Multi-Pass Geometric Algorithms

Timothy M. Chan[*]
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
tmchan@uwaterloo.ca

Eric Y. Chen
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
y28chen@uwaterloo.ca

ABSTRACT

We initiate 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.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*

General Terms

Algorithms, Theory

Keywords

Streaming algorithms, convex hulls, linear programming

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 [25], 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., [2, 7, 31]).

Algorithms that are allowed to make multiple (but a small number of) passes over the input with limited working space have also been touched upon in some of the previous papers both in geometry (e.g., [1, 31]) and in other areas (e.g., [3, 12, 14, 15, 17]). Here, we are concerned with 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*.

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 [24] in 1980 defined the multi-pass 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 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 finding 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 finding 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. 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.)

[*]Research of this author has been supported by an NSERC Research Grant and a Premier's Research Excellence Award.

As an example, consider the point of finding a line that minimizes the largest vertical distance to a collection of n data points in the plane. Results in the data-stream model [7] 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 [5], 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: not only do we lack random access to the input, but the input space can only be read and not written. This *read-only* characteristic is desirable, because the original input is retained during and after the execution of the algorithm. (Of course, we can make a copy of the input first, but that defeats the purpose of space-efficiency.) In-place algorithms for many geometric problems [5] 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.

If read-only random access to the input is allowed and the number of passes is no longer a concern, some of our results can be improved. Specifically, 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), where the vertices are stored in an array in order. Previous in-place algorithms for linear programming cannot be used, because permuting vertices would destroy the polygon.

Read-only algorithms have been previously considered for the sorting problem [4, 27] but are apparently less often studied. One advantage of such algorithms 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 [8], 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 [24] observed, there is a simple algorithm for sorting n numbers using $O(s)$ space, $O(n/s)$ passes, and $O(n(\log n + n/s))$ comparisons: the algorithm simply find the next s smallest elements in each pass. 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:

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. The pseudocode is as follows:

```
0. $v$ = the leftmost point
1. repeat:
2.     find a vertical slab $\sigma$ containing $s$ points
       with its left wall through $v$
3.     let $\langle q_0, \ldots, q_j \rangle$ be the upper hull of
       the points inside $\sigma$
4.     for each point $p$ to the right of $\sigma$:
5.         while $p$ is above $\overleftrightarrow{q_{j-1}q_j}$, $j = j - 1$
6.         set $j = j + 1$ and $q_j = p$
7.     print out $q_0, \ldots, q_j$ and set $v = q_j$
```

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 σ. This is accomplished by imitating Graham's scan [28] (lines 4–6), except that points are processed not necessarily in sorted order (although points inside σ are examined in line 3 before points to the right of σ). By induction, one can check that at the end of the for loop, $q_{j-1}q_j$ is indeed the desired bridge. After at most $\lceil n/s \rceil$ iterations, the entire upper hull is computed.

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 repeat the following: 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, and remove the s largest elements from the buffer. 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 takes at most $2\lceil n/s \rceil$ passes, uses $O(s)$ space, and runs in $O((n/s) \cdot (n + s \log s))$ time. $\square$

Even in the relaxed read-only model, no algorithm with substantially better time-space product is possible, due to known lower bounds for sorting [4]. 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 δ.)

3.1 Prune-and-search in 2-d

We begin in 2-d with the first published linear-time algorithm by Megiddo [20] and Dyer [13], 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 σ, $\textsc{List}_{r,\sigma}(H)$ outputs a stream of vertical lines, and $\textsc{Filter}_{r,\sigma}(H)$ outputs a subset of halfplanes.

```
$\textsc{List}_{r,\sigma}(H)$: repeat:
  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 $\sigma$

$\textsc{Filter}_{r,\sigma}(H)$: repeat:
  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 participate
     in $I \cap \sigma$
```

For an input stream of size n, both subroutines 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:

```
0. let $\sigma_0$ be the whole plane
1. for $i = 0, 1, \ldots$:
2.     if size of $\textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$
       is below $r_i$ then return solution 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
       $\textsc{List}_{r_i,\sigma_i}(\textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$
4.     decide which subslab contains the solution, and
       let this subslab be $\sigma_{i+1}$
```

Let $H_i = \textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$, and let $n_i = |H_i|$. Before iteration i, we assume that the solution lies in the vertical slab σ_i and is defined only by halfplanes in H_i. Halfplanes not in H_i can be considered pruned, although they are not physically removed from the input (and H_i is not actually stored by the algorithm). After iteration i, since we know that the solution lies in σ_{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. The invariant is thus preserved.

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 σ_{i+1}. Observe that if $h_1 \cap \cdots \cap h_{r_i}$ has j vertices inside σ_{i+1}, then at most $j+2$ halfplanes can participate in $h_1 \cap \cdots \cap h_{r_i} \cap \sigma_{i+1}$. Thus, $\textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$ generates at most $O(n_i/r_i) + 2\lceil n_i/r_i \rceil$ halfplanes. In other words, $n_{i+1} = O(n_i/r_i)$.

