Intersection Searching Amid Tetrahedra in 4-Space and Efficient Continuous Collision Detection

Esther Ezra
School of Computer Science, Bar Ilan University, Ramat Gan, Israel

Micha Sharir
School of Computer Science, Tel Aviv University, Tel Aviv, Israel

Abstract

We develop data structures for intersection detection queries in four dimensions that involve segments, triangles and tetrahedra. Specifically, we study two main problems: (i) Preprocess a set of n tetrahedra in $\mathbb{R}^4$ into a data structure for answering segment-intersection queries amid the given tetrahedra (referred to as *segment-tetrahedron intersection queries*), and (ii) Preprocess a set of n triangles in $\mathbb{R}^4$ into a data structure that supports triangle-intersection queries amid the input triangles (referred to as *triangle-triangle intersection queries*). As far as we can tell, these problems have not been previously studied.

For problem (i), we first present a "standard" solution which, for any prespecified value $n \leq s \leq n^6$ of a so-called storage parameter s, yields a data structure with $O^*(s)$ storage and expected preprocessing, which answers an intersection query in $O^*(n/s^{1/6})$ time (here and in what follows, the $O^*(\cdot)$ notation hides subpolynomial factors). For problem (ii), using similar arguments, we present a solution that has the same asymptotic performance bounds.

We then improve the solution for problem (i), and present a more intricate data structure that uses $O^*(n^2)$ storage and expected preprocessing, and answers a segment-tetrahedron intersection query in $O^*(n^{1/2})$ time. Using the parametric search technique of Agarwal and Matoušek [3], we can obtain data structures with similar performance bounds for the *ray-shooting* problem amid tetrahedra in $\mathbb{R}^4$. Unfortunately, so far we do not know how to obtain a similar improvement for problem (ii).

Our algorithms are based on a primal-dual technique for range searching with semi-algebraic sets, based on recent advances in this area [2, 11]. As this is a result of independent interest, we spell out the details of this technique.

As an application, we present a solution to the problem of "continuous collision detection" amid moving tetrahedra in 3-space. That is, the workspace consists of n tetrahedra, each moving at its own fixed velocity, and the goal is to detect a collision between some pair of moving tetrahedra. Using our solutions to problems (i) and (ii), we obtain an algorithm that detects a collision in $O^*(n^{12/7})$ expected time. We also present further applications, including an output-sensitive algorithm for constructing the arrangement of n tetrahedra in $\mathbb{R}^4$ and an output-sensitive algorithm for constructing the intersection or union of two or several nonconvex polyhedra in $\mathbb{R}^4$.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Computational geometry, Ray shooting, Tetrahedra in $\mathbb{R}^4$, Intersection queries in $\mathbb{R}^4$, Polynomial partitioning, Range searching, Semi-algebraic sets, Tradeoff

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.51

Related Version *Full Version*: `http://arxiv.org/abs/2208.06703`

Funding *Esther Ezra*: Work partially supported by NSF CAREER under Grant CCF:AF-1553354 and by Grant 824/17 from the Israel Science Foundation.
Micha Sharir: Work partially supported by Grant 260/18 from the Israel Science Foundation.

1 Introduction

In this paper we consider various intersection problems involving segments, triangles and tetrahedra in $\mathbb{R}^4$. In four dimensions, the interesting setups involve (i) intersections between (one-dimensional) segments and (three-dimensional) tetrahedra, and (ii) between (two-dimensional) triangles and (two-dimensional) triangles. We study both problems, and derive efficient solutions to each of them.

As an interesting application, we consider the *continuous collision detection* problem, where the input consists of n tetrahedra in $\mathbb{R}^3$, each of which is moving at some constant velocity of its own, and the goal is to detect whether any pair of them collide. Adding the time as a fourth coordinate, this becomes a batched version of intersection detection in $\mathbb{R}^4$, involving both setups (i) and (ii). Other applications include output-sensitive construction of the arrangement of n tetrahedra in $\mathbb{R}^4$, and an output-sensitive algorithm for computing the intersection or the union of two or several not necessarily convex polyhedra in $\mathbb{R}^4$. In the three-dimensional versions of these problems, which were recently studied in [7], the only setup that needed to be considered was segment intersection amid triangles. In four dimensions, though, we also face the triangle-triangle intersection problem, since we also need to find intersections between pairs of 2-faces of the input objects.

Setup (i): Segment-tetrahedron intersection queries. Consider first the case of segments vs. tetrahedra. In the setup considered here, the input objects are n (not necessarily pairwise openly disjoint) tetrahedra in $\mathbb{R}^4$ and the query objects are segments, and the goal is to detect, count, or report intersections between the query segment and the input tetrahedra.

As far as we can tell, this problem has not been explicitly studied so far. We first present, in Section 2, a "traditional" (albeit novel) solution, in which the problem is reduced to a range searching problem in a suitable parametric space, which, in the case of (lines supporting) segments in $\mathbb{R}^4$, is six-dimensional. We carefully adapt and combine recent techniques, developed by Agarwal et al. [2] and Matoušek and Patáková [11], which provide algorithmic constructions of intricate space decompositions based on partitioning polynomials. Using this machinery, we solve the problem so that, with a so-called storage parameter s, a segment intersection query can be answered in[1] $O^*(n/s^{1/6})$ time, for any $n \le s \le n^6$, and the storage and preprocessing cost are both $O^*(s)$.

A special case of this setup is an extension to four dimensions of the classical *ray shooting* problem, which has mostly been studied in two and three dimensions. In a general setting, we are given a collection S of n simply-shaped objects, and the goal is to preprocess S into a data structure that supports efficient ray shooting queries, where each query specifies a ray ρ and asks for the first object of S hit by ρ, if such an object exists. In this work we only consider the (already challenging) case of input tetrahedra. Using the parametric search technique of Agarwal and Matoušek [3], ray shooting queries can be reduced to segment-intersection detection queries, up to a polylogarithmic factor in the query cost. By the above discussion, we obtain the following result:

▶ **Theorem 1.** *Given a collection $\mathcal{T}$ of n tetrahedra in $\mathbb{R}^4$, and any storage parameter $n \le s \le n^6$, we can preprocess $\mathcal{T}$ into a data structure of size $O^*(s)$, in randomized $O^*(s)$ expected time, so that we can answer any segment-intersection or ray-shooting query in $\mathcal{T}$ in $O^*(n/s^{1/6})$ time. The query time bound applies to segment-intersection detection and counting queries (and to ray shooting queries). The cost is $O^*(n/s^{1/6}) + O(k)$ for reporting queries.*

[1] As in the abstract, the $O^*(\cdot)$ notation hides subpolynomial factors, typically of the form n^ε, for any $\varepsilon > 0$, and their coefficients which depend on ε.

We later improve upon this standard algorithm in Section 4, where we show:

▶ **Theorem 2.** *A collection S of n arbitrary tetrahedra in $\mathbb{R}^4$ can be preprocessed into a data structure of size $O^*(n^2)$, in expected time $O^*(n^2)$, which supports segment-intersection detection and counting queries and ray-shooting queries in time $O^*(n^{1/2})$ per query.*

This indeed improves the bounds stated in Theorem 1, which, with $s = O^*(n^2)$ storage, has query time $O^*(n^{2/3})$. Furthermore, with the storage bound specified in Theorem 2, the query bound is similar to that obtained for ray-shooting amid hyperplanes (rather than tetrahedra) in $\mathbb{R}^4$ [3].

