\delta$ (and readjusting $\delta$). $\square$

We can make the running time output-sensitive as well, provided that an upper bound on $h$ is given. (Note that a standard trick [8] of "guessing" the output size cannot be applied, because it would increase the number of passes by a nonconstant factor.)

**Theorem 5.2** *The running time in Theorem 5.1 can be improved to $O((1/\delta^2)(n\log^{(c)} n + n\log \bar{h}))$ time, where $\bar{h}$ is a known upper bound on $h$.*

**Proof:** The algorithm is the same, but we provide a more careful analysis using a different choice of parameters.

For line 4, we use a variant of Munro and Paterson's approximate quantiles algorithm, as in the proof of Theorem 3.3. For line 5, we use Theorem 3.3. Then in iteration $i$ of the outer loop, the running time becomes $O(n\log h + (1/\delta)(n_i \log^{(c)} n_i + n_i \log r_i))$, where $n_i$ is the number of points inside the gaps of $H$ at the beginning of the iteration.

We choose $r_0 = \bar{h}\log^{(c-1)} n$, $r_1 = \bar{h}\log^{(c-2)} n$, $\ldots$, $r_{c-2} = \bar{h}\log n$, and $r_{c-1} = r_c = \cdots = \bar{h}n^{\delta/2}$, with at most $c - 1 + \lceil 2/\delta \rceil$ iterations. Since $n_i = O(\bar{h}n/r_{i-1})$, we have

$$O(n_0 \log r_0) = O(n\log^{(c)} n + n\log \bar{h}), \quad \text{and} \quad O(n_i \log r_i) = O\left(\frac{\bar{h}n}{r_{i-1}} \log r_i\right) = O(n\log \bar{h}) \quad (i \geq 1).$$

So the running time of each iteration is at most $O((1/\delta)(n\log^{(c)} n + n\log \bar{h}))$. The upper bounds on the number of passes and space are unchanged. $\Box$

We can also obtain a tradeoff result analogous to Theorem 2.1. (Due to the sorting lower bound, our result is near optimal except for the $n^\delta$ and logarithmic factors.)

**Theorem 5.3** *Given $n$ points in $\mathbb{R}^2$ and a parameter $s$, we can generate the $h$ vertices of the upper hull from left to right, by an $O((1/\delta^2)\lceil h/s \rceil)$-pass algorithm that uses $O((1/\delta^2)sn^\delta)$ space and runs in $O((1/\delta^2)\lceil h/s \rceil n\log n)$ time.*

**Proof:** It is straightforward to modify the algorithm in Theorem 5.1 to output the leftmost $s$ hull edges with $O((1/\delta^2)sn^\delta)$ space (by only keeping the leftmost $s$ edges of $H$ in every iteration). Repeating this process $\lceil h/s \rceil$ times yields the theorem. $\Box$

## 6 3-d Convex Hulls

Finally, we consider the 3-d convex hull problem. As before, due to sorting lower bounds, we cannot hope for a nontrivial result with a constant number of passes in general. Still, we show that a result analogous to Theorem 2.1 is possible. This time, the algorithm is more involved but relies on a standard divide-and-conquer approach in dual space (similar to the proof of Theorem 3.4 as well as an algorithm by Clarkson and Shor [15]).

**Theorem 6.1** *Given $n$ points in $\mathbb{R}^3$ and $s \geq n^{1/c}$ for a fixed constant $c$, we can generate the facets of the upper hull by an $O((n/s)\,\mathrm{polylog}\, n)$-pass algorithm that uses $O(s)$ space and runs in $O((n^2/s)\,\mathrm{polylog}\, n)$ time w.h.p.*

**Proof:** We solve the dual problem of computing the vertices of the lower envelope of a set $H$ of planes in $\mathbb{R}^3$ by the following recursive procedure (initially, we call $\textsc{Env}_0(\mathbb{R}^3)$):

$\textsc{Env}_i(\Delta)$:
0. if $i = c$ then
   compute and print the vertices of the lower envelope of $\{h \in H \mid h \text{ intersects } \Delta\}$
   inside $\Delta$ and return
1. take a random sample $R$ of expected size $r\log n$ of $\{h \in H \mid h \text{ intersects } \Delta\}$
2. compute a triangulation $T$ of the lower envelope of $R$ restricted inside $\Delta$
3. for each simplex $\Delta' \in T$ do $\textsc{Env}_{i+1}(\Delta')$

