A Deterministic Partition Tree and Applications

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, UT, USA

Abstract

In this paper, we present a deterministic variant of Chan's randomized partition tree [Discret. Comput. Geom., 2012]. This result leads to numerous applications. In particular, for d-dimensional simplex range counting (for any constant $d \geq 2$), we construct a data structure using $O(n)$ space and $O(n^{1+\epsilon})$ preprocessing time, such that each query can be answered in $o(n^{1-1/d})$ time (specifically, $O(n^{1-1/d}/\log^{\Omega(1)} n)$ time), thereby breaking an $\Omega(n^{1-1/d})$ lower bound known for the semigroup setting. Notably, our approach does not rely on any bit-packing techniques. We also obtain deterministic improvements for several other classical problems, including simplex range stabbing counting and reporting, segment intersection detection, counting and reporting, ray-shooting among segments, and more. Similar to Chan's original randomized partition tree, we expect that additional applications will emerge in the future, especially in situations where deterministic results are preferred.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases partition trees, simplex range searching, segment intersection queries, ray-shootings, multi-level data structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.114

Related Version *Full Version*: `https://arxiv.org/abs/2507.01775`

Acknowledgements We would like to thank Timothy Chan for the discussions on derandomizing his partition tree.

1 Introduction

Simplex range searching is a fundamental problem in computational geometry. Given a set P of n points in the d-dimensional space $\mathbb{R}^d$ for a constant $d \geq 2$, the goal is to build a data structure so that points of P inside a query simplex can be found efficiently. The problem has been extensively studied (see [1,2,19] for some excellent surveys). For solving the problem with small space (e.g., near linear), one powerful technique is *partition trees*, e.g., [7,14,16–18,23–25]. In particular, using a partition tree Matoušek [17] built a data structure of $O(n)$ space in $O(n \log n)$ time and each simplex range query can be answered in $O(n^{1-1/d} \log^{O(1)} n)$ time. Subsequently Matoušek [18] gave another more complicated data structure of $O(n)$ space with $O(n^{1+\epsilon})$ preprocessing time and $O(n^{1-1/d})$ query time; throughout the paper let ϵ be an arbitrarily small positive constant. Chazelle [10] proved that $\Omega(n^{1-1/d}/\log n)$ (and $\Omega(\sqrt{n})$ for $d = 2$) is a lower bound on the query time for an $O(n)$-space data structure; it is widely believed that the $\log n$ factor is an artifact of the proof. Although the result of [18] seems to achieve optimal query time with linear space, it is not quite satisfactory. One reason is that partition trees are often used as backbone for designing multi-level data structures and certain properties of the result of [18] makes this challenging, e.g., it has a special root of $O(n^{1/d} \log n)$ degree whose children may have overlapping cells and it does not guarantee the optimal crossing number except at the bottom most level of the tree. To address these issues, Chan [7] proposed a randomized partition tree of $O(n)$ space that can be built in $O(n \log n)$ expected time and the query time is bounded by $O(n^{1-1/d})$ w.h.p. Comparing to Matoušek's work [18], Chan's result has many nice properties, e.g., each node has $O(1)$ children, crossing number is optimal (with w.h.p) at almost all layers (except the top few layers), and children's cells at each node are pairwise disjoint. These properties make Chan's partition tree quite amenable to multi-level data structures [7–9].

Simplex range searching has several versions: (1) Counting: compute the number of points of P inside the query simplex Δ; (2) reporting: report all points of P inside Δ; (3) semigroup query (which generalizes the counting query): compute the sum of the weights of the points of P inside Δ, assuming that each point of P is assigned a weight from a semigroup. The above results [7, 17, 18] are applicable to semigroup queries and can also be modified to solve range reporting with an additive term k in the query time, where k is the output size. In particular, Chazelle's lower bound [10] is for the semigroup setting.

Since Chan's partition tree is randomized, for those who need deterministic algorithms, Matoušek's partition trees [17, 18] are still the main resort. In this paper, we "partially" derandomize Chan's partition tree and obtain a deterministic tool, especially for designing multi-level data structures. More specifically, our partition tree is similar to Chan's (e.g., $O(n)$ space, optimal crossing number at all levels except the top few levels, disjointness of children's cells of each node); however, the degree of each node is logarithmic instead of constant (that is why we used "partially" above). Albeit this drawback, our tree is still powerful enough to have many applications, as discussed below.

Simplex range counting. For simplex range counting, we construct a data structure of $O(n)$ space that can answer each query in $O(n^{1-1/d}/\log^{\Omega(1)} n)$ time. Note that this does not violate Chazelle's lower bound [10] as it is for the more general semigroup queries. The preprocessing time is $O(n^{1+\epsilon})$. To achieve the result, we construct our partition tree so that each leaf has $O(\log^{\tau} n)$ points of P for an arbitrarily small constant $\tau > 0$. Because this number is small, we can afford to preprocess all leaves in $O(n)$ time by considering all possible configurations for queries; in this way, each query on a single leaf v can be answered in $O(\log\log n)$ time. While this kind of technique may not be quite surprising, it has never been used in simplex range searching, perhaps because previous work has been focusing on the semigroup queries while this technique does not work in that setting. Note that our above result is also applicable to simplex range emptiness queries.

It should be noted that very recently Chan and Zheng [9] obtained similar randomized results (i.e., $O(n)$ words of space and $o(n^{1-1/d})$ query time w.h.p.) for other data structures and also mentioned a possibility of achieving such result for simplex range counting. One difference is that their technique uses bit-packing by assuming each word has $\Omega(\log n)$ bits (note that the $\log n$-bit word RAM is also a conventional computational model), while ours does not use bit-packing. In addition, their result is randomized while ours is deterministic.

Simplex range stabbing. Given a set of n simplices in $\mathbb{R}^d$, the *simplex range stabbing counting problem* is to build a data structure to compute the number of simplices containing a query point. Using Chan's partition tree, Chan and Zheng [9] built a randomized data structure of $O(n \log\log n)$ space (or $O(n)$ words of space using the bit-packing tricks) in $O(n \log n)$ expected preprocessing time and the query time is $O(n^{1-1/d})$ w.h.p. Using our new deterministic partition tree, we build a deterministic data structure of $O(n \log\log n)$ space in $O(n^{1+\epsilon})$ preprocessing time and the query time is $O(n^{1-1/d}/\log^{\Omega(1)} n)$. Previously, the best deterministic results [17, 18] have $O(n^{1+\epsilon})$ preprocessing time, $O(n \log^{O(1)} n)$ space, and $O(n^{1-1/d} \log^{O(1)} n)$ query time; or $O(n 2^{\sqrt{\log n}})$ preprocessing time and space, and $n^{1-1/d} \cdot 2^{O(\sqrt{\log n})}$ query time. For the *reporting problem* (i.e., report all simplices containing a query point), the randomized data structure of [9] has the same performance as above except that the query time becomes $O(k + n^{1-1/d})$ w.h.p., where k is the output size. We also obtain the same deterministic result as above with $O(k + n^{1-1/d}/\log^{\Omega(1)} n)$ query time.

Segment intersection searching. Given a set of n (possibly intersecting) line segments in the plane, the *segment intersection counting problem* is to build a data structure to compute the number of segments intersecting a query segment. Chan and Zhen [9] built a randomized data structure of $O(n \log \log n)$ space (which again can be reduced to $O(n)$ words of space if bit-packing tricks are allowed) in $O(n \log n)$ expected preprocessing time and the query time is $O(\sqrt{n})$ w.h.p. Using our new deterministic partition tree, we build a deterministic data structure of $O(n \log \log n)$ space in $O(n^{1+\epsilon})$ preprocessing time and the query time is $O(\sqrt{n}/\log^{\Omega(1)} n)$. The previously best deterministic result [6] built a data structure of $O(n \log^2 n)$ space in $O(n^{3/2})$ time that can answer each query in $O(\sqrt{n} \log n)$ time.

For the *reporting problem* (i.e., report all segments intersecting a query segment), the randomized data structure of [9] has the same performance as above except that the query time becomes $O(k + \sqrt{n})$ w.h.p., where k is the output size. We also obtain the same deterministic result as above with $O(k + \sqrt{n}/\log^{\Omega(1)} n)$ query time.

Segment intersection detection. Given n (possibly intersecting) line segments in the plane, we wish to build a data structure to decide whether a query line intersects any segment [12,22]. The previous deterministic result [22] builds an $O(n)$-space data structure in $O(n^{3/2})$ time, with $O(\sqrt{n} \log n)$ query time. Using our new partition tree, we build an $O(n)$-space data structure with $O(\sqrt{n}/\log^{\Omega(1)} n)$ query time and $O(n^{1+\epsilon})$ preprocessing time.