We then go on, in the full version of the paper[2] to extend the result to obtain a tradeoff between storage (and expected preprocessing time) and query time. We show that, with storage parameter s, which can vary between n and n^6, we can answer a segment intersection or a ray shooting query in time

$$Q(n,s) = \begin{cases} O^*\left(\frac{n^{7/6}}{s^{1/3}}\right) & \text{for } s = O(n^2) \\ O^*\left(\frac{n^{3/4}}{s^{1/8}}\right) & \text{for } s = \Omega(n^2). \end{cases} \tag{1}$$

See Figure 1 for an illustration. This yields algorithms that answer m segment intersection or ray-shooting queries on n tetrahedra in $\max\left\{O^*(m^{3/4}n^{7/8}+n),\ O^*(m^{8/9}n^{2/3}+m)\right\}$ time and storage. The first (resp., second) bound dominates when $m \le n^{3/2}$ (resp., $m \ge n^{3/2}$).

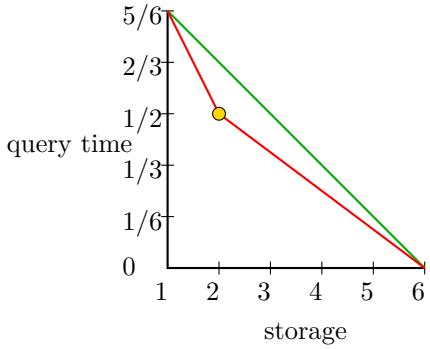

Figure 1 The tradeoff between storage and query time. The breakpoint in the graph represents the case studied in Section 4. Both axes are drawn on a logarithmic scale.

Setup (ii): Triangle-triangle intersection detection. We next consider the other setup of intersection queries, where both input and query objects are triangles in $\mathbb{R}^4$. We show that this setup can also be reduced, similar to setup (i), to a multi-level range searching problem in $\mathbb{R}^6$ involving semi-algebraic ranges. This allows us to obtain the same performance bounds here too. Namely we have:

▶ **Theorem 3.** *Given a collection $\mathbf{\Delta}$ of n triangles in $\mathbb{R}^4$, and any storage parameter $n \le s \le n^6$, we can preprocess $\mathbf{\Delta}$ into a data structure of size $O^*(s)$, in randomized $O^*(s)$ expected time, so that we can answer any triangle-intersection query in $\mathbf{\Delta}$ in $O^*(n/s^{1/6})$ time.*

[2] Soon to be available on arXiv.

Since both input and query objects are triangles, it is also interesting to consider the bichromatic batched version of the problem. Namely we have:

▶ **Theorem 4.** *Given two collections R and B of triangles in $\mathbb{R}^4$, of respective sizes m and n, We can detect an intersection between some triangle of R and some triangle of B, or count all such intersections, in time $O^*(m^{6/7}n^{6/7} + m + n)$. We can also report all these intersections in time $O^*(m^{6/7}n^{6/7} + m + n + k)$, where k is the output size.*

As a consequence, integrating this bound with the one obtained in Theorem 1 (in which, similar to the preceding argument, we need to set $s = n^{12/7}$ to match the performance with that stated above, as is easily verified), we obtain an overall $O^*\left(n^{12/7}\right)$ expected time solution for the continuous collision detection problem, that is:[3]

▶ **Theorem 5.** *Given n tetrahedra in $\mathbb{R}^3$, each of which is moving at some constant velocity of its own, one can detect a collision between any pair of moving tetrahedra in $O^*\left(n^{12/7}\right)$ expected time.*

Collision detection has been widely studied – see Lin, Manocha and Kim [10] for a recent comprehensive survey, and the references therein. We are not aware of any work that addresses the exact algorithmic approach for the specific setup considered here, although there are some works, such as Canny [6] or Schömer and Thiel [12], that address similar contexts.

We then consider the applications of our techniques to the problems of output-sensitive construction of an arrangement of tetrahedra in $\mathbb{R}^4$, and of constructing the intersection or union of two or several (nonconvex) polyhedra in $\mathbb{R}^4$. Using the bounds for setups (i) and (ii), we obtain, in Section 5:

▶ **Theorem 6.** *(i) Let $\mathcal{T}$ be a collection of n tetrahedra in general position in $\mathbb{R}^4$. We can construct the arrangement $\mathcal{A}(\mathcal{T})$ of $\mathcal{T}$ in $O^*(n^{12/7} + n^{1/2}k_2 + k_4)$ expected time, where k_2 is the number of intersecting pairs of tetrahedra in $\mathcal{T}$, and k_4 is the number of vertices of $\mathcal{A}(\mathcal{T})$. (ii) Given two arbitrary polyhedra R and B in $\mathbb{R}^4$, each of complexity $O(n)$, that lie in general position with respect to one another, the intersection $R \cap B$ can be computed in expected time $O^*(n^{12/7} + n^{1/2}k_2 + k_4)$, where k_2 is the number of 2-faces of $\mathcal{A}(R \cup B)$, and k_4 is the number of vertices of $\mathcal{A}(R \cup B)$.*

As another application of our technique we present, in the full version, an efficient algorithm for detecting or reporting intersections between n 2-flats and n lines in $\mathbb{R}^4$. We show that, given n lines and n 2-flats in $\mathbb{R}^4$, one can detect whether some line intersects some 2-flat in $O^*(n^{13/8})$ expected time. One can also report all k intersections in $O^*(n^{13/8} + k)$ expected time. This result is a degenerate special case of the triangle-triangle intersection setup, and admits a faster solution. (Note that in general position 2-flats and lines are not expected to meet in $\mathbb{R}^4$, which makes this special case interesting.)

Setup (iii): Tetrahedron-segment intersection queries. We can also handle a symmetric setup, in which the input consists of n segments in $\mathbb{R}^4$ and the query is with a tetrahedron T, where the goal is to detect, count or report intersections between T and the input segments. Using a similar machinery, we obtain the same asymptotic performance bounds, as in the standard solution, for this setup too.

[3] Here we use an obvious divide-and-conquer approach in order to reduce the general (non-bichromatic) problem to the bichromatic version.

▶ **Theorem 7.** *Given a collection S of n segments in $\mathbb{R}^4$, and any storage parameter $n \leq s \leq n^6$, we can preprocess S into a data structure of size $O^*(s)$, in randomized $O^*(s)$ expected time, so that we can answer any tetrahedron-intersection query in S in $O^*(n/s^{1/6})$ time. The query time bound applies to tetrahedron-intersection detection and counting queries. The cost is $O^*(n/s^{1/6}) + O(k)$ for reporting queries.*

2 Segment-Intersection amid Tetrahedra: An Initial Algorithm

In this section we present an initial solution to the problem of segment-intersection detection amid tetrahedra in four dimensions, which is based on a careful combination of the recent range searching machinery of [2, 11]. We do so because (a) as far as we can tell, such a solution has not yet been spelled out in the literature, (b) the adaptation of the available techniques to this problem is not simple, requires nontrivial and careful enhancements, and is of independent interest, and (c) this gives a yardstick for appreciating the improvement obtained by our improved algorithm, presented in Section 4.

The parametric search technique of Agarwal and Matoušek [3] reduces ray shooting queries to segment-intersection detection queries, so it suffices to consider the latter problem. The reporting and counting variants are simple extensions of the same technique, as will be discussed as we go.

To obtain a tradeoff between the storage of the structure and the query time, our algorithm uses a primal-dual approach. However, both the primal and dual setups suffer from the fact that, in four dimensions, segments and tetrahedra require too many parameters to specify. Specifically, a segment requires eight parameters (e.g., by specifying its two endpoints), while a tetrahedron requires 16 parameters (e.g., by specifying the coordinates of its four vertices).