We analyze the cost of iteration i of the algorithm. The key observation is that although H_i cannot be stored, it can be generated as an output stream from the original input, by running the i $\textsc{Filter}$ operations in a pipeline. The execution of this 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 feed this output stream to $\textsc{List}$ and then to 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 [24] provided a simple tree-based algorithm for this task that requires one pass and $O(r_i \log^2 n_i)$ space (they 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); it can be checked that the running time is $O(n_i \log(r_i \log n_i))$. (See [16] for another approximate quantiles algorithm.)

For line 4, recall [13, 20] that deciding whether the solution is to the left or right of a line ℓ reduces to computing the intersection of the halfplanes at ℓ, 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 would be $r_0 = r_1 = \cdots = r$, with $O(\log_r n)$ iterations. The algorithm thus makes $O(\log_r n)$ passes, uses $O(r \log^2 n)$ 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 can be checked that the space bound is now $O(br_i \log_b^2 n_i)$ and the time bound is $O(n_i \log(r_i \log_b n_i))$. 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 method by repeating the following: read in the next r_i halfplanes, compute their intersection I in $O(r_i \log r_i)$ time, compute the intersection of I with the $O(r_i)$ vertical lines in an additional $O(r_i \log r_i)$ time, and update the current answers at the $O(r_i)$ vertical lines. 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. The algorithm still makes $O(1/\delta)$ passes and uses at most $O((1/\delta^2)n^\delta)$ space. Since $n_{j+1} = O(n_j/r_j)$ for all j, the running time of the i-th iteration is now

$$O(n_0 \log r_0 + \cdots + n_i \log r_i) \;=\; O(n \log^{(c)} n). \qquad \square$$

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 σ_i is not a single slab but the union of up to q slabs. In line 3, σ_i is divided into up to $q+r_i$ subslabs. In line 4, σ_{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. $\square$

3.2 Prune-and-search in higher dimensions

Megiddo [21] 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 $|\{h \in H \mid h$ intersects $\Delta\}| \le r \log n$ then return
       solution directly
3.     take a random sample $R$ of expected size $r \log n$
       from $\{h \in H \mid h$ intersects $\Delta\}$
4.     compute the *canonical 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
```

Clearly, the solution is always contained in Δ and is defined only by hyperplanes that intersect Δ.

Let Δ_i be the simplex Δ at the beginning of the i-th iteration, let $H_i = \{h \in H \mid h$ intersects $\Delta_i\}$, and let $n_i = |H_i|$. It is well-known [9, 23] that for a random sample $R \subseteq H_i$ of size $r \log n$, w.h.p. the canonical 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 Δ_{i+1} must intersect Δ_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 Δ_i, put it into R with probability $(r \log n)/n_i$). Line 4 takes $O((r \log n)^d)$ time and space in main memory. (Actually, the bound can be reduced to $O((r \log n)^{\lfloor d/2 \rfloor})$, since only one cell of the arrangement needs to be triangulated.)

For line 5, recall that deciding which side of a given hyperplane h contains a solution reduces to solving a linear program restricted inside h. Thus, line 4 can be done by solving $(r \log n)^{O(1)}$ subproblems in $d-1$ dimensions. We handle these subproblems by making $(r \log n)^{O(1)}$ calls to a $(d-1)$-dimensional algorithm not in series but in parallel. In other words, in every pass, we simulate 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 $P_d(n) = O((\log_r n)P_{d-1}(n))$, $S_d(n) = O((r \log n)^{O(1)} S_{d-1}(n))$, and $T_d(n) = O(((r \log n)^{O(1)}(\log_r n) T_{d-1}(n))$ imply that $P_d(n) = O(\log_r^{d-1} n)$, $S_d(n) = O((r \log n)^{O(1)})$, and $T_d(n) = O(nr^{O(1)} \log^{O(1)} n)$. The theorem follows by setting $r = n^{\Theta(\delta)}$. $\square$

This 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 [19]. Bagchi *et al.* [2] have recently described a tree-based algorithm for computing an $(1/r)$-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 $O(nr^{O(1)} \log^{O(1)} n)$ time.

The canonical 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 [22]. (The decision step here requires a selection algorithm, which was provided by Munro and Paterson.)

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 ℓ such that each side of ℓ 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 [10]. The nonrecursive version of his algorithm, 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 $i = 0, \ldots, d$, let $v_i$ be the solution for
   $\{h \in H \mid h \in R$ or $\exists j < i,\ v_j$ violates $h\}$
2. 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 [10] 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 [23] 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 to get a randomized algorithm with linear running time and little space.

Theorem 3.8. *The running time in Theorem 3.4 can be improved to $O((1/\delta^{O(1)})n)$ w.h.p.*

Proof. We use the same algorithm as in the previous proof, except that line 1 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 $r = n^\delta$. □

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.*

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 $|\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R$ or $\exists v \in V,\ v$ vio-
       lates $h)\}| \le 2r \log n$ then return solution directly
  1. take a random sample $R'$ of expected size $r \log n$ from
       $\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R$ or $\exists v \in V,\ v$ violates $h)\}$
  2. for $i = 0, \ldots, d$,
       $v_i = $ CLARKSON$(\mathcal{C} \cup \{(R', \{v_0, \ldots, v_{i-1}\})\})$
  3. 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 $|H_{\mathcal{C} \cup \{(R', \{v_0, \ldots, v_{i-1}\})\}}| \le r \log n + O(|H_{\mathcal{C}}|/r)$ 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 $r = n^{\Theta(\delta)}$ yields the theorem. □

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 δ, but Clarkson's recursive algorithm has a few advantages. First, the number of passes is polynomial in d for a fixed δ. Second, the randomized version can be applied to the entire class of *LP-type* problems [30].

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 [24] 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.*

Proof. It is more convenient to describe the proof in dual space, where the problem is to find the bridge (upper-hull edge) at the y-axis for a given point set. The lemma below roughly states that after each pass, the problem on $2n$ points remains as difficult as the problem on $2n/s$ points. Applying the lemma repeatedly, we immediately obtain a lower bound of about $\log_s n$ passes for an algorithm that uses s space. The theorem follows by setting $s \approx n^\delta$. □

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 ℓ_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.

We now take a point $q \in D_2$ on a tangent line ℓ_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 $(i \neq k)$. 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. □

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 [29]. 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 [6] (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 [30] showed that a variation of Seidel's randomized incremental algorithm [29] for linear programming can be used for convex programming. 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 $O(\log_r n)$. □

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 [28]; 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 slab whose interior does not intersect any edges of H; the *gap size* of H refers to the maximum number of points in the gaps.

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 σ of H from left to right and computing the portion of the hull within σ, by applying an internal-memory algorithm (Graham's scan) to the points within σ. 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 [28] 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 [26, 28] 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 σ_A of H_A that contains a vertex defining the solution. In one pass, we read in the $O(g)$ points of A inside σ_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 σ_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 σ_A and the $O(g)$ points of B in σ_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, 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 $\sqrt{n}$ vertical slabs each with
   $\sqrt{n}$ points
1. for each slab $\sigma$, create a partial hull of the points
   inside $\sigma$ with 0 edges
2. repeat:
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. 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. We make the following reasonable assumption: to print a point to the output stream, the point must currently reside in memory.

Theorem 4.4. *Any $O(1)$-pass algorithm that can generate 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.*

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 ε whenever the i-th bit of z is 1. By making ε and ε' 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 of 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, $ps \geq n/r = \Omega(\sqrt{n/\log n})$. □

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 $B_1, \ldots, B_r$ 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. A variant of Graham's scan can be used to compute all tangents of the upper hull of r vertically separated convex objects, by performing $O(r)$ "primitive operations"—testing whether an object is below a line, and finding the bridge between two objects. The first type of operation can be carried out in $O(n/r)$ time and $O(1)$ space by scanning the points in a block. The second type can be handled by Theorem 3.12 in $O(n/r)$ expected time and $O(\log n)$ space. As a result, we obtain 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 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. □

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 [28] (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 [18] 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. We repeatedly insert edges to a partial hull H until it becomes the complete hull:

```
0. $H = \emptyset$
1. repeat:
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 $r$ subslabs each containing
             $O(1/r)$-th of the points
5.         compute the bridges at the walls of these
             subslabs and insert them to $H$
```

The number of iterations of the outer loop is clearly $O(\log_r n)$. Line 4 can be carried out by an approximate quantiles algorithm [16, 24]. 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 σ.

At 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. Since each point belongs to only one gap of H (which can be identified in $O(\log h)$ time), 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 δ. $\square$

If h is known, the above time bound can be reduced to $O(n \log^{(c)} n + n \log h)$ by using different branching factors at different levels, as in the proof of Theorem 3.2. We omit the details.

We can also obtain a tradeoff result analogous to Theorem 2.1:

Theorem 5.2. *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. $\square$

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 [11]).

Theorem 6.1. *Given n points in $\mathbb{R}^3$ and $s \ge n^{1/c}\log n$ for a fixed constant c, we can generate the facets of the upper hull by an $O((n/s)\log^c n)$-pass algorithm that uses $O(s)$ space and runs in $O((n^2/s)\log^c n)$ 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 $\{h \in H \mid h$ intersects $\Delta\}$ inside $\Delta$
     and return
  1. take a random sample $R$ of expected size $O(r \log n)$
     of $\{h \in H \mid h$ intersects $\Delta\}$
  2. compute the canonical 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 of the recursion, we have $|H_\Delta| = O(n/r^c)$ w.h.p. Since lower envelopes have linear complexity in $\mathbb{R}^3$, $|T| = O(r \log n)$. 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 be done in one pass. Line 2 takes $O(r \log^2 n)$ time 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. Setting $r = (n/s)^{1/c}$ yields the theorem. $\square$

7. CONCLUDING REMARKS

We have given new algorithms and lower bounds for some basic geometric problems under the multi-pass model. 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 [1] do favor streaming computation with multiple passes. The multi-pass model can also be viewed as a simpler form of external-memory algorithms.

There are many interesting theoretical open problems concerning tradeoffs of space, the number of passes, and running time. 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 [24] 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.

8. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[30] 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.

[31] 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.