Ray-shooting among non-intersecting segments. Given n non-intersecting line segments in the plane, we wish to build a data structure to find the first segment hit by a query ray [3,6,20]. The previous best deterministic result [22] builds an $O(n)$-space data structure in $O(n^{3/2})$ time, with $O(\sqrt{n} \log n)$ query time. With our partition tree, we build an $O(n)$-space data structure with $O(\sqrt{n}/\log^{\Omega(1)} n)$ query time and $O(n^{1+\epsilon})$ preprocessing time.

If the segments are allowed to intersect, then the problem has also been studied [3,4,6,7, 9,12,15,20,22]. The previously best deterministic result [22] builds an $O(n \log n)$-space data structure in $O(n^{3/2})$ time and each query can be answered in $O(\sqrt{n} \log n)$ time.

Outline. After introducing notation in Section 2, we present our partition tree in Section 3. The simplex range counting problem is treated in Section 4. The simplex range stabbing and the segment intersection searching problems are discussed in Section 5. We solve the segment intersection detection and the ray-shooting problems in Sections 6 and 7, respectively. Due to the space limit, many proofs and details are omitted but can be found in the full paper.

2 Preliminaries

Let H be a set of n hyperplanes in $\mathbb{R}^d$. We use $\mathcal{A}(H)$ to denote the arrangement of H, which can be computed in $O(n^d)$ time [13]. For a compact region $R \in \mathbb{R}^d$, we use H_R to denote the subset of hyperplanes of H that intersect the relative interior of R but does not contain R (we also say that these hyperplanes *cross* R). A *cutting* for H is a collection Ξ of closed cells (each of which is a simplex, possibly unbounded) with disjoint interiors, which together cover the entire space $\mathbb{R}^d$ [11,18]. The *size* of Ξ is the number of cells of Ξ. For a parameter r with $1 \le r \le n$, a $(1/r)$-*cutting* for H is a cutting Ξ satisfying $|H_\sigma| \le n/r$ for every cell $\sigma \in \Xi$.

For any $1 \le r \le n$, a $(1/r)$-cutting of size $O(r^d)$ for H can be computed in $O(nr^{d-1})$ time [11]. We further have Lemma 1 (similar results were mentioned before [5, 7, 21]). Throughout the paper, β always refers to the one in the lemma.

▶ **Lemma 1.** (Cutting Lemma) *Let H be a set of n hyperplanes and Δ a simplex in $\mathbb{R}^d$. For any $1 \le r \le n$, we can compute a $(1/r)$-cutting of $O(K \cdot (r/n)^d + r^{d-1+\beta})$ cells for H whose union is Δ in $O(K \cdot (r/n)^{d-1} + nr^{d-2+\beta})$ time, where K is the number of vertices of $\mathcal{A}(H)$ inside Δ and β is an arbitrarily small positive constant.*

Proof. We apply Chazelle's algorithm [11] but only on the region inside Δ (i.e., starting with $C_0 = \Delta$ following the notation of [11]). A detailed analysis for the 2D case is given in [21, Appendix A]. Below we sketch how to modify the analysis in [11] accordingly by following the notation there.

The analysis follows the same approach except that we use K to replace $\binom{n}{d}$ in the formula $\sum_{s \in C_{k-1}} v(H_{|s}; s) \le \binom{n}{d}$ in [11, Page 153]. As such, the subsequent formula becomes

$$|C_k| \le c\left(\frac{r_0^k \log r_0}{n}\right)^d \cdot K + cr_0^{d-1}(\log r_0)^d |C_{k-1}|.$$

Then, one can prove by induction that $|C_k| \le r_0^{d(k+1)} \cdot K/n^d + r_0^{(k+1)\cdot(d-1+\beta)}$, for an arbitrarily small constant $\beta > 0$. Therefore, the total number of cells of the cutting inside Δ is as stated in the lemma.

For the time analysis, following the same formula $\sum_{0 \le k \le \lceil \log_{r_0} r \rceil} \frac{n}{r_0^k} |C_k|$ in [11, Page 153] and using the above inequality for $|C_k|$, we can derive the time complexity as stated in the lemma. ◀

3 Deterministic partition tree

In this section, we present our deterministic partition tree. The following lemma "partially" derandomizes Chan's partition refinement theorem (i.e., Theorem 3.1 [7]).

▶ **Lemma 2.** *Let P be a set of n points and H a set of m hyperplanes in $\mathbb{R}^d$. Suppose there are t interior-disjoint cells whose union covers P, such that each cell contains at most $2n/t$ points of P and each hyperplane crosses at most κ cells. Then, for any $b \ge 4$, we can divide every cell into $O(b)$ disjoint subcells each containing at most $2n/(bt)$ points of P, for a total of at most bt subcells, so that the total number of subcells crossed by any hyperplane in H is bounded by*

$$O((b \cdot t)^{1-1/d} + b^{1-1/(d-1+\beta)} \cdot \kappa \cdot \log t + b \cdot \log m). \tag{1}$$

Remark. Comparing to Chan's partition refinement theorem, there is an extra $\log t$ in the second term of (1). As will be seen next, due to this extra factor, we have to make each node of our partition tree have logarithmically many children instead of constant. That is why we said before that we "partially" derandomized Chan's result.

3.1 Proving Lemma 2

This subsection is devoted to the proof of Lemma 2.

Let S denote the set of all t given cells in Lemma 2. Let H' be a multiset containing C copies of each hyperplane in H, for a parameter C that is a sufficiently large power of b (so that all future multiplicities are integers; the actual value of C is not important). For any multiset H'' of H, the *size* of H'', denoted by $|H''|$, is the sum of the multiplicities of the hyperplanes in H''. For a cell Δ, let $N_\Delta(H'')$ denote the number of vertices of the arrangement $\mathcal{A}(H'')$ of H'' inside Δ, counting multiplicities (note that the multiplicity of a vertex is the product of the multiplicities of its defining hyperplanes); let $H''(\Delta)$ denote the (multi)-subset of hyperplanes of H'' crossing Δ.

We process the t cells of S iteratively one by one. The order they will be processed is carefully chosen (in contrast, a random order is used in the proof of [7], which is a major difference between our proof and that in [7]). We will assign indices to the cells following the reverse order they are processed. Suppose we have processed $t-i$ cells, which have been assigned indices and denoted by $\Delta_t, \Delta_{t-1}, \ldots, \Delta_{i+1}$. In the next iteration, we will find a cell among the unprocessed cells of S and process it (and the cell will be assigned index i as Δ_i). Suppose we now have a multiset H_i after cell Δ_{i+1} is processed (initially let $H_{t+1} = H'$). Let S_i denote the subset of unprocessed cells of S. Hence, $|S_i| = i$.

For each cell $\Delta \in S_i$, define

$$A_\Delta = \left(\frac{N_\Delta(H_i)}{b}\right)^{1/d}, \quad B_\Delta = \frac{|H_i(\Delta)|}{b^{1/(d-1+\beta)}}.$$

Define S_{i1} as the subset of cells $\Delta \in S_i$ with $A_\Delta \geq B_\Delta$. Let $S_{i2} = S \setminus S_{i1}$. If $|S_{i1}| \geq i/2$, then we define Δ_i as the cell Δ of S_{i1} that minimizes the value A_Δ; otherwise define Δ_i as the cell Δ of S_{i2} that minimizes B_Δ. Note that the above way of defining Δ_i is a key difference from Chan's approach [7], where Δ_i is chosen from S_i randomly. We now process Δ_i in the following three steps (which is similar to Chan's approach).

1. Construct a $(1/r_i)$-cutting for H_i inside Δ_i with

$$r_i = c \cdot \min\left\{ |H_i(\Delta_i)| \cdot \left(\frac{b}{N_{\Delta_i}(H_i)}\right)^{1/d}, b^{1/(d-1+\beta)} \right\},$$

for some constant c. By the cutting lemma, the number of subcells inside Δ_i is $O(N_{\Delta_i}(H_i) \cdot (r_i/|H_i(\Delta_i)|)^d + r_i^{d-1+\beta})$, which can be made at most $b/4$ for a sufficiently small c.

2. We further subdivide each subcell of Δ_i (e.g., using vertical cuts) so that each subcell contains at most $2n/(tb)$ points of P. The number of extra cuts is $O(b)$ as Δ_i contains at most $2n/t$ points of P. The total number of extra cuts for processing all t cells of S is at most $\frac{n}{2n/(tb)} + t = bt/2 + t$, and thus the total number of subcells after processing all t cells is at most $bt/4 + bt/2 + t = 3bt/4 + t$, which is at most bt for $b \geq 4$.