To address this issue, we use a multi-level data structure, where each level caters to one aspect of the condition that a segment crosses a tetrahedron. This is done so that, at each of these levels, the number of parameters that a segment or a tetrahedron requires is at most six. Specifically, the condition that a segment e, that lies on a line ℓ, intersects a tetrahedron Δ, supported by a hyperplane h_Δ, is the conjunction of the following conditions:

(i) The two endpoints of e lie on different sides of h_Δ.

(ii) With a suitable choice of a direction of ℓ and an orientation of Δ, ℓ has a positive orientation with respect to each of the 2-planes that support the four 2-faces of Δ.

Conditions (i) and (ii) are the conjunction of a total of six sub-conditions: the first two conditions tests the position of some endpoint of e with respect to the hyperplanes h_Δ, and the other four conditions tests the orientation of ℓ with respect to the 2-planes supporting specific 2-faces of the tetrahedra. Thus, the dual structure has six levels, two for testing the sub-conditions of condition (i) and four for testing the sub-conditions of condition (ii).

More precisely, each but the last level collects all the tetrahedra Δ that satisfy the corresponding sub-condition for the query segment (that a specific endpoint of e lies in a specific side of h_Δ for the first two levels, and that the oriented 2-plane supporting a specific 2-face of Δ is positively oriented with respect to the directed line ℓ for the last four levels), as the disjoint union of precomputed canonical sets of tetrahedra. The last level just tests whether the last sub-condition is satisfied for any tetrahedron in the current canonical set.

We use the fact that lines in $\mathbb{R}^4$ require six real parameters to specify. The space of lines in $\mathbb{R}^4$ is actually projective, but for simplicity of presentation we regard it as a real space, and ignore the special cases in which the real representation fails. Handling these cases follows the same approach, and is in fact simpler. Alternatively, a generic (say random) rotation of the coordinate frame allows us to ignore them altogether.

One simple way to represent a line ℓ in $\mathbb{R}^4$ is by the points $u_\ell^0 = (x_0, y_0, z_0, 0)$ and $u_\ell^1 = (x_1, y_1, z_1, 1)$ at which ℓ crosses the hyperplanes $w = 0$ and $w = 1$, respectively (ignoring lines that are orthogonal to the w-axis), so the line ℓ can be represented as the point $p_\ell = (x_0, y_0, z_0, x_1, y_1, z_1)$ in $\mathbb{R}^6$, as desired.

Similarly, 2-planes in $\mathbb{R}^4$ also require six parameters to specify. This is simply because the duality in $\mathbb{R}^4$ maps lines to 2-planes and vice versa, but a concrete way to represent 2-planes by six parameters is to specify three points on a 2-plane π that are intersections of π with three fixed 2-planes, such as, say, $x = y = 0$, $x = 0$ and $y = 1$, and $x = y = 1$ (again ignoring special directions of π). Each of the intersection points has two degrees of freedom (as two of its coordinates are fixed), for a total of six. Denote these points as $v_\pi^{(00)}$, $v_\pi^{(01)}$, and $v_\pi^{(11)}$, and put $q_\pi = \left(v_\pi^{(00)}, v_\pi^{(01)}, v_\pi^{(11)}\right)$, listing only the w- and z-coordinates of each point, so q_π is a point in $\mathbb{R}^6$.

These observations are meaningful only for the last four levels of the structure. The first two levels are simpler, as they deal with points (the endpoints of e) and hyperplanes (those supporting the tetrahedra of $\mathcal{T}$) in $\mathbb{R}^4$. Thus each of the first two levels is a halfspace range searching structure for points and halfspaces in $\mathbb{R}^4$. (Actually, this is the case when we pass to the dual 4-space; in the primal we have a point-enclosure problem, where the query is a point and the input consists of halfspaces bounded by the relevant hyperplanes.) Using standard techniques (see, e.g., [1]), this can be done, for N halfspaces in the current canonical subset and using $O^*(N)$ storage, so that a query costs $O^*(N^{3/4})$ time.[4] This cost will be subsumed by the query time bounds for the last four levels. The cost of a query includes the cost of reporting its output, as a list of canonical sets.

We next consider the (more involved) situation in the last four levels of the structure. Here the query segment is replaced by its supporting line ℓ, and each tetrahedron Δ is replaced by the 2-plane supporting a specific 2-face of Δ. In the primal setup, the line ℓ is represented as a point p_ℓ in (projective) 6-space, in the manner just described, and a tetrahedron Δ, represented by a suitable 2-plane π, is represented as a semi-algebraic region K_π, consisting of all points that represent (directed) lines that are positively oriented with respect to π. The problem that we face is a point-enclosure query, in which we want to determine whether p_ℓ lies in any of the regions K_π (alternatively, count or report all these regions). In the dual setup, the 2-planes π are represented as points in $\mathbb{R}^6$, and the (directed) query line ℓ is represented as a semi-algebraic region Q_ℓ that consists of all (oriented) 2-planes that are positively oriented with respect to ℓ. The problem here is a semi-algebraic range searching query, where we want to determine whether Q_ℓ contains any input point (alternatively, count or report all these points).

The orientation test of ℓ with respect to π amounts to computing the sign of the 5×5 determinant

$$\begin{vmatrix} u_\ell^0 & 1 \\ u_\ell^1 & 1 \\ v_\pi^{(00)} & 1 \\ v_\pi^{(01)} & 1 \\ v_\pi^{(11)} & 1 \end{vmatrix}, \tag{2}$$

with a suitable orientation of the pair of points u_ℓ^0, u_ℓ^1 on ℓ (dictating the direction of ℓ), and of the triple of points $v_\pi^{(00)}$, $v_\pi^{(01)}$, $v_\pi^{(11)}$ on π (dictating the orientation of π).

[4] A tradeoff between storage and query time is also available, but we do not need it here.

To compute these signs, at each of the four latter levels of the structure, we use a primal-dual approach, where the top part of the structure is in the primal, and at each of its leaf nodes we pass to the dual.

The dual setup. The dual setup is simpler, so we begin with its description. In the dual setup, each tetrahedron Δ of the current canonical subset of $\mathcal{T}$ is mapped to the point $q_\pi = \left(v_\pi^{(00)}, v_\pi^{(01)}, v_\pi^{(11)}\right)$ in $\mathbb{R}^6$, where π is the 2-plane supporting the 2-face of Δ that corresponds to the present level. The query line ℓ is mapped to a semi-algebraic region Q_ℓ of constant complexity in $\mathbb{R}^6$, consisting of all points q_π that represent (oriented) 2-planes that have positive orientation with respect to ℓ, that is, the corresponding determinant in (2) is positive. (The resulting polynomial is cubic in q_π.)

As already mentioned, the task at hand, at each but the last level, is to collect the points q_π that lie in Q_ℓ, as the disjoint union of a small number of precomputed canonical sets of tetrahedra, and the task at the last level is to determine whether Q_ℓ contains any point q_π, for π corresponding to the last 2-faces of the tetrahedra in the present canonical subset of $\mathcal{T}$. In other words, we have, at each of these levels, a problem involving range searching with semi-algebraic ranges in $\mathbb{R}^6$. Using the algorithm of Matoušek and Patáková [11], which is a simplified version of the algorithm of Agarwal et al. [4], this can be done, for N tetrahedra with $O^*(N)$ storage, so that a query takes $O^*(N^{5/6})$ time (including the cost of reporting, without enumerating, the output canonical sets). See [1, Theorem 6.1] for more details.

The primal setup. With this procedure at hand, we go back to the primal structure, at each of the last four levels. As noted, the problem that we face there is a point enclosure problem, where the input consists of some N constant-complexity semi-algebraic regions in $\mathbb{R}^6$ of the form K_π, and the query is the point p_ℓ that represents ℓ, as defined earlier, and the task is to collect all the regions K_π that contain p_ℓ, as the disjoint union of a small number of precomputed canonical sets, or, at the last level, to determine whether p_ℓ is contained in any such region.