Let $H_\Delta = \{h \in H \mid h \text{ intersects } \Delta\}$. As in the analysis of Theorem 3.4, we have $|H_{\Delta'}| = O(|H_\Delta|/r)$ for all $\Delta' \in T$ w.h.p. So, at each leaf (on the $c$-th level) of the recursion, we have $|H_\Delta| = O(n/r^c)$ w.h.p. Since lower envelopes have linear complexity in $\mathbb{R}^3$, we can find a triangulation $T$ of size $O(r \log n)$. So, there are $O((r \log n)^c)$ leaves in the recursion.

Line 0 requires a single pass with $O(n/r^c)$ space and $O(n + (n/r^c)\log(n/r^c))$ time w.h.p. Line 1 can be easily done in one pass. Line 2 takes $O(|R|\log|R|) = O(r \log^2 n)$ time [29, 34] in main memory. The overall number of passes is thus $O((r \log n)^c)$, the space usage is $O(n/r^c + r \log n)$, and the running time is $O((r \log n)^c \cdot [n + (n/r^c)\log(n/r^c) + r \log^2 n])$ w.h.p. The theorem follows by setting $r = (n/s)^{1/c}$ (and noting that $(n/s)^{1/c} = o(s)$ for $s \geq n^{1/c}$). $\square$

By the standard lifting transformation, we immediately obtain the same result for Voronoi diagrams of 2-d point sets.

# 7 Concluding Remarks

We have given new algorithms and lower bounds for some basic geometric problems under the multi-pass model. See Table 1 for a summary. We hope that the techniques here (like pipelining) may be applicable to solve other geometric problems and may inspire further work. Although one-pass streaming algorithms admittedly are more desirable than multi-pass algorithms in many settings involving massive data sets, certain hardware systems such as the graphics card [2] do favor streaming computation with multiple passes. The multi-pass model can also be viewed as a simpler form of external-memory algorithms.

In some ways, one might regard streaming algorithms, which process input sequentially, as the opposite of parallel algorithms. Curiously though, many of our algorithms (e.g., see Theorem 3.4) are naturally parallelizable, as each pass is parallelizable. Apparently, designing algorithms that minimize the number of passes or rounds share some similarities to designing parallel algorithms that minimize processor time.

Our work on the multi-pass model raises many interesting theoretical open problems concerning tradeoffs of space, the number of passes, and running time for geometric problems. For example, we mention the following questions:

- In view of Theorem 3.7, what is the exact smallest number of passes required to solve $d$-dimensional linear programming with sublinear space? Can one prove a lower bound of $d+1$ passes?

- In view of Theorem 3.2, what is the most time-efficient deterministic algorithm for linear programming in 2-d (or higher dimensions) that uses a constant number of passes and $O(n^\delta)$ space? A similar question for the selection problem also seems to be open (Munro and Paterson's algorithm [30] requires superlinear time).

- In view of Section 4, can one find good orderings of the input that enable other problems to be solved effectively in the streaming setting?

**Acknowledgement.** We thank Ian Munro for suggesting that we look into algorithms in the read-only model.

| # passes | space | time | reference |
|---|---|---|---|
| | | 2-d convex hull | |
| $O(n/s)$ | $O(s)$ | $O(n(\log n + n/s))$ | Theorem 2.1 |
| $O(n/s)$ | $\Omega(s)$ | $\Omega(n(\log n + n/s))$ | [30] |
| $O(1)$ | $O(hn^{\delta})$ | $O(n \log^{(c)} n + n \log \bar{h})$ | Theorems 5.1 and 5.2 |
| | | 2-d convex hull: sorted case | |
| $O(1)$ | $O(n^{1/2+\delta})$ | $O(n)$ | Theorems 4.1 and 4.3 |
| $O(1)$ | $\Omega(\sqrt{n/\log n})$ | $\Omega(n)$ | Theorem 4.4 |
| read-only | $O(n^{\delta})$ | $O(n)$ | Theorems 4.5 and 4.6 |
| | | 3-d convex hull | |
| $O((n/s) \operatorname{polylog} n)$ | $O(s)$ | $O((n^2/s) \operatorname{polylog} n)$ | Theorem 6.1 |
| | | fixed-d linear programming | |
| $d+1$ | $O(\sqrt{n \log n})$ | $O(n)$ randomized | Theorem 3.7 |
| $O(1)$ | $O(n^{\delta})$ | $O(n)$ randomized | Theorem 3.8 |
| $O(1)$ | $O(n^{\delta})$ | $O(n^{1+\delta})$ | Theorems 3.4 and 3.5 |
| $O(1)$ | $O(n^{\delta})$ | $O(n \log^{(c)} n)$ for $d = 2$ | Theorems 3.1 and 3.2 |
| $O(1)$ | $\Omega(n^{\delta})$ | $\Omega(n)$ | Theorem 3.10 |
| read-only | $O(\log n)$ | $O(n)$ randomized | Theorem 3.12 |