3. For each distinct hyperplane $h \in H_i$, multiply the multiplicity of h in H_i by $(1+1/b)^{\lambda_i(h)}$, where $\lambda_i(h)$ is the number of subcells of Δ_i crossed by h. Let H_{i-1} be the resulting multiset after this.

To prove Lemma 2, it remains to prove Bound (1).

In the third step, since each of the $O(b)$ subcells of Δ_i is crossed by at most $|H_i(\Delta_i)|/r_i$ hyperplanes of H_i, we have $\sum_{h \in H_i} \lambda_i(h) = O(b \cdot |H_i(\Delta_i)|/r_i)$. Since $\lambda_i(h) = O(b)$, we have

$$|H_{i-1}| = \sum_{h \in H_i} (1+1/b)^{\lambda_i(h)} = \sum_{h \in H_i} (1 + O(\frac{\lambda_i(h)}{b})) = |H_i| + \sum_{h \in H_i} O(\frac{\lambda_i(h))}{b}).$$

We thus obtain

$$|H_{i-1}| - |H_i| = O(\frac{1}{b} \cdot \sum_{h \in H_i} \lambda_i(h)) = O(|H_i(\Delta_i)|/r_i) = O(\alpha_i) \cdot |H_i|, \text{ where}$$

$$\begin{aligned}\alpha_i = \frac{|H_i(\Delta_i)|}{r_i \cdot |H_i|} &= O\left(\frac{1}{|H_i|} \cdot \max\left\{ \left(\frac{N_{\Delta_i}(H_i)}{b}\right)^{1/d}, \frac{|H_i(\Delta_i)|}{b^{1/(d-1+\beta)}} \right\}\right) \\ &= O\left(\frac{1}{|H_i|} \cdot \max\{A_{\Delta_i}, B_{\Delta_i}\}\right).\end{aligned}$$

After all t cells of S are processed, we have a multiset H_0. Recall that $H_{t+1} = H'$ and $|H'| = C \cdot m$. According to the above analysis, we have

$$|H_0| = |H_{t+1}| \cdot \Pi_{i=1}^{t}(1 + O(\alpha_i)) \leq C \cdot m \cdot \exp\left(O(\sum_{i=1}^{t} \alpha_i)\right). \tag{2}$$

The following lemma gives an upper bound for $\sum_{i=1}^{t} \alpha_i$.

▶ **Lemma 3.** $\sum_{i=1}^{t} \alpha_i = O\left(\frac{t^{1-1/d}}{b^{1/d}} + \frac{\kappa \cdot \log t}{b^{1/(d-1+\beta)}}\right)$.

By Lemma 3 and (2), we have

$$|H_0| \leq C \cdot m \cdot \exp\left(O\left(\frac{t^{1-1/d}}{b^{1/d}} + \frac{\kappa \cdot \log t}{b^{1/(d-1+\beta)}}\right)\right). \tag{3}$$

For any hyperplane $h \in H$, let $\lambda(h)$ be the total number of subcells crossed by h. Our goal is to prove that $\lambda(h)$ is bounded by (1). By the way H_0 is produced, we have $C \cdot (1+1/b)^{\lambda(h)} \leq |H_0|$. Hence,

$$\lambda(h) \leq \log_{1+1/b} \frac{|H_0|}{C} = O(b \cdot \log \frac{|H_0|}{C}) = O(b \log m + (bt)^{1-1/d} + b^{1-1/(d-1+\beta)} \cdot \kappa \cdot \log t).$$

This proves Lemma 2.

3.2 Constructing the partition tree

Using Lemma 2, we can construct a partition tree in the following theorem.

▶ **Theorem 4.** (Hierarchical Partition Theorem) *Given a set P of n points in $\mathbb{R}^d$ and a parameter r with $r_0 \leq r \leq n$ for a sufficiently large constant r_0, for $b = \log^{\rho} r$ for a sufficiently large constant $\rho > 0$, there exists a sequence $\Pi_0, \Pi_1, \ldots, \Pi_{k+1}$ with $k = \lfloor \log_b r \rfloor$ and $b' = \lceil r/b^k \rceil$, where each Π_i is a collection of disjoint simplicial cells (i.e., each cell is a simplex) whose union covers P with the following properties:*

1. *Π_0 has only one cell that is $\mathbb{R}^d$ and the number of cells of Π_i is $O(b' \cdot b^{i-1})$ for $i \geq 1$ (in particular, Π_{k+1} has $O(r)$ cells; the total number of cells in all collections is also $O(r)$).*
2. *Each cell of Π_i contains at least one point and at most $2n/(b' \cdot b^{i-1})$ points of P for all $i \geq 0$ (in particular, each cell of Π_{k+1} contains at most $2n/r$ points).*
3. *Each cell in Π_{i+1} is contained in a single cell of Π_i.*
4. *Each cell of Π_i contains $O(b)$ cells of Π_{i+1} for $i \geq 1$ and the cell of Π_0 contains $O(b')$ cells of Π_1.*
5. *Any hyperplane crosses at most $O((b' \cdot b^{i-1})^{1-1/d} + \log^{O(1)} n)$ cells of Π_i for all $i \geq 1$.*

All collections of cells can be constructed in $O(n^{d^2} \cdot (r^{2-2/d} + \log^{O(1)} n))$ time.

Proof. We only sketch the proof here. The details can be found in the full paper.

Let H be a set of $m = n^{O(1)}$ hyperplanes in $\mathbb{R}^d$. We can prove the following *critical lemma*: With respect to H, one can construct a sequence $\Pi_0, \Pi_1, \ldots, \Pi_{k+1}$ as stated in the theorem with the same properties except that Property (5) only works for any hyperplane in H; also, the construction time is bounded by $O(m^d \cdot (r^{2-2/d} + \log^{O(1)} n) + n \log n + r^{3-1/d} + r^2 \cdot \log^{O(1)} n + m \cdot r^{3-2/d})$.

To prove the above lemma, we first assume that r is a power of b. Then, $r = b^k$ and $b' = 1$. We apply Lemma 2 with H to construct the collections Π'_t iteratively for $t = 1, b, b^2, \ldots$ until $b^k = r$. More specifically, Π'_1 consists of a single cell that is $\mathbb{R}^d$, and Π'_{bt} is obtained by applying Lemma 2 on all cells of Π'_t. We then let $\Pi_0 = \Pi_1 = \Pi'_1$ and $\Pi_i = \Pi'_{b^{i-1}}$. By

Lemma 2, the properties (1)-(4) hold. The proof for Property (5) as well as the case where r is not a power of b are given in the full paper. The algorithm implementation for constructing all collections of cells is also presented in the full paper.

The above critical lemma only guarantees the crossing numbers for the hyperplanes in H. To make it work for any hyperplane in $\mathbb{R}^d$, we resort to the following lemma, which was known before, e.g., [7].

▶ **Lemma 5.** (Test Set Lemma [7]) *For a set P of n points in $\mathbb{R}^d$, we can compute in $O(n^d)$ time a set H (called* test set*) of $O(n^d)$ hyperplanes with the following property: for any set of cells each containing at least one point of P, if κ is the maximum number of cells crossed by any hyperplane of H, then the maximum number of cells crossed by any hyperplane is $O(\kappa)$.*

Now to prove Theorem 4, we just apply the above critical lemma with H as the test set given in Lemma 5, which can be computed in $O(n^d)$ time. Since $m = O(n^d)$ and $r \leq n$, we obtain the theorem from the above critical lemma. ◀

The collections of cells of Theorem 4 naturally form a tree structure, called a *hierarchical partition tree*, in which each node corresponds to a cell. Specifically, the only cell in Π_0 is the root. Cells of Π_{k+1} are the leaves. If a cell $\Delta \in \Pi_i$ contains another cell $\Delta' \in \Pi_{i+1}$, then Δ is the parent of Δ' and Δ' is a child of Δ. As such, each node of the tree has $O(b)$ children (the root has $O(b')$ children). Although the construction time of Theorem 4 is large, it can usually be reduced (e.g., to $O(n^{1+\epsilon})$ time) in applications by constructing an "upper" partition tree using other techniques (e.g., [17]) and then processing the leaves (each containing a small number of points) using Theorem 4, as will be demonstrated in the subsequent sections.

4 Simplex range counting

Given a set P of n points in $\mathbb{R}^d$, we wish to construct a data structure so that the number of points of P inside a query simplex can be quickly computed. If we set $r = n$ in Theorem 4, we can have a data structure of $O(n)$ space and $O(b \cdot n^{1-1/d})$ query time, which is not $O(n^{1-1/d})$ as $b = \Theta(\log n)$. In the following, we reduce the query time to $O(n^{1-1/d}/\log^{\Omega(1)} n)$.