This problem has recently been studied in Agarwal et al. [2], using a multi-level polynomial partitioning technique, for the case where we allow maximum storage for the structure (that is, $O^*(N^6)$ in our case) and want the query time to be logarithmic. We next show that the structure can be modified so that its preprocessing stops "prematurely" when its overall storage attains some prescribed value, and each of the subproblems at the new leaves can be handled via the dual algorithm presented above.

The crucial technical tool in [2], on which their technique is based, is the following result. We give here a restricted specialized version that suffices for our purposes. (When applying this tool in d dimensions, the parameter 6 has to be replaced by d.)

▶ **Theorem 8** (A specialized version of Agarwal et al. [2, Corollary 4.8]). *Given a set Ψ of N constant-degree algebraic surfaces in $\mathbb{R}^6$, and a parameter $0 < \delta < 1/6$, there are finite collections $\Omega_0, \ldots, \Omega_6$ of semi-algebraic sets in $\mathbb{R}^6$ with the following properties.*

- *For each index i, each cell $\omega \in \Omega_i$ is a connected semi-algebraic set of constant complexity. The size $|\Omega_i|$ of Ω_i (the number of its sets) is a constant that depends on δ.*
- *For each index i and each $\omega \in \Omega_i$, at most $\frac{N}{4|\Omega_i|^{1/6-\delta}}$ surfaces from Ψ cross ω (intersect ω without fully containing it).*
- *The cells partition $\mathbb{R}^6$, in the sense that $\mathbb{R}^6 = \bigsqcup_{i=0}^{6} \bigsqcup_{\omega\in\Omega_i} \omega$, where $\bigsqcup$ denotes disjoint union.*

The sets in $\Omega_0, \ldots, \Omega_6$ can be computed in $O(n+m)$ expected time, where the constant of proportionality depends on δ, by a randomized algorithm. For each i and for every set $\omega \in \Omega_i$, the algorithm returns a semi-algebraic representation of ω, a reference point inside ω, and the subset of surfaces of Ψ that cross ω.

In our case, the surfaces of Ψ are the boundaries of the regions K_π. A straightforward enhancement of the algorithm of [2] also yields, for each i and each $\omega \in \Omega_i$, the set of regions K_π that fully contain ω, within the same asymptotic time bound.

We compute the partition of Theorem 8 and find, for each $\psi = \partial K_\pi \in \Psi$, the sets $\omega \in \Omega_i$, over all $i = 0, \ldots, 6$, that ψ crosses, and those that are fully contained in K_π. For each i and $\omega \in \Omega_i$, let $\mathcal{K}_{i,\omega}$ (resp., $\mathcal{K}^0_{i,\omega}$) denote the set of tetrahedra $\Delta \in \mathcal{T}$ for which ∂K_π crosses ω (resp., K_π fully contains ω).

The overall size of the sets $\mathcal{K}^0_{i,\omega}$, over all i and $\omega \in \Omega_i$, is $O(N)$, with a constant that depends on δ (that is, on the sizes $|\Omega_i|$, which depend on δ).

For each i and ω we also have a recursive subproblem that involves the subset $\mathcal{K}_{i,\omega}$ of the tetrahedra Δ for which ∂K_π crosses ω. Putting $r_i := |\Omega_i|$, for $i = 0, \ldots, 6$, we have, for each i and ω, $|\mathcal{K}_{i,\omega}| \leq \dfrac{N}{4r_i^{1/6-\delta}}$. We run the recursion, but not all the way through, as in [2]. Instead, we use the following storage allocation rule. We fix the storage that we are willing to allocate to the structure, and distribute it among the nodes of the recursion, as follows. To simplify the analysis, we distinguish between the storage itself, and the so-called *storage parameter* s, which is what we actually manage, but we have the property that the actual storage will always be $O^*(s)$.

Let s be the storage parameter that we allocate at the root of the structure. For each i and each set $\omega \in \Omega_i$, we allocate the storage parameter $s/(4|\Omega_i|)$ for ω. Hence, when we reach some set ω at a deeper level of recursion, say level j, the storage parameter allocated to ω is $\dfrac{s}{4^j|\Omega^{(1)}_{i_1}| \cdot |\Omega^{(2)}_{i_2}| \cdots |\Omega^{(j)}_{i_j}|}$, where $\Omega^{(1)}_{i_1}$, $\Omega^{(2)}_{i_2}, \ldots, \Omega^{(j)}_{i_j}$, for indices $0 \leq i_1, i_2, \ldots, i_j \leq 6$, are the partition families at the ancestors of ω in the recursion.

We stop the recursion when we reach nodes for which the allocated storage parameter is (roughly) equal to the number of tetrahedra at the node; a more precise statement of the termination rule is given shortly.

Put, for each set ω, $r_\omega := |\Omega^{(1)}_{i_1}| \cdot |\Omega^{(2)}_{i_2}| \cdots |\Omega^{(j)}_{i_j}|$, using the above notation for ω. The storage parameter allocated to ω is thus $s/(4^j r_\omega)$. Also, by Theorem 8, the number of tetrahedra Δ that participate in the subproblem at ω is at most

$$\frac{n}{4^j|\Omega^{(1)}_{i_1}|^{1/6-\delta} \cdot |\Omega^{(2)}_{i_2}|^{1/6-\delta} \cdots |\Omega^{(j)}_{i_j}|^{1/6-\delta}} = \frac{n}{4^j r_\omega^{1/6-\delta}},$$

and the stopping condition that we use is that $\dfrac{s}{4^j r_\omega} = \dfrac{n}{4^j r_\omega^{1/6-\delta}}$, or $r_\omega = (s/n)^{(6/5)/(1+6\delta/5)}$. The size of a subproblem at a leaf is (using the $O^*(\cdot)$ notation to hide exponents that are proportional to δ and constants of proportionality that depend on δ)

$$n_\omega = \frac{n}{4^j r_\omega^{1/6-\delta}} = \frac{1}{4^j} O^*\left(\frac{n^{6/5}}{s^{1/5}}\right) = O^*\left(\frac{n^{6/5}}{s^{1/5}}\right).$$

In more detail, since all the parameters r_j in Theorem 8 are at least some sufficiently large constant that we can control, we can make the factor 4^j to be $O(s^\delta)$, for any $\delta > 0$ of our choice. To be more precise, the choice of δ determines how large the parameters r_j have to be taken to ensure that $4^j = O(s^\delta)$, and the choice of these parameters adds a constant factor to the query cost (incurred by the cost of locating, in brute force, the cells ω, at the various recursive levels, that contain p_ℓ), which depends on these parameters, and thus on δ.

At each leaf ω we pass to the dual structure reviewed above. It uses $O^*(n_\omega)$ storage and answers a query in time $O^*(n_\omega^{5/6}) = O^*(n/s^{1/6})$. To answer a query with a line ℓ in the combined structure, we search with its point p_ℓ in the primal substructure, in $O(\log n)$ time (with a constant of proportionality that depends on δ; see below), to locate the leaf cell ω that contains p_ℓ (using the properties that (a) each recursive step involves a partitioning of constant size, and (b) the cells in the partition are pairwise disjoint). We then search with Q_ℓ in the dual structure at ω, which takes, as just noted, $O^*(n/s^{1/6})$ time. The overall cost of the query is therefore $O^*(n/s^{1/6})$.