Table 1: Summary of results. (Here, $\delta > 0$ is an arbitrarily small constant and $c$ is an arbitrarily large constant, where asymptotic notation may hide dependence on these constants; $s$ is any value between 1 to $n$, except for the 3-d convex hull case, where we must have $s \geq n^{\delta}$.)

## References


[1] I. Adler and R. Shamir. A randomization scheme for speeding up algorithms for linear and convex quadratic programming problems with a high constraints-tovariables ratio. *Math. Programming*, 61:39–52, 1993.

[2] P. K. Agarwal, S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *Proc. 11th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 2832, Springer-Verlag, pages 544–555, 2003.

[3] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. In *Proc. 20th ACM Sympos. Comput. Geom.*, pages 144–151, 2004.

[4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 209–218, 2002.

[5] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11:287–297, 1982.

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

[7] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. 20th ACM Sympos. Comput. Geom.*, pages 239–246, 2004.

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

[9] T. M. Chan. An optimal randomized algorithm for maximum Tukey depth. In *Proc. 15th ACM-SIAM Sympos. Discrete Algorithms*, pages 423–429, 2004.

[10] T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proc. 20th ACM Sympos. Comput. Geom.*, pages 152–159, 2004.

[11] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10:229–249, 1990.

[12] B. Chazelle, D. Liu, and A. Magen. Sublinear geometric algorithms. In *Proc. 35th ACM Sympos. Theory Comput.*, pages 531–540, 2003.

[13] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[14] K. L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *J. ACM*, 42:488–499, 1995.

[15] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[16] P. Drineas and R. Kannan. Pass efficient algorithms for approximating large matrices. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms*, pages 223–232, 2003.

[17] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM J. Comput.*, 13:31–45, 1984.

[18] M. E. Dyer and A. M. Frieze. A randomized algorithm for fixed-dimensional linear programming. *Math. Programming*, 44:203–212, 1989.

[19] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proc. 31st Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 3142, Springer-Verlag, pages 531–543, 2004.

[20] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proc. 16th ACM-SIAM Sympos. Discrete Algorithms*, pages 745–754, 2005.

[21] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. SIGMOD*, pages 58–66, 2001.

[22] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987.

[23] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Note 1998-011, Digital Systems Research Center, Palo Alto, CA, 1998.

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

[25] J. Matoušek. Derandomization in computational geometry. In *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), pages 559–595, Elsevier, Amsterdam, 2000.

[26] N. Megiddo. Linear time algorithms for linear programming in $R^3$ and related problems. *SIAM J. Comput.*, 12:759–776, 1983.

[27] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31:114–127, 1984.

[28] N. Megiddo. Partitioning with two lines in the plane. *J. Algorithms*, 6:430–433, 1985.

[29] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

[30] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoret. Comput. Sci.*, 12:315–323, 1980.

[31] S. Muthukrishnan. Data streams: Algorithms and applications. Manuscript, `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`, 2003.

[32] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.*, 23:166–204, 1981.

[33] J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *Proc. 39th IEEE Sympos. Found. Comput. Sci.*, pages 264–268, 1998.

[34] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[35] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

[36] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, Lect. Notes Comput. Sci., vol. 577, Springer-Verlag, pages 569–579, 1992.

[37] S. Suri, C. D. Tóth, and Y. Zhou. Range counting over multidimensional data streams. In *Proc. 20th ACM Sympos. Comput. Geom.*, pages 160–169, 2004.