For a region R in the plane, let $P(R)$ denote the subset of points of P in R.

Setting $r = n/b^{2d/(d-1)}$ with $b = \log^{\rho} n$, we apply Theorem 4 to obtain a partition tree T with collections Π_i, $0 \leq i \leq k+1$. The total number of cells is $O(r)$. For each cell Δ, we store $|P(\Delta)|$. By Theorem 4, each cell in Π_{k+1} contains $O(n/r) = O(b^{2d/(d-1)}) = O(\log^{2d\rho/(d-1)} n)$ points of P. The runtime to construct T is $O(n^{d^2+2})$.

Given a query simplex σ, starting from the root of T, for each cell Δ of T, if Δ is contained in σ, then we add $|P(\Delta)|$ to a total count. If Δ is completely outside σ, then we ignore it. Otherwise, a bounding halfplane of σ must cross Δ and we proceed on all children of Δ unless Δ is a leaf. In this way, we obtain a set V of $O(r^{1-1/d})$ leaf cells that are crossed by the bounding halfplanes of σ. The runtime is $O((r/b)^{1-1/d} \cdot b)$, which is $O(r^{1-1/d} \cdot b^{1/d})$.

We now build the data structure recursively on the points in $P(\Delta)$ for each leaf cell Δ of T. If $Q(n)$ is the query time, we have the following recurrence relation:

$$Q(n) = O(r^{1-1/d} \cdot b^{1/d}) + O(r^{1-1/d}) \cdot Q_1(n/r), \tag{4}$$

where $Q_1(\cdot)$ is the query time for each leaf cell Δ, which contains $O(n/r)$ points of P.

To solve $Q_1(n/r)$, we preprocess $P(\Delta)$ for each leaf cell $\Delta \in T$ recursively as above by using different parameters. Specifically, let $n_1 = |P(\Delta)|$, which is $O(n/r)$. Let $\tau > 0$ be an arbitrarily small constant to be set later. Setting $r_1 = n_1/\log^{\tau} n$ with $b_1 = \log^{\rho} n_1$, we apply Theorem 4 to construct a partition tree $T(\Delta)$ in $O(n_1^{d^2+2})$ time. The total time for constructing $T(\Delta)$ for all leaf cells Δ of T is bounded by $O(n^{d^2+2})$. Using $T(\Delta)$ to handle queries on $P(\Delta)$ and following the same analysis as above, we obtain

$$Q_1(n_1) = O(r_1^{1-1/d} \cdot b_1^{1/d}) + O(r_1^{1-1/d}) \cdot Q_2(n_1/r_1), \tag{5}$$

where $Q_2(\cdot)$ is the query time for each leaf of $T(\Delta)$, which contains $O(n_1/r_1)$ points of P.

Combining (4) and (5) leads to (see the full paper for the detailed proof):

$$\begin{aligned} Q(n) &= O(r^{1-1/d} \cdot b^{1/d}) + O(r^{1-1/d}) \cdot Q_1(n/r) \\ &= O(r^{1-1/d} \cdot b^{1/d}) + O(r^{1-1/d} \cdot r_1^{1-1/d} \cdot b_1^{1/d}) + O(r^{1-1/d} \cdot r_1^{1-1/d}) \cdot Q_2(n_1/r_1) \\ &= O\left(\frac{n^{1-1/d}}{\log^{\Omega(1)} n}\right) + O\left(\left(\frac{n}{t}\right)^{1-1/d}\right) \cdot Q_2(t), \text{ where } t = \log^{\tau} n. \end{aligned} \tag{6}$$

In summary, the above first builds a partition tree T and then builds a partition tree $T(\Delta)$ for each leaf Δ of T (so there are two recursive steps but with different parameters). For notational convenience, we still use T to refer to the entire tree (by attaching $T(\Delta)$ for all leaves Δ), which has $O(n/t)$ leaves, each containing $O(t)$ points. The total space is bounded by $O(n)$ since there are only two recursive steps. In Section 4.1, by using the property that t is very small, we show that after $O(n)$ space and $O(n \log n)$ time preprocessing, each simplex range counting query on any leaf cell of T can be answered in $O(\log t)$ time (i.e., $Q_2(t) = O(\log t)$, which is $O(\log \log n)$; we consider the query on each leaf cell a *subproblem*). As such, we obtain that $Q(n) = O(n^{1-1/d}/\log^{\Omega(1)} n)$. The total preprocessing time is $O(n^{d^2+2})$, dominated by the time for constructing T. We thus have the following result.

▶ **Lemma 6.** *Given a set P of n points in $\mathbb{R}^d$, there is a data structure of $O(n)$ space that can compute the number of points of P in any query simplex in $O(n^{1-1/d}/\log^{\Omega(1)} n)$ time. The data structure can be built in $O(n^{d^2+2})$ time.*

We will reduce the preprocessing time to $O(n^{1+\epsilon})$ in Section 4.2.

4.1 Solving the subproblems

We first consider halfspace queries and then extend the techniques to the simplex case.

A basic data structure. Let A be a set of t points in $\mathbb{R}^d$. We first build a straightforward data structure (called a *basic data structure*) of $O(t^d)$ space in $O(t^{d+1})$ time that can answer each halfspace range counting query on A in $O(\log t)$ time.

Let H be the set of dual hyperplanes of A. We compute the arrangement $\mathcal{A}$ of H in $O(t^d)$ time and space [13]. We then build a point location data structure on $\mathcal{A}$, which can be done in $O(t^d)$ time and supports $O(\log t)$-time point location queries [11]. In addition, for each face f of $\mathcal{A}$, we compute the number of hyperplanes above f (resp., below f) and store these two numbers at f, e.g., by checking every hyperplane of A. This finishes our preprocessing, which can be easily done in $O(t^{d+1})$ time and $O(t^d)$ space. The preprocessing time can be reduced to $O(t^d)$, e.g., by taking an Eulerian tour of the dual graph of the arrangement.

Given a query halfspace σ, the goal is to compute the number of points of A inside σ. Without loss of generality, we assume that σ is an upper halfspace. Then, it is equivalent to computing the number of hyperplanes of H below p, where p is the dual point of the bounding hyperplane of σ. Using the point location data structure, we find the face f of $\mathcal{A}$ that contains p; f stores the number of hyperplanes of H below it and we return that number as our answer to the query. The query time is thus $O(\log t)$.

Handling halfspace queries. Next, we show that after $O(n \log n)$ time and $O(n)$ space preprorcessing we can answer each halfspace range counting query in $O(\log t)$ time on $P(\Delta)$ for each leaf cell Δ of our partition tree T.

We build an algebraic decision tree T_D for the arrangement construction algorithm [13] on a set of t hyperplanes in $\mathbb{R}^d$ so that each node of T_D corresponds to a comparison in the algorithm. The height of T_D is $O(t^d)$ and T_D has $2^{O(t^d)}$ leaves. Each leaf v of T_D corresponds to a "configuration" of t hyperplanes in the following sense. Let A be a set of t hyperplanes with indices $1, 2, \ldots, t$. Following T_D in a top-down manner, we can reach a leaf v such that all comparisons of the nodes in the path of T_D from the root to v are consistent with A; we say that A has the same configuration as v. Let A and B are two sets of t hyperplanes each. Let $\mathcal{A}_A$ and $\mathcal{A}_B$ be the arrangements of A and B, respectively. If the configurations of A and B are both the same as a leaf v of T_D, then there is a one-to-one correspondence between faces of $\mathcal{A}_A$ and faces of $\mathcal{A}_B$ such that if a face f of $\mathcal{A}_A$ corresponds to a face f' of $\mathcal{A}_B$, then the i-th hyperplane of A is above f if and only if the i-th hyperplane of B is above f'.

By the above observation, we do the following preprocessing. For each leaf v of T_D, let A_v be a set of t hyperplanes whose configuration corresponds to v (note that the sets A_v's for all leaves v can be constructed in $t^d \cdot 2^{O(t^d)}$ time by following T_D, which basically enumuerates all possible configurations for the arrangements of a set of t hyperplanes). We construct the above basic data structure on A_v, denoted by $\mathcal{D}_v$. Doing this for all leaves of T_D takes $t^d \cdot 2^{O(t^d)}$ time and space, which is $O(n)$ since $t = \log^\tau n$ if we choose a small enough τ.

For each leaf cell Δ of T, recall $|P(\Delta)| \leq t$ (if $|P(\Delta)| < t$, we add $t - |P(\Delta)|$ dummy points to $P(\Delta)$ so that $P(\Delta)$ has exactly t points). Let $H(\Delta)$ be the set of dual hyperplanes of the points of $P(\Delta)$. We arbitrarily assign indices to the hyperplanes of $H(\Delta)$. Following T_D, we find the leaf v of T_D that has the same configuration as $H(\Delta)$, which can be done in time linear in the height of T_D, i.e., $O(t^d)$; we associate v with Δ. Since T has $O(n/t)$ leaves, doing this for all leaves of T takes $O(n/t \cdot t^d)$ time, which is $O(n \log n)$ if τ is small enough.