As to the actual storage used by the structure, the allocation mechanism ensures that each level of the recursion uses storage that is at most $7/4$ times larger than the storage used in the previous level, because each node has seven child collections $\Omega_0, \ldots, \Omega_6$, each of which is allocated an amount of storage $s/4$. Hence the overall storage used is $O((7/4)^j s)$, where j is the recursion depth. Arguing as in the query time analysis, we can make the factor $(7/4)^j$ to be $O(s^\delta)$, for any $\delta > 0$. That is, the overall storage used is $O(s^{1+\delta})$, or, in our notation, $O^*(s)$.

The above description of the structure applies to any single level among the four latter levels of the structure. The first two levels are considerably simpler and more efficient. The primal-dual approach is straightforward for halfspace range searching, and the parametric dimension is only four for the first two levels. The standard machinery (reviewed, e.g., in [1]) implies that, with s storage and N input tetrahedra, the cost of a query at each of these levels is $O^*(N/s^{1/4})$.

Putting everything together, and using standard arguments in the analysis of multi-level structures (see [1, Theorem 6.1] for details), the overall size of the six-level structure is $O^*(s)$, for any prescribed storage parameter s between n and n^6, and a query takes $O^*(n/s^{1/6})$ time. That is, this finally concludes the proof of Theorem 1 for the case of intersection detection queries. Counting and reporting queries are handled similarly, with a similar analysis, exploiting the fact that the decomposition in Theorem 8 is into disjoint subsets, as is a similar decomposition used in the machinery of [11]. For reporting query, their cost involves an additional term $O(k)$, where k is the output size. □

Remark. Our mechanism is in fact a special instantiation of the following general result, which is of independent interest, and which yields a trade-off bound for semi-algebraic range searching in any dimension d. That is, consider a general problem of this kind, that involves n points in $\mathbb{R}^d$, and aims to answer semi-algebraic range queries, where the ranges have constant complexity, and each range has d degrees of freedom (so the problem has a symmetric dual version). One can solve such a problem in time $O^*(n/s^{1/d})$ per query, using $O^*(s)$ space and preprocessing, where s is any parameter between n and n^d. These queries include detecting whether a query range contains any point from the input, counting the number of such points, or reporting them (with an additional term $O(k)$ in the query cost, where k is the output size). Using duality, we obtain the same performance bounds for point-enclosure queries, where the input consists on n constant-complexity semi-algebraic regions in $\mathbb{R}^d$, and the query is with a point p, where the goal is to detect, count or report containments of p in the input regions. The same asymptotic bound is obtained for simplex range searching [1], but our analysis shows that this bound corresponds to a much more general family of query ranges. The two extreme cases $s = n$ and $s = n^d$ have been treated in [11] and [2], respectively, but the tradeoff between these extreme cases has not been treated explicitly (for $d > 4$), as far as we can tell. As evidenced in the preceding analysis, this tradeoff is not as routine as one might think, because of the complicated nature of the partitioning used in Theorem 8 (as well as in [11, Theorem 1.1]). We summarize this result in the following corollary:

▶ **Theorem 9.** *Let P be a set of n points in $\mathbb{R}^d$, for any dimension d, and let Γ be a family of semi-algebraic ranges of constant complexity in $\mathbb{R}^d$, each of which has d degrees of freedom. Let $n \leq s \leq n^d$ be a prespecified storage parameter. Then one can preprocess P into a data structure of storage and preprocessing $O^*(s)$, such that a range-query, with a range $\gamma \in \Gamma$, can be answered in $O^*(n/s^{1/d})$ time. Such queries include detecting whether γ contains any point of P, counting the number of such points, and reporting them (with an additional $O(k)$ term in the latter case, where k is the number of these points). The same performance bounds apply to the dual point-enclosure case, where the input consists of n regions from Γ and the query is with a point $p \in \mathbb{R}^d$.*

Remarks.

(i) Theorem 9 can be extended to the case where the number of degrees of freedom of the ranges is different from d, but the resulting performance bound has a more complicated expression, which is not spelled out in this work.

(ii) We note that our technique can be extended to segment intersection detection queries amid a collection of n $(d-1)$-simplices in any dimension d. In that case the structure has $d+2$ levels. The first two levels ensure that the endpoints of the query segment e lie on different sides of the hyperplane containing the input simplex Δ, and are implemented by halfspace range searching structures in $\mathbb{R}^d$. The last d levels ensure that the line containing e has positive orientation with respect to each of the $(d-2)$-flats containing the facets of Δ, with suitable orientations of the line and the flats. Since lines and $(d-2)$-flats in $\mathbb{R}^d$ have $2d-2$ degrees of freedom, these levels are implemented using semi-algebraic range searching structures, where both primal and dual parts are in $\mathbb{R}^{2d-2}$. Hence the cost of the query at each of the last d levels dominates the overall cost, which is thus $O^*(n/s^{1/(2d-2)})$. The parameter s can vary between n and n^{2d-2}.

(iii) A very similar mechanism, with the same performance bounds, handles the reverse situation, mentioned in the introduction, in which the input is a set of n segments in $\mathbb{R}^4$, and the query is with a tetrahedron T, and the goal is to detect, count, or report intersections between T and the input segments. This property is stated in Theorem 7; we defer the relatively easy details to the full version.

3 Triangle-Triangle Intersection Queries in $\mathbb{R}^4$

Let $\mathbf{\Delta}$ be a set of n triangles in $\mathbb{R}^4$. We consider various triangle-triangle intersection problems, the simplest of which is just to detect whether the query triangle intersects any input triangle. Alternatively, we may want to count or to report all such intersections. For concreteness we consider only the detection problem in what follows, but, as in the previous section, the algorithm can be extended to also handle the other kinds of problems.

Similar to the preceding section, we use a multi-level data structure, where each level caters to one aspect of the condition that a triangle crosses another triangle. Specifically, let Δ_1 and Δ_2 be two triangles, and let π_1, π_2 be the respective 2-planes that contain them. Assuming general position, π_1 and π_2 always intersect at a single point ξ, and Δ_1 intersects Δ_2 if and only if ξ belongs to both triangles. As is easily verified, this latter condition is equivalent, with suitable orientations of π_1, π_2, and of the lines supporting the edges of both triangles, to the conjunction of the following conditions:

(i) π_1 is positively oriented with respect to each of the lines that support the edges of Δ_2.

(ii) π_2 is positively oriented with respect to each of the lines that support the edges of Δ_1.

Conditions (i) and (ii) are the conjunction of a total of six sub-conditions, each of which tests the orientation of, say, the 2-plane π_1 with respect to the line supporting some specific edge of Δ_2, or vice versa.

We can therefore apply a suitable variant of the same machinery of the preceding section, and obtain a proof of Theorem 3.

The batched bichromatic version. For the batched version of the triangle-triangle intersection problem, with m red triangles and n blue triangles (see Theorem 4), we choose the storage parameter s to be such that the cost of m queries with the red triangles is asymptotically roughly the same as the cost of preprocessing the blue triangles. That is, we set $s = mn/s^{1/6}$, or $s = m^{6/7}n^{6/7}$. For this choice to make sense, we need to ensure that $n \leq s \leq n^6$, or that $n^{1/6} \leq m \leq n^6$. When $m > n^6$ we only use the data structure of [2] and obtain the running time $O^*(m + n^6) = O^*(m)$, and when $m < n^{1/6}$ we only use the data structure of [11] and obtain the running time $O^*(mn^{5/6} + n) = O^*(n)$. Altogether we obtain the bound in Theorem 4.