Given a query halfspace σ, suppose we want to compute the number of points of $P(\Delta)$ inside σ for a leaf cell Δ of T. This can be done in $O(\log t)$ time as follows. Let v be the leaf of T_D associated with Δ. Let p be the dual point of the bounding hyperplane of σ. Without loss of generality, we assume that σ is an upper halfspace. Hence it is equivalent to finding the number of hyperplanes of $H(\Delta)$ below p. We apply the point location query algorithm using the data structure $\mathcal{D}_v$ with p, but whenever the algorithm attempts to use the i-th hyperplane of A_v to make a comparison, we use the i-th hyperplane of $H(\Delta)$ instead. The point location algorithm will eventually return a face, which stores the number of hyperplanes below it; we return that number as our answer to the query of σ.

Handling simplex queries. We can easily extend the above idea to simplex queries, by using a multilevel data structure. The details are given in the full paper. In summary, with $O(n)$ space and $O(n \log n)$ time preprocessing, a simplex range counting query on $P(\Delta)$ for any leaf cell Δ of T can be answered in $O(\log t)$ time.

4.2 Reducing the preprocessing time

We now reduce the preprocessing time of Lemma 6 to $O(n^{1+\epsilon})$. The idea is to build an "upper partition tree" of $O(1)$ depth using Matoušek's method [17] so that each leaf has $O(n^\delta)$ points of P for a small constant $\delta > 0$ and then construct our data structure in Lemma 6 on each leaf (which form the "lower" part of the partition tree). The details are given below.

A *simplicial partition* for P is a collection $\Pi = \{(P_1, \sigma_1), (P_2, \sigma_2), \ldots, (P_m, \sigma_m)\}$, where the P_i's are pairwise disjoint subsets forming a partition of P, and each σ_i is a simplex containing all points of P_i. The subsets P_i's are called the *classes* and the simplices σ_i's are called *cells*, which may overlap. The *crossing number* of Π is the maximum number of cells crossed by any hyperplane. The following result is from [17].

▶ **Lemma 7.** ([17]) *For a set P of n points in $\mathbb{R}^d$ and a parameter $r \le n^{\epsilon'}$ for any constant $\epsilon' < 1$, a simplicial partition $\Pi = \{(P_1, \sigma_1), (P_2, \sigma_2), \ldots, (P_{O(r)}, \sigma_{O(r)})\}$ whose classes satisfy $|P_i| \le n/r$ and whose crossing number is $O(r^{1-1/d})$ can be constructed in $O(n \log r)$ time.*

We build a partition tree T by Lemma 7 recursively, until we obtain a partition of P into subsets of sizes $O(n^\delta)$ for a small enough constant $\delta > 0$ to be fixed later, which form the leaves of T. Each inner node v of T corresponds to a subset P_v of P as well as a simplicial partition Π_v of P_v, which form the children of v. At each child u of v, we store the cell σ_u of Π_v containing P_u and also store the cardinality $|P_u|$. The simplicial partition Π_v is constructed using Lemma 7 with $r = n^{(1-\delta)/k}$ for a constant integer k to be fixed later, i.e., every internal node of T has $O(r)$ children. If we recurse k times, i.e., the depth of T is k, then the subset size of each leaf of T is $O(n^\delta)$. Hence, the number of leaves of T is $O(r^k)$. Next, for each leaf v of T, we construct the data structure of Lemma 6 on P_v, denoted by $\mathcal{D}_v$, in $O(|P_v|^{d^2+2})$ time. Since $|P_v| = O(n^\delta)$, we can make δ small enough so that the total time of Lemma 6 on all leaves of T is $O(n^{1+\epsilon})$. This finishes the preprocessing, which takes $O(n^{1+\epsilon})$ time. The space of the data structure is $O(n)$ since the depth of T is $O(1)$.

Given a query simplex σ, starting from the root of T, for each node v, we check whether σ contains the cell σ_v. If yes, we add $|P_v|$ to the total count. Otherwise, if the boundary of σ crosses σ_v, we proceed to the children of v. In this way, we reach a set V of leaves v of T whose cells σ_v are crossed by the bounding hyperplanes of σ. Since the number of leaves of T is $O(r^k)$ and the depth of T is k, which is a constant, the size of V is $O(r^{k\cdot(1-1/d)})$. If $Q(m)$ is the query time for a subset of size m, then we have the following recurrence

$$Q(m) = O(r) + O(r^{1-1/d}) \cdot Q(m/r), \text{ where } r = n^{(1-\delta)/k} \text{ with } n \text{ as the global input size.}$$

Starting with $m = n$ and recursing k times (using the same r) gives us

$$Q(n) = O(r^{(k-1)\cdot(1-1/d)+1}) + O(r^{k\cdot(1-1/d)}) \cdot Q(n/r^k).$$

Using $r = n^{(1-\delta)/k}$, by setting k to a constant integer larger than $(1/\delta - 1)/(d-1)$, we have

$$Q(n) = O(n^{1-1/d-\delta'}) + O(r^{k\cdot(1-1/d)}) \cdot Q(n/r^k), \tag{7}$$

for another small constant $\delta' > 0$.

Finally, for each leaf node $v \in V$ (i.e., those subproblems $Q(n/r^k)$ in (7)), we use the data structure $\mathcal{D}_v$ to compute the number of points of P_v in σ, in $O(|P_v|^{1-1/d}/\log^{\Omega(1)}|P_v|)$ time by Lemma 6, which is $O((n/r^k)^{1-1/d}/\log^{\Omega(1)} n)$ as $|P_v| = O(n/r^k)$. Plugging this into (7) gives us $Q(n) = O(n^{1-1/d}/\log^{\Omega(1)} n)$. We thus conclude as follows.

▶ **Theorem 8.** *Given a set of n points in $\mathbb{R}^d$, there is a data structure of $O(n)$ space that can compute the number of points in any query simplex in $O(n^{1-1/d}/\log^{\Omega(1)} n)$ time. The data structure can be built in $O(n^{1+\epsilon})$ time for any $\epsilon > 0$.*

5 Simplex range stabbing and segment intersection searching

Given a set S of n simplices in $\mathbb{R}^d$, the problem is to construct a data structure to compute the number of simplices that contain a query point (we also say that those simplices are *stabbed* by the query point). In their data structure, Chan and Zheng [9] utilized a randomized hierarchical partition. We follow their approach and instead use our deterministic hierarchical partition in Theorem 4. Extra effort needs to be taken because the degree of the partition tree in [7] is $O(1)$ while ours is logarithmic. In the following, we first briefly review Chan and Zheng's approach [9] and then discuss how to make changes.

A review of Chan and Zheng's randomized approach [9]. Each simplex is bounded by $d+1$ hyperplanes. A point p stabs a simplex s if p is in the "correct" side of every bounding hyperplane of s (to simplify the discussion, we assume these are lower halfplanes). In the dual space, this is equivalent to having the dual hyperplane of p above the dual point of each bounding hyperplane of s. Therefore, we have the following problem in the dual space. Given a set S^* of n $(d+1)$-tuples of points in $\mathbb{R}^d$, we wish to construct a data structure to compute the number of tuples whose points are all below a query hyperplane p^*. We can solve the problem using a multi-level data structure as follows.

Let P be the set of the $(d+1)$-th points of all tuples of S^*. Apply the simplicial partition of Lemma 7 to P to obtain a partition $\Pi = \{(P_i, \sigma_i)\}$, with a parameter r to be fixed later. Given a query hyperplane p^*, for every cell σ_i of Π that is crossed by p^*, we recurse on the subset of tuples whose $(d+1)$-th points are in P_i (i.e., apply Lemma 7 on P_i recursively). For each cell σ_i completely below p^*, we recurse on the subset of tuples whose $(d+1)$-th points are in P_i but as a level-d problem (the original problem is a level-$(d+1)$ problem; the recurrence stops at a level-0 problem in which we only need to return the size of the subset). As analyzed in [7], choosing $r = n^\epsilon$ with n as the global input size (i.e., value r is fixed for all levels of the recursion and this makes the recursion depth $O(1)$) can obtain an $O(n)$ space data structure with $O(n^{1-1/d+\epsilon})$ query time, for any $\epsilon > 0$. The preprocessing time is $O(n \log n)$. We summarize this (deterministic) result in the lemma below.

▶ **Lemma 9.** ([9]) *Given a set of n simplices in $\mathbb{R}^d$, one can build a data structure of $O(n)$ space in $O(n \log n)$ time so that the number of simplices containing a query point can be computed in $O(n^{1-1/d+\epsilon})$ time, for any $\epsilon > 0$.*

To improve the query time, Chan and Zheng [9] reduced the problem to a special case where the first d points in each tuple of S^* all lie in $\mathbb{R}^{d-1}$ (equivalently, in the primal setting all but one bounding halfplanes of each input simplex are parallel to the d-th axis). As such, we can reduce some subproblems in the recurrence to the $(d-1)$-dimensional problem and then solve these subproblems by Lemma 9 (where d becomes $d-1$). If $Q(n)$ (resp., $Q'(n)$) is the query time for the problem in $\mathbb{R}^d$ (resp., $\mathbb{R}^{d-1}$), then we have the following recurrence:

$$Q(n) = O(r^{1-1/d}) \cdot Q(n/r) + O(r) \cdot Q'(n/r). \tag{8}$$

By Lemma 9, $Q'(n) = O(n^{1-1/(d-1)+\epsilon})$. If we choose $r = n^\delta$ for a small constant $\delta > 0$, the recursion depth is $O(\log \log n)$ and thus the space is $O(n \log \log n)$. The query time is $O(n^{1-1/d} \log^{O(1)} n)$ due to a constant-factor blowup. To improve the query time to $O(n^{1-1/d})$, a randomized hierarchical partition was used in [9] to replace all recursive simplicial partitions in the preprocessing algorithm for Lemma 9. Here to achieve a deterministic result, we instead use our deterministic hierarchical partition in Theorem 4, as follows.

Our deterministic result. With $r = n/b^c$ for a sufficiently large constant c and $b = \log^\rho n$, we apply Theorem 4 to P to obtain a partition tree T consisting of Π_j, $0 \le j \le k+1$. By Theorem 4, Π_j has $O(b' \cdot b^{j-1})$ cells and each cell has at most $2n/(b' \cdot b^{j-1})$ points of P. We define a sequence of numbers $t_0 < t_1 < \cdots < t_l$ as follows. For each i, $0 \le i \le l$, define $t_i = b' \cdot b^{j_i - 1}$ (and thus $|\Pi_{j_i}| = O(t_i)$), with $j_0 = 1$ and

$$j_i = \left\lceil \log_b \frac{r^{1-(1-\epsilon)^i}}{b'} \right\rceil + 1, \quad 1 \le i \le l,$$

where l is the smallest integer so that $r/b < b' \cdot b^{j_l - 1}$. Hence, $l = O(\log\log r)$, which is $O(\log\log n)$. Note that $b' \cdot b^{j_l - 1} \le r$ always holds. By definition, $t_0 = b'$ and t_i is equal to $r^{1-(1-\epsilon)^i}$ within a factor of b for $i \ge 1$.

We replace the recursive partitions in the preprocessing algorithm of Lemma 9 with this sequence of partitions of T: $\Pi_0, \Pi_{j_0}, \Pi_{j_1}, \ldots, \Pi_{j_l}$. As $l = O(\log\log n)$, the space of the data structure is still $O(n\log\log n)$. The cells of the last partition Π_{j_l} will be further preprocessed later. The following lemma analyzes the query time.

▶ **Lemma 10.** *The query time excluding the time spent on the cells of* Π_{j_l} *is bounded by* $O(b \cdot r^{1-1/d} \cdot (n/r)^{1-1/(d-1)+\epsilon})$.

Lemma 10 leads to the following.

▶ **Corollary 11.** *For a sufficiently large constant c, the query time excluding the time spent on the cells of* Π_{j_l} *is bounded by* $O(n^{1-1/d}/b)$.

Proof. By Lemma 10, it suffices to show that $b \cdot r^{1-1/d} \cdot (n/r)^{1-1/(d-1)+\epsilon} = O(n^{1-1/d}/b)$, which is equivalent to $b^2 \cdot (n/r)^{1-1/(d-1)+\epsilon} = O((n/r)^{1-1/d})$. This in turn is equivalent to $b^2 = O((n/r)^{1/(d-1)-1/d-\epsilon})$, which holds for a sufficiently large c since $n/r = b^c$. ◀

Preprocessing cells of Π_{j_l}. Recall that Π_{j_l} has $O(b' \cdot b^{t_l - 1})$ cells each containing $2n/(b' \cdot b^{t_l-1})$ points of P, and $r/b < b' \cdot b^{t_l-1} \le r$. Let $r' = b' \cdot b^{t_l-1}$. By Corollary 11, we obtain the following recurrence on the query time $Q(n)$: $Q(n) = O(n^{1-1/d}/b) + O(r'^{1-1/d}) \cdot Q_1(n/r')$, where $Q_1(n/r')$ is the query time for each subproblem of size $O(n/r')$ since each cell of Π_{j_l} contains $O(n/r')$ points of P. As $b = \log^\rho n$ and $r/b < r' \le r$, we can write

$$Q(n) = O(n^{1-1/d}/\log^{\Omega(1)} n) + O((n/n_1)^{1-1/d}) \cdot Q_1(n_1), \tag{9}$$

with $n_1 = \log^{\Theta(1)} n$.

For notational convenience, let T refer to the sequence of partitions $\Pi_0, \Pi_{j_0}, \Pi_{j_1}, \ldots, \Pi_{j_l}$, and cells of Π_{j_l} form the leaves of T.

To solve $Q_1(n_1)$ in (9), we preprocess $P(\Delta)$ for each leaf cell $\Delta \in T$ recursively as above by using different parameters. Specifically, let $\tau > 0$ be an arbitrarily small constant to be set later. Setting $r_1 = n_1/\log^\tau n$, we apply Theorem 4 to construct a partition tree $T(\Delta)$ with $b_1 = \log^\rho n_1$ in $O(n_1^{d^2+2})$ time. As above (using the new parameters r_1 and b_1), we only use a subset of partitions of $T(\Delta)$ in our query data structure. For notational convenience, we use $T(\Delta)$ to refer to those partitions. The total time for constructing $T(\Delta)$ for all leaves $\Delta \in T$ is bounded by $O(n^{d^2+2})$ and the total space is still $O(n\log\log n)$.

With the same analysis as Lemma 10, we can obtain that the query time on $T(\Delta)$ excluding the time spent on the leaf cells of $T(\Delta)$ is $O(b_1 \cdot r_1^{1-1/d} \cdot (n_1/r_1)^{1-1/(d-1)+\epsilon})$. Define r_1' with respect to r_1 in the same way as above for r' with respect to r, i.e., $T(\Delta)$ has $O(r_1')$ leaf cells and each cell contains $O(n_1/r_1')$ points of P. As above, $r_1/b_1 < r_1' \le r_1$ holds. Consequently, we obtain $Q_1(n) = O(b_1 \cdot r_1^{1-1/d} \cdot (n_1/r_1)^{1-1/(d-1)+\epsilon}) + O(r_1'^{1-1/d}) \cdot Q_2(n_1/r_1')$, where $Q_2(n_1/r_1')$ is the query time for each leaf cell of $T(\Delta)$. Let $t = n_1/r_1'$. We have

$$Q_1(n) = O(b_1 \cdot r_1^{1-1/d} \cdot (n_1/r_1)^{1-1/(d-1)+\epsilon}) + O((n_1/t)^{1-1/d}) \cdot Q_2(t). \tag{10}$$

Combining (9) and (10) leads to:

$$Q(n) = O\left(\frac{n^{1-1/d}}{\log^{\Omega(1)} n}\right) + O\left(\left(\frac{n}{t}\right)^{1-1/d}\right) \cdot Q_2(t). \tag{11}$$

Since $t = n_1/r'_1$ and $r_1/b_1 < r'_1 \leq r_1$, we have $n/t = n/n_1 \cdot r'_1 \leq n/n_1 \cdot r_1 = n/\log^\tau n$. Therefore, we further obtain from the above

$$Q(n) = O\left(\frac{n^{1-1/d}}{\log^{\Omega(1)} n}\right) + O\left(\left(\frac{n}{\log^\tau n}\right)^{1-1/d}\right) \cdot Q_2(t)$$
$$= O\left(\frac{n^{1-1/d}}{\log^{\Omega(1)} n}\right) + O\left(\frac{n^{1-1/d}}{\log^{\Omega(1)} n}\right) \cdot Q_2(t). \qquad (12)$$

Since $t = n_1/r'_1$ and $r_1/b_1 < r'_1 \leq r_1$, we have $t \leq n_1/r_1 \cdot b_1 = O(\log^\tau n \cdot \log^\rho \log n)$. Therefore, we can write $t = O(\log^{\tau'} n)$ for another arbitrarily small constant τ'.