4 Segment-Intersection amid Tetrahedra: An Improved Solution

In this section we present an improved algorithm for setup (i) of the paper, for a data structure of roughly quadratic size. Let $\mathcal{T}$ be a collection of n tetrahedra in $\mathbb{R}^4$. Our improved solution constructs a data structure that uses $O^*(n^2)$ storage (and expected preprocessing time), and answers a query in $O^*(n^{1/2})$ time. This is indeed a significant improvement over the standard algorithm in Section 2, in which, with storage $O^*(n^2)$, the query cost is $O^*(n^{2/3})$. With a suitable tradeoff, presented in the full version, the improvement can be extended for any storage parameter between n and n^6, although it is most substantial when the storage is nearly quadratic; see Figure 1.

Assume, without loss of generality, that the query segment is bounded. The algorithm constructs a partitioning polynomial F in $\mathbb{R}^4$ of degree $O(D)$, for some large but constant parameter D, so that each cell of the partition is crossed by at most n/D^2 2-faces of the tetrahedra in $\mathcal{T}$ and by a total of at most n/D tetrahedra. The existence of such a polynomial follows from Guth [8], and an expected linear-time algorithm for its construction (for constant D) is given in [2]. We classify each tetrahedron $\Delta \in \mathcal{T}$ as being *narrow* (resp., *wide*) with respect to a partition cell τ if a 2-face of Δ crosses τ (resp., Δ crosses τ but none of its 2-faces crosses τ). Let $\mathcal{N}_\tau$ (resp., $\mathcal{W}_\tau$) denote the set of narrow (resp., wide) tetrahedra at τ.

There are two cases to consider in our analysis, depending on whether the query segment ρ is contained or not contained in the zero set $Z(F)$ of F. Each of these cases requires its own data structure. The latter case is an extension of the analysis in [7] (given there for the three-dimensional version of the problem), and the case where $\rho \subset Z(F)$ requires a different approach than that taken in [7] for handling queries on the zero set. Due to lack of space we only sketch the general framework; the details are given in the full version.

A sketch of the analysis. A query segment ρ that is not contained in $Z(F)$ crosses at most $O(D)$ cells of the partition. For each partition cell τ (an open connected component of $\mathbb{R}^4 \setminus Z(F)$) we construct an auxiliary data structure on the wide tetrahedra at τ, and preprocess the narrow tetrahedra at τ recursively. As we show below, the structure for the wide tetrahedra uses $S_0(n) = O^*(n^2)$ storage, and a query amid them takes $Q_0(n) = O^*(n^{1/2})$ time. We then output the wide tetrahedron returned by querying the auxiliary structure at τ, if such a tetrahedron exists. Otherwise, we return the tetrahedron produced by the

recursive call, if one exists. If no tetrahedron, wide or narrow, has been found, we proceed to the next cell τ' crossed by ρ, repeat the whole procedure at τ', and keep doing this till we either find a tetrahedron hit by ρ or run out of cells, and then conclude that ρ does not hit any tetrahedron of $\mathcal{T}$.

The correctness of this procedure is clear (modulo that of the procedure for handling wide tetrahedra). Denote by $S(n)$ (resp., $Q(n)$) the maximum storage (resp., query time) required by the overall structure for n tetrahedra. Also denote by $S_1(n)$ (resp., $Q_1(n)$) the maximum storage (resp., query time) required for processing the input tetrahedra for intersection queries with segments contained in $Z(F)$, for any set of n tetrahedra in $\mathbb{R}^4$. We then have, for a suitable absolute constant $c > 0$ (where the constant hidden in the $O_D(\cdot)$ notation depends on D),

$$S(n) = O_D(S_0(n/D)) + S_1(n) + cD^4 S(n/D^2)$$
$$Q(n) = \max\left\{O_D(Q_0(n/D)) + cDQ(n/D^2),\ Q_1(n)\right\}.$$

We show, in the full version, that $S_1(n) = O_D^*(n^2)$ and $Q_1(n) = O_D^*(n^{1/2})$. Substituting these bounds, as well as the bounds for $S_0(n)$ and $Q_0(n)$, the solutions of these recurrences is $S(n) = O^*(n^2)$ and $Q(n) = O^*(n^{1/2})$. This establishes Theorem 2.

Handling the wide tetrahedra. Handling the wide tetrahedra is done via a secondary recursion, as follows. We choose some large constant parameter $r_0 \gg D$, and partition $\partial\tau$ into $O_D(1)$ $x_1x_2x_3$-monotone strata (assuming a generic choice of the coordinate frame). This is fairly standard to do, see, e.g. [5]. We construct, for each stratum σ, a $(1/r_0)$-cutting for the set of (constant-degree algebraic) 2-surfaces of intersection of σ with the wide tetrahedra in $\mathcal{W}_\tau$. The cutting is constructed by projecting σ and the 2-surfaces that it contains onto the $x_1x_2x_3$-subspace, constructing a $(1/r_0)$-cutting, within that subspace, on the projected surfaces, and then lifting the resulting cutting back to σ. Using standard results on vertical decomposition in three dimensions (see, e.g., [13]) and the theory of cuttings [9], we obtain $O^*(r_0^3)$ cells of the cutting (referred to as (pseudo-)prisms, in accordance with the way in which the vertical-decomposition–based cutting is constructed), each of which is crossed by (intersects but not contained in) at most n/r_0 wide tetrahedra.

For each pair ψ_1, ψ_2 of prisms, we define S_{ψ_1,ψ_2} to be the set of all segments e so that e has an endpoint in ψ_1 and an endpoint in ψ_2, and the relative interior of e is fully contained in τ. Clearly, S_{ψ_1,ψ_2} is a semi-algebraic set of constant complexity in a 6-dimensional parametric space,[5] and we decompose it into its $O(1)$ connected components.

For each segment $e \in S_{\psi_1,\psi_2}$, let $\mathcal{T}(e)$ denote the set of all wide tetrahedra Δ of $\mathcal{W}_\tau$ that e crosses. We have the following crucial technical lemma, akin to Lemma 2.2 in [7]:

▶ **Lemma 10.** *Each connected component C of S_{ψ_1,ψ_2} can be associated with a fixed set $\mathcal{T}_C$ of wide tetrahedra Δ of $\mathcal{W}_\tau$, none of which crosses $\psi_1 \cup \psi_2$, so that, for each segment $e \in C$, $\mathcal{T}_C \subseteq \mathcal{T}(e)$, and each tetrahedron Δ in $\mathcal{T}(e) \setminus \mathcal{T}_C$ crosses either ψ_1 or ψ_2.*

We illustrate the proof in Figure 10, and delegate the rest of the details to the full version of this paper.

[5] Each segment is specified by its two endpoints; since they lie on $\partial\tau$, each has three degrees of freedom.

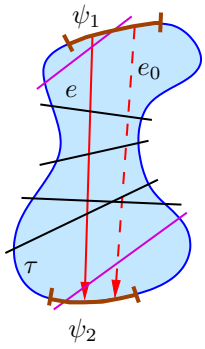

Figure 2 The set $\mathcal{T}_C$ (consisting of the tetrahedra depicted as black segments), and an illustration of the proof of Lemma 10: The tetrahedra that cross some fixed segment e_0 between ψ_1 and ψ_2 are the same tetrahedra that cross any other such segment e, except for those that cross ψ_1 or ψ_2 (like those depicted as magenta segments).

The analysis for wide tetrahedra. For each prism ψ, the *conflict list* K_ψ of ψ is the set of all wide tetrahedra that cross ψ. By construction, $|K_\psi| \leq n/r_0$. The same bound for crossing tetrahedra holds when ψ is lower-dimensional. If a lower-dimensional prism is contained in some tetrahedron there is no need to process ψ further, since any segment that meets ψ hits all these tetrahedra.