In summary, the above first builds a partition tree T and then builds partition trees $T(\Delta)$ for all leaf cells Δ of T (using different parameters). For notational convenience, we still use T to refer to the entire tree (i.e., attaching all leaf trees $T(\Delta)$ to T), which has $O(n/t)$ leaves, each containing $O(t)$ points. The space is bounded by $O(n \log\log n)$ and the total preprocessing time is $O(n^{d^2+2})$. Since t is tiny, we show in the full paper that after additional $O(n \log n)$ time and $O(n)$ space preprocessing, each query on any leaf cell of T can be answered in $O(\log t)$ time (i.e., $Q_2(t) = O(\log\log n)$). Consequently, the query time is bounded by $O(n^{1-1/d}/\log^{\Omega(1)} n)$. We can reduce the preprocessing time to $O(n^{1+\epsilon})$ using idea similar to Section 4.2 (i.e., build an "upper partition tree" of O(1) depth on top using simplicial partitions [17]). We conclude with the following theorem.

▶ **Theorem 12.** *Given a set of n simplices in $\mathbb{R}^d$, one can build a data structure of $O(n \log\log n)$ space so that the number of simplices containing a query point can be computed in $O(n^{1-1/d}/\log^{\Omega(1)} n)$ time. The preprocessing time is $O(n^{1+\epsilon})$ for any $\epsilon > 0$.*

5.1 Other related problems

As studied in [9], some related problems (as stated in the following two theorems) can be solved by similar techniques. For each problem, we basically follow the same high-level algorithmic framework as [9] but use our deterministic partition tree instead of the randomized one in [7]; this is very similar to the above simplex stabbing counting problem, so we omit these details. For each problem, however, we still need to come up with a method to answer queries in $O(\log t)$ time for subproblems of small sizes $t = \log^\tau n$ (for an arbitrarily small constant $\tau > 0$) and these are discussed in the full paper.

▶ **Theorem 13.** *Given a set of simplices in $\mathbb{R}^d$, one can build a data structure of $O(n \log\log n)$ space so that the simplices containing a query point can be reported in $O(n^{1-1/d}/\log^{\Omega(1)} n + k)$ time, where k is the output size. The preprocessing time is $O(n^{1+\epsilon})$ for any $\epsilon > 0$.*

▶ **Theorem 14.** *Given a set of n segments in the plane, one can build a data structure of $O(n \log\log n)$ space so that the number of segments intersecting a query segment can be computed in $O(\sqrt{n}/\log^{\Omega(1)} n)$ time (these segments can be reported in additional $O(k)$ time, where k is the output size). The preprocessing time is $O(n^{1+\epsilon})$ for any $\epsilon > 0$.*

6 Segment intersection detection

Let S be a set of n line segments in $\mathbb{R}^2$. We wish to build a data structure to decide whether a query line intersects any segment of S. The problem can be solved by Theorem 14. This section presents a new method that only needs $O(n)$ space while the query time is the same.

Let P be the set of $2n$ endpoints of the segments of S. With $r = 4n/b^3$ and $b = \log^\rho n$, we apply Theorem 4 to P to obtain a partition tree T consisting of collections Π_i, $0 \le i \le k+1$. By Theorem 4, the total number of cells is $O(r)$, each cell in Π_{k+1} contains at most $4n/r = \log^{3\rho} n$ points of P, and the runtime to construct T is $O(n^6)$. For each node $v \in T$, let $\Delta(v)$ denote the corresponding cell of v, which is a triangle in $\mathbb{R}^2$.

The following is a result from the previous work [22] for a special case of the problem. We will use the result as a subroutine in our approach.

▶ **Lemma 15.** ([22]) *If all segments of S intersect a given line segment, then one can build a data structure of $O(n)$ space in $O(n \log n)$ time so that whether a query line intersects any segment of S can be determined in $O(\log n)$ time.*

We store the segments of S in the partition tree T as follows (the idea is similar to [22]). For each segment $s \in S$, starting from the root, for each node v whose cell $\Delta(v)$ contains s (which is true initially when v is the root), if v is a leaf, then we store s at v (let S_v denote the set of all such segments stored at v). Otherwise, we check every child of v. If v has a child u whose cell $\Delta(u)$ contains s, then we proceed on u. Otherwise, for each child u, if $\Delta(u)$ contains an endpoint of s, then since $\Delta(u)$ does not contain s, s must intersect an edge e of $\Delta(u)$; we store s at e (let S_e denote the set of segments stored at e); note that since s has two endpoints, there are two such edges e but it suffices to store s in one such edge. This finishes the algorithm for storing s, which takes $O(b \log n)$ time. Because s is stored at either a leaf or a cell edge, the total space for storing all segments is $O(n)$. The time is $O(nb \log n)$.

Next, for each edge e of each cell of T, since all segments of S_e intersect e, we preprocess S_e using Lemma 15. Doing this for all cell edges e of T takes $O(n \log n)$ time and $O(n)$ space. For those segments stored in S_v for all leaves v, we will preprocess them into a data structure $\mathcal{D}_v$. Before discussing $\mathcal{D}_v$, we describe the query algorithm and analyze the time complexity.

Given a query line ℓ, starting from the root of T, for each node v, assume that ℓ intersects the boundary of $\Delta(v)$, which is true initially when v is the root. If v is a leaf, then we call the data structure $\mathcal{D}_v$ to check whether ℓ intersects a segment of S_v. Otherwise, for each child u of v, for each edge e of $\Delta(u)$, we apply the query algorithm of Lemma 15 to check whether ℓ intersects a segment of S_e; further, if ℓ crosses $\Delta(u)$, then we proceed on u. The following lemma justifies the correctness of the query algorithm.

▶ **Lemma 16.** *The query algorithm works correctly.*

Proof. If the query algorithm detects an intersection, then it is obviously true that ℓ intersects a segment of S. On the other hand, suppose ℓ intersects a segment s, say, at a point p. We argue that the query algorithm must detect an intersection. Indeed, according to our query algorithm, all nodes u of T whose cells $\Delta(u)$ are crossed by ℓ will be processed. If s is stored at a leaf v, then s is contained in $\Delta(v)$. Since ℓ intersects s, ℓ must cross $\Delta(v)$, and thus v must be processed and the data structure $\mathcal{D}_v$ will detect an intersection between ℓ and S_v.

If s is not stored at a leaf, then there must exist an internal node v such that $s \in \Delta(v)$ and s is not in $\Delta(u)$ for any child u of v. According to our preprocessing algorithm, s must be stored in $S_{e'}$ for an edge e' of some cell $\Delta(u)$ of a child u of v. Since $p \in s \subseteq \Delta(v)$ and $p \in \ell$, ℓ must cross $\Delta(v)$. Therefore, our query algorithm will process v by applying the query algorithm of Lemma 15 on S_e for every edge e of every child cell of v. When it is applied to $S_{e'}$, the intersection will be detected. ◀

We now analyze the query time. Recall that T has $O(r)$ leaves. By Theorem 4, the total number internal nodes of T whose cell boundaries are crossed by ℓ is $O(\sqrt{r/b})$, and for each such node, we need to call the query algorithm of Lemma 15 $O(b)$ times and each call takes $O(\log n)$ time. As such, the query time other than the time spent on calling $\mathcal{D}_v$ for those leaves v whose cell boundaries are crossed by ℓ (let V be the set of all such cells) is $O(\sqrt{r/b} \cdot b \cdot \log n) = O(\sqrt{n}/b \cdot \log n) = O(\sqrt{n}/\log^{\rho-1} n)$ since $r = 4n/b^3$ and $b = \log^{\rho} n$. By Theorem 4, $|V| = O(\sqrt{r}) = O(\sqrt{n/\log^{3\rho} n})$ and $|S_v| \leq 4n/r = \log^{3\rho} n$ for each leaf v of T.

Let $Q(n)$ be the query time. Following the above analysis, we obtain the following

$$Q(n) = O(\sqrt{rb} \cdot \log n) + O(\sqrt{r}) \cdot Q_1(n/r), \tag{13}$$

where $Q_1(\cdot)$ is the query time for each leaf $v \in V$, whose cell contains $O(n/r)$ points of P.

To solve $Q_1(n/r)$, we recursively process S_v for each leaf cell v of T as above by using different parameters. Specifically, let n_1 be the number of endpoints of S_v; hence $n_1 = O(n/r) = O(\log^{3\rho} n)$. Let $\tau > 0$ be an arbitrarily small constant to be set later. Setting $r_1 = n_1/\log^{\tau} n$, we apply Theorem 4 to construct a partition tree $T(v)$ with $b_1 = \log^{\rho} n_1$ in $O(n_1^6)$ time. The total time for constructing $T(\Delta)$ for all leaf cells Δ of T is thus bounded by $O(n^6)$. Using $T(v)$ to handle queries on S_v and following the above analysis, we obtain

$$Q(n_1) = O(\sqrt{r_1 b_1} \cdot \log n_1) + O(\sqrt{r_1}) \cdot Q_2(n_1/r_1), \tag{14}$$

where $Q_2(\cdot)$ is the query time for each leaf cell of $T(v)$, which contains $O(n_1/r_1)$ points of P.

Combining (13) and (14) leads to:

$$Q(n) = O\left(\frac{\sqrt{n}}{\log^{\Omega(1)} n}\right) + O\left(\sqrt{\frac{n}{t}}\right) \cdot Q_2(t), \text{ where } t = \log^{\tau} n. \tag{15}$$

In summary, the above first builds a partition tree T and then builds partition trees $T(v)$ for all leaves v of T (using different parameters). For notational convenience, we use T to refer to the entire tree (by attaching all trees $T(v)$ to T), which has $O(n/t)$ leaves, each containing $O(t)$ points. The space is bounded by $O(n)$. As t is small, with $O(n \log n)$ additional time and $O(n)$ space preprocessing, each subproblem $Q(t)$ in (15) can be solved in $O(\log t)$ time. This makes the total query time $Q(n)$ bounded by $O(\sqrt{n}/\log^{\Omega(1)} n)$. We can also reduce the preprocesing time to $O(n^{1+\epsilon})$. See the full paper for the details.

▶ **Theorem 17.** *Given a set of n segments in the plane, there is a data structure of $O(n)$ space that can determine whether a query line intersects any segment in $O(\sqrt{n}/\log^{\Omega(1)} n)$ time. The data structure can be built in $O(n^{1+\epsilon})$ time for any $\epsilon > 0$.*

7 Ray-shooting among non-intersecting segments

Let S be a set of n line segments in the plane such that no two segments intersect. The problem is to build a data structure to compute the first segment hit by a query ray.

The following is a result from the previous work [22] for a special case of the problem. We will use it as a subroutine in our approach.

▶ **Lemma 18.** ([22]) *If all segments of S intersect a given line segment, then one can build a data structure of $O(n)$ space in $O(n \log n)$ time so that a ray-shooting query can be answered in $O(\log n)$ time.*

We build the partition tree T and store S in T in a way similar to the segment intersection detection problem in Section 6 except that we use Lemma 18 to preprocess S_e for each cell edge e of T. In addition, for each leaf v of T, we will build a data structure $\mathcal{D}_v$ on S_v.

Given a query ray ρ, starting from the root of T, for each node v, assume that ρ intersects the boundary of $\Delta(v)$, which is true initially when v is the root. If v is a leaf, then we use the data structure $\mathcal{D}_v$ to find the first segment of S_v hit by ρ as our candidate solution segment. Otherwise, for each child u of v, for each edge e of $\Delta(u)$, apply the query algorithm of Lemma 18 to find the first ray of S_e hit by ρ as a candidate; further, if ρ crosses $\Delta(u)$, then we proceed on u. Finally, among all candidate segments, we return the one whose intersection with ρ is closest to the origin of ρ. The correctness follows a similar argument as Lemma 16.

As in Section 6, for each leaf v of T, we construct the data structure $\mathcal{D}_v$ by preprocessing S_v recursively once. The query time analysis follows exactly the same method as in Section 6 since the query time of Lemma 18 is the same as that of Lemma 15, and thus we can also obtain the recurrence (15) with the same value of t. As t is small, with additional $O(n \log n)$ time and $O(n)$ space preprocessing, each subproblem $Q(t)$ in (15) can be solved in $O(\log t)$ time. This makes the total query time $Q(n)$ bounded by $O(\sqrt{n}/\log^{\Omega(1)} n)$. We thus have the following result.

▶ **Lemma 19.** *Given a set of n segments in the plane, there is a data structure of $O(n)$ space that can compute the first segment hit by a query ray in $O(\sqrt{n}/\log^{\Omega(1)} n)$ time. The data structure can be built in $O(n^6)$ time.*

As before, we can reduce the preprocesing time to $O(n^{1+\epsilon})$. See the full paper for the details.

▶ **Theorem 20.** *Given a set of n segments in the plane, there is a data structure of $O(n)$ space that can compute the first segment hit by a query ray in $O(\sqrt{n}/\log^{\Omega(1)} n)$ time. The data structure can be built in $O(n^{1+\epsilon})$ time for any $\epsilon > 0$.*

References

1 Pankaj K. Agarwal. Range searching, in *Handbook of Discrete and Computational Geometry,* C.D. Tóth, J. O'Rourke, and J.E. Goodman (eds.), pages 1057–1092. CRC Press, 3rd edition, 2017.

2 Pankaj K. Agarwal. Simplex range searching and its variants: a review. In *A Journey Through Discrete Mathematics*, pages 1–30. Springer, 2017. doi:10.1007/978-3-319-44479-6_1.

3 Pankaj K. Agarwal and Jiří Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993. doi:10.1137/0222051.

4 Pankaj K. Agarwal and Micha Sharir. Applications of a new space-partitioning technique. *Discrete and Computational Geometry*, 9:11–38, 1993. doi:10.1007/BF02189304.

5 Pankaj K. Agarwal and Micha Sharir. Pseudoline arrangements: Duality, algorithms, and applications. *SIAM Journal on Computing*, 34:526–552, 2005. doi:10.1137/S0097539703433900.

6 Reuven Bar-Yehuda and Sergio Fogel. Variations on ray shootings. *Algorithmica*, 11:133–145, 1994. doi:10.1007/BF01182772.

7 Timothy M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47:661–690, 2012. doi:10.1145/1810959.1810961.

8 Timothy M. Chan and Da Wei Zheng. Hopcroft's problem, log-star shaving, 2D fractional cascading, and decision trees. *ACM Transactions on Algorithms*, 2023. doi:10.1145/3591357.

9 Timothy M. Chan and Da Wei Zheng. Simplex range searching revisited: How to shave logs in multi-level data structures. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1493–1511, 2023. doi:10.1137/1.9781611977554.ch54.

10 Bernard Chazelle. Lower bounds on the complexity of polytope range searching. *Journal of the American Mathematical Society*, 2(4):637–666, 1989. doi:10.2307/1990891.

11 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9(2):145–158, 1993. doi:10.1007/BF02189314.

12 Siu Wing Cheng and Ravi Janardan. Algorithms for ray-shooting and intersection searching. *Journal of Algorithms*, 13:670–692, 1992. doi:10.1016/0196-6774(92)90062-H.

13 Herbert Edelsbrunner, J. O'Rourke, and Raimund Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15:341–363, 1986. doi:10.1137/0215024.

14 Herbert Edelsbrunner and Emo Welzl. Halfplanar range search in linear space and $O(n^{0.695})$ query time. *Information Processing Letters*, 23:289–293, 1986. doi:10.1016/0020-0190(86)90088-8.

15 Leonidas J. Guibas, Mark H. Overmars, and Micha Sharir. Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 64–73, 1988. doi:10.1007/3-540-19487-8_7.

16 David Haussler and Emo Welzl. ϵ-nets and simplex range queries. *Discrete and Computational Geometry*, 2:127–151, 1987. doi:10.1007/BF02187876.

17 Jiří Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8(3):315–334, 1992. doi:10.1007/BF02293051.

18 Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10(1):157–182, 1993. doi:10.1007/BF02573972.

19 Jiří Matoušek. Geometric range searching. *ACM Computing Survey*, 26:421–461, 1994. doi:10.1145/197405.197408.

20 Mark H. Overmars, Haijo Schipper, and Micha Sharir. Storing line segments in partition trees. *BIT Numerical Mathematics*, 30:385–403, 1990. doi:10.1007/BF01931656.

21 Haitao Wang. Unit-disk range searching and applications. *Journal of Computational Geometry*, 14:343–394, 2023. doi:10.20382/jocg.v14i1a13.

22 Haitao Wang. Algorithms for subpath convex hull queries and ray-shooting among segments. *SIAM Journal on Computing*, 53:1132–1161, 2024. doi:10.1137/21M145118X.

23 Dan E. Willard. Polygon retrieval. *SIAM Journal on Computing*, 11:149–165, 1982. doi:10.1137/0211012.

24 F. Frances Yao. A 3-space partition and its applications. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 258–263, 1983. doi:10.1145/800061.808755.

25 F. Frances Yao, David P. Dobkin, Herbert Edelsbrunner, and Mike Paterson. Partitioning space for range queries. *SIAM Journal on Computing*, 18:371–384, 1989. doi:10.1137/0218025.