For each pair of prisms ψ_1, ψ_2, we compute S_{ψ_1,ψ_2} and decompose it into its connected components. For each component C we compute the set $\mathcal{T}_C$ of the wide tetrahedra, as in Lemma 10 (see the full version for details). This requires $O_D(r_0^6 n) = O_D(n)$ storage and computation time.

Let s be the storage parameter associated with the problem; we require that $n \leq s \leq n^3$. For each canonical set $\mathcal{T}_C$, we replace its tetrahedra by their supporting hyperplanes, and preprocess the resulting collection of hyperplanes for efficient segment intersection queries amid hyperplanes in $\mathbb{R}^4$. Using the technique of Agarwal and Matoušek [3], this problem can be solved using $O^*(s)$ storage (and preprocessing), and a query takes $O(n \operatorname{polylog}(n)/s^{1/4}) = O^*(n/s^{1/4})$ time (see also [1]). Lemma 10 guarantees the correctness of this procedure (namely, that replacing each tetrahedron in $\mathcal{T}_C$ by its supporting hyperplane does not cause any "false positive" answer).

We now process recursively each conflict list K_ψ, over all prisms ψ of the partition. Each recursive subproblem uses the same parameter r_0, but the allocated storage parameter is now s/r_0^3. We keep recursing until we reach conflict lists of size close to $n^{3/2}/s^{1/2}$. More precisely, after j levels of recursion, we get a total of at most $(c_0 r_0^3)^j = c_0^j r_0^{3j}$ subproblems, each involving at most n/r_0^j wide tetrahedra, for some constant c_0 that depend on D (but is considerably smaller than r_0).

We stop the recursion at the first level j^* at which $\frac{n}{r_0^{j^*}} \leq n^{3/2}/s^{1/2}$. As a result, we have ${r_0}^{j^*} = O(s^{1/2}/n^{1/2})$, and we get $c_0^{j^*} {r_0^{3j^*}} = O^*(s^{3/2}/n^{3/2})$ subproblems. Each of these subproblems involves at most $\frac{n}{r_0^{j^*}} = O^*\left(\frac{n^{3/2}}{s^{1/2}}\right)$ tetrahedra. Hence the overall size of the inputs, as well as of the canonical sets, at all the subproblems throughout the recursion, is $O^*\left(\frac{s^{3/2}}{n^{3/2}} \cdot \frac{n^{3/2}}{s^{1/2}}\right) = O^*(s)$. In particular, this is the asymptotic cost at the bottom level of the recursion.

As just described, at the bottom of the recursion, each subproblem contains at most $O^*(n^{3/2}/s^{1/2})$ wide tetrahedra, and we detect intersections with them by brute force. We thus obtain the following recurrence for the overall storage $S_0(N_W, s_W)$ for the structure constructed on N_W wide tetrahedra, where s_W denotes the storage parameter allocated to the structure (at the root $N_W = n$, $s_W = s$).

$$S_0(N_W, s_W) = \left\{ \begin{array}{ll} O_D^*(r_0^6 s_W) + c_0 r_0^3 S_0\left(\frac{N_W}{r_0}, \frac{s_W}{r_0^3}\right) & \text{for } N_W \geq \Theta^*(n^{3/2}/s^{1/2}), \\ O(N_W) & \text{for } N_W < \Theta^*(n^{3/2}/s^{1/2}). \end{array} \right\}$$

Unfolding the recurrence up to the terminal level j^*, where $N_W = O^*(n^{3/2}/s^{1/2})$, the sum of the nonrecursive overhead terms, over all nodes at a fixed level j, is

$$c_0^j r_0^{3j} \cdot O^*\left(\frac{s_W}{r_0^{3j}}\right) = O^*\left(s_W\right).$$

Hence, starting the recurrence at $(N_W, s_w) = (n, s)$, the overall contribution of the overhead terms is $O^*(s)$. We showed above that this is also the asymptotic cost at the bottom of the recurrence. Therefore, the overall storage used by the data structure is $O^*(s)$. Using similar considerations, one can show that the overall expected preprocessing time is $O^*(s)$ as well, since the time obeys a similar asymptotic recurrence.

Answering a query. Given a query segment ρ, which is not contained in $Z(F)$, we find its $O(D)$ intersections with $Z(F)$, which decompose it into $O(D)$ segments, each fully contained in some partition cell. Moreover, except for the first and last segment, the endpoints of each of the other segments lie on the boundary of its cell. We process the segments in their order[6] along ρ. Let e be the currently processed segment. If e is not the first or last segment, we find the prisms ψ_1, ψ_2 that contain its endpoints, and find the component C of S_{ψ_1,ψ_2} that contains e. If e is the first or last segment, we extend it backwards or forwards, respectively, till it meets the boundary of its cell, and call the resulting segment e'. We now compute for e' the corresponding set S_{ψ_1,ψ_2} and its component C that contains e'. Since D and r_0 are constants, all this takes constant time.

The query, on the wide tetrahedra at the present cell τ, performs a segment intersection detection query with e (or with e' when e is the first or last segment) in the set of hyperplanes containing the tetrahedra of $\mathcal{T}_C$, and, if no intersection is detected, continues recursively with $\mathcal{T}_{\psi_1}$ and $\mathcal{T}_{\psi_2}$ (at the bottom of recursion we apply a brute-force search). If no tetrahedron is found, in all the r_0-recursive steps, we conclude that (the present subsegment of) ρ does not hit any wide tetrahedron within τ. Once again, the correctness of this procedure follows from Lemma 10.

The query time $Q_0(N_W, s_W)$ satisfies the recurrence

$$Q_0(N_W, s_W) = \left\{ \begin{array}{ll} O_D(1) + O^*\left(\frac{N_W}{s_W^{1/4}}\right) + 2Q\left(\frac{N_W}{r_0}, \frac{s_W}{r_0^3}\right) & \text{for } N_W \geq \Theta^*(n^{3/2}/s^{1/2}), \\ O(N_W) & \text{for } N_W < \Theta^*(n^{3/2}/s^{1/2}). \end{array} \right\}$$

Unfolding the recurrence, the overall bound for the nonrecursive overhead terms, starting from $(N_W, s_W) = (n, s)$, is at most

$$O^*\left(\sum_{j \geq 0} \left(\frac{2}{r_0^{1/4}}\right)^j \cdot \frac{n}{s^{1/4}}\right) = O^*\left(\frac{n}{s^{1/4}}\right).$$

[6] The order is immaterial for segment intersection detection queries, but is important for ray shooting.

Adding the cost at the 2^{j^*} subproblems at the bottom level j^* of the recursion, where the cost of each subproblem is at most $O^*(n^{3/2}/s^{1/2})$, we obtain the query time

$$Q_0(n,s) = O^*\left(\frac{n}{s^{1/4}} + \frac{n^{3/2}}{s^{1/2}}\right). \tag{3}$$

Therefore, for $s = n^2$ the query time is $O^*(n^{1/2})$. The bounds $S_0(n) := S_0(n, n^2) = O^*(n^2)$ and $Q_0(n) := Q_0(n, n^2) = O^*(n^{1/2})$ are the bounds promised earlier for the wide tetrahedra at a cell.

5 Output-Sensitive Construction of Arrangements of Tetrahedra and of Intersections of Polyhedra in $\mathbb{R}^4$

The results of Section 3 can be applied to construct the arrangement $\mathcal{A}(\mathcal{T})$ of a set $\mathcal{T}$ of n tetrahedra in $\mathbb{R}^4$ in an output-sensitive manner. A complete discrete representation of $\mathcal{A}(\mathcal{T})$ requires, at the least, the collection of all faces, of all dimensions, of the arrangement, and their adjacency structure. Concretely, for each j-dimensional face φ, for $j = 0, 1, 2, 3$, we want the set of all $(j+1)$-dimensional faces that have φ on their boundary. Conversely, for each j-dimensional face φ, for $j = 1, 2, 3, 4$, we want the set of all $(j-1)$-dimensional faces that comprise $\partial\varphi$.

We begin by considering the task of computing all the nonempty intersections of pairs, triples, and quadruples of tetrahedra of $\mathcal{T}$. This will yield the set of vertices, and provide an infrastructure for computing the j-faces, for $j = 1, 2, 3$. Denote the number of these intersections as k_2, k_3, and k_4, respectively. Note that we always have $k_4 \geq k_3 \geq k_2$.

To simplify the description we assume that the tetrahedra are in general position, although a suitable adaptation of the following machinery can handle degenerate cases too.

Reporting pairwise intersections. Two tetrahedra in general position in $\mathbb{R}^4$ intersect in a two-dimensional convex polygon of constant complexity, and it suffices to report one vertex of each nonempty polygon, in order to detect all intersecting pairs of tetrahedra. As is easily checked, such a vertex is either an intersection of an edge of one tetrahedron with the other tetrahedron, or an intersection of two 2-faces (triangles), one from each tetrahedron.

Reporting vertices of the first kind (edge-tetrahedron intersections) can be done using the machinery in Theorem 1, whose details are provided in Section 2, which takes $O^*(n^{12/7} + k_2)$ time.[7] Reporting vertices of the second kind (triangle-triangle intersections) is done using the machinery in Section 3, which also takes $O^*(n^{12/7} + k_2)$ time.

Reporting triple and quadruple intersections. We iterate over the input tetrahedra. For each fixed tetrahedron T_0, the previous step provides us with all the other tetrahedra that intersect T_0. Denote their number as k_{T_0}, and observe that $\sum_{T_0} k_{T_0} = 2k_2$. We form the nonempty intersections $T_0 \cap T$, and triangulate each of them. We obtain a collection of $O(k_{T_0})$ triangles, all contained in (T_0 and therefore also in) the hyperplane h_{T_0} supporting T_0.

We have thus reduced our problem to that of reporting all pairwise and triple intersections in a set of $m = O(k_{T_0})$ triangles in $\mathbb{R}^3$. This can be solved using the algorithm in [7], by a procedure that runs in $O^*(m^{3/2} + \ell_{T_0})$ time, where ℓ_{T_0} is the number of triple intersections of the triangles. Note that $\sum_{T_0} \ell_{T_0} = O(k_4)$.

[7] Although this part can be performed faster, as described in Section 4, we use the standard solution, since we do not have a similar improvement for the construction of vertices of the second kind.

Adding up this cost over all tetrahedra T_0, the overall running time is

$$O^*\left(\sum_T k_T^{3/2} + k_4\right) = O^*\left(n^{1/2}\sum_T k_T + k_4\right) = O^*(n^{1/2}k_2 + k_4).$$

Constructing the arrangement. For each tetrahedron T_0, it is fairly routine to obtain, from the information collected so far, the full three-dimensional arrangement within T_0, using standard techniques in three dimensions; we omit here these standard details. This gives us all the j-faces of the four-dimensional arrangement $\mathcal{A}$, for $j = 0, 1, 2, 3$, and their adjacency information. The local adjacency information in $\mathbb{R}^4$ is also available from this data. By local adjacency we mean the adjacency between a j-face and the j'-faces on its boundary, for $j' < j$, over all such pairs of faces. For completion we need to identify disconnected pieces of the boundary of each four-dimensional cell, and record their adjacency to that cell. This can be done by x_4-vertical ray shooting from the x_4-highest point of each connected three-dimensional complex of faces. This calls for performing $O(n)$ x_4-vertical ray shooting queries in a set of n tetrahedra in $\mathbb{R}^4$, which can be done using the machinery presented in Theorem 1, or by an even simpler mechanism (since all the rays are vertical).

We have thus established the bound stated in Theorem 6.

Output-sensitive construction of the intersection of polyhedra in $\mathbb{R}^4$. As another application, consider the problem where we have two not necessarily convex polyhedra R and B in $\mathbb{R}^4$, whose boundaries consist of, or can be triangulated into $O(n)$ faces of all dimensions, which are segments, triangles, and tetrahedra. The goal is to construct their intersection $R \cap B$ in an output-sensitive manner; a similar application has been shown in [7] for the three-dimensional problem. We note that computing the union $B \cup R$ can be done using a very similar approach, within the same asymptotic time bound.

In order to compute $R \cap B$, we first apply the above algorithm to construct, in an output-sensitive manner, the arrangement $\mathcal{A}(R \cup B)$ of the two polyhedra R and B (specifically, we build the arrangement of the tetrahedra comprising the boundaries of B and R). We then label each cell (of any dimension) of $\mathcal{A}(R \cup B)$ with the appropriate Boolean operation, that is, whether it either lies in $R \setminus B$, $B \setminus R$, $B \cap R$, or in the complement of $B \cup R$. Collecting all the cells of the desired kind (e.g., those in $B \cap R$), and computing the adjacency relation between them, we obtain a suitable representation of the intersection. This establishes the bound stated in Theorem 6(ii).

We comment that extending the analysis to the intersection of more than two (albeit, still a constant number of) input polyhedra can also be done, following the same machinery as in the construction of an arrangement of tetrahedra, as presented above. It is easy to verify that in this case we obtain the same asymptotic bound stated in Theorem 6(ii).

References

1 P. K. Agarwal. Simplex range searching and its variants: A review. In *Journey through Discrete Mathematics: A Tribute to Jiří Matoušek*, pages 1–30. Springer Verlag, Berlin-Heidelberg, 2017.

2 P. K. Agarwal, B. Aronov, E. Ezra, and J. Zahl. An efficient algorithm for generalized polynomial partitioning and its applications. *SIAM J. Comput.*, 50:760–787, 2021. Also in *Proc. Sympos. on Computational Geometry (SoCG)*, 2019, 5:1–5:14. Also in `arXiv:1812.10269`.

3 P. K. Agarwal and J. Matoušek. Ray shooting and parameric search. *SIAM J. Comput.*, 22:794–806, 1993.

4 P. K. Agarwal, J. Matoušek, and M. Sharir. On range searching with semialgebraic sets II. *SIAM J. Comput.*, 42:2039–2062, 2013. Also in `arXiv:1208.3384`.

5 S. Basu, R. Pollcak, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer-Verlag, Berlin–Heidelberg, 2nd edition, 2006.

6 J. Canny. Collision detection for moving polyhedra. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 8:200–209, 1986.

7 E. Ezra and M. Sharir. On ray shooting for triangles in 3-space and related problems. *SIAM J. Comput.*, to appear. Also in Proc. 37th Sympos. on Computational Geometry, 2021, 34:1–34:15, and in `arXiv:2102.07310`.

8 L. Guth. Polynomial partitioning for a set of varieties. *Math. Proc. Camb. Phil. Soc.*, 159:459–469, 2015. Also in `arXiv:1410.8871`.

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

10 M. C. Lin, D. Manocha, and Y. J. Kim. Collision and proximity queries. In *Handbook on Discrete and Computational Geometry*, chapter 39, pages 1029–1056. CRC Press, Boca Raton, Florida, 3rd edition, 2017.

11 J. Matoušek and Z. Patáková. Multilevel polynomial partitions and simplified range searching. *Discrete Comput. Geom.*, 54:22–41, 2015.

12 E. Schömer and Ch. Thiel. Efficient collision detection for moving polyhedra. In *Proc. 11th Sympos. on Computational Geometry*, pages 51–60, 1995.

13 M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, Cambridge-New York-Melbourne, 1995.