

Efficient Tradeoff Schemes in Data Structures for Querying Moving Objects*

Pankaj K. Agarwal[†] Lars Arge[†] Jeff Erickson[‡] Hai Yu[†]

Abstract

The ability to represent and query continuously moving objects is important in many applications of spatio-temporal database systems. In this paper we develop data structures for answering various queries on moving objects, including range and proximity queries, and study tradeoffs between various performance measures—query time, data structure size, and accuracy of results.

1 Introduction

With the rapid advances in geographical positioning technologies, it has become feasible to track continuously moving objects accurately. In many applications, one would like to store these moving objects so that various queries on them can be answered quickly. For example, a mobile phone service provider may wish to know how many users are currently present in a specified area. Unfortunately, most traditional data structure techniques assume that data do not change over time. Therefore the problem of efficiently representing and querying moving objects has been extensively researched in both the database and computational geometry communities. Recently, several new techniques have been developed; see [1, 7, 9, 15, 17, 22] and references therein.

The kinetic data structure framework (KDS) proposed by Basch *et al.* [9] has been successfully applied to a variety of geometric problems to efficiently maintain data structures for moving objects. The key observation of KDS is that though the actual structure defined by a set of moving objects may change continuously over time, its combinatorial description changes only at discrete time instances. Although this framework has proven to be very useful for maintaining geometric structures, it is not clear whether it is the right framework if one simply wants to query moving objects: on the one hand, there is no need to maintain the underlying structure explicitly, and on the other hand the KDS maintains a geometric structure on the current configuration of objects, while one may actually wish to answer queries on the past as well as the future configurations of objects.

For example, one can use the KDS to maintain the convex hull of a set of linearly moving points [9]. While the points move, $O(n^2)$ events are processed when the combinatorial description of the convex hull changes, each requiring $O(\log^2 n)$ time. Using the maintained structure, one can easily determine whether a query point lies in the convex hull of the current configuration of points. But suppose one is interested in queries of the following form: “Does a point p lie in the convex hull of the point set at a

*A preliminary version of the paper was presented at the 12th Annual European Symposium on Algorithms. Research by the first and fourth authors is supported by NSF under grants CCR-0086013, EIA-9870724, EIA-0131905, and CCR-0204118, and by a grant from the U.S.–Israel Binational Science Foundation. Research by the second author is supported by NSF under grants EIA-9972879, CCR-9984099, EIA-0112849, and INT-0129182. Research by the third author is supported by NSF under grants CCR-0093348, DMR-0121695 and CCR-0219594.

[†]Department of Computer Science, Duke University, Durham, NC 27708-0129, USA. {[@cs.duke.edu](mailto:pankaj,large,fishhai)}

[‡]Department of Computer Science, University of Illinois, Urbana, IL 61801-2302, USA. jeffe@cs.uiuc.edu

future time t_q ?” In order to answer this query, there is no need to maintain the convex hull explicitly. The focus of this paper is to develop data structures for answering this and similar types of queries and to study tradeoffs between various performance measures such as query time, data structure size, and accuracy of results.

Problem statement. Let $S = \{p_1, p_2, \dots, p_n\}$ be a set of moving points in \mathbb{R}^1 or \mathbb{R}^2 . For any time t , let $p_i(t)$ be the position of p_i at time t , and let $S(t) = \{p_1(t), p_2(t), \dots, p_n(t)\}$ be the configuration of S at time t . We assume that each p_i moves along a straight line at some fixed speed. We are interested in answering the following queries.

- **Interval query:** Given an interval $I = [y_1, y_2] \subseteq \mathbb{R}^1$ and a time stamp t_q , report $S(t_q) \cap I$, that is, all k points of S that lie inside I at time t_q .
- **Rectangle query:** Given an axis-aligned rectangle $R \subseteq \mathbb{R}^2$ and a time stamp t_q , report $S(t_q) \cap R$, that is, all k points of S that lie inside R at time t_q .
- **Approximate nearest-neighbor query:** Given a query point $p \in \mathbb{R}^2$, a time stamp t_q and a parameter $\delta > 0$, report a point $p' \in S$ such that $d(p, p'(t_q)) \leq (1 + \delta) \cdot \min_{q \in S} d(p, q(t_q))$.
- **Approximate farthest-neighbor query:** Given a query point $p \in \mathbb{R}^2$, a time stamp t_q , and a parameter $0 < \delta < 1$, report a point $p' \in S$ such that $d(p, p'(t_q)) \geq (1 - \delta) \cdot \max_{q \in S} d(p, q(t_q))$.
- **Convex-hull query:** Given a query point $p \in \mathbb{R}^2$ and a time stamp t_q , determine whether p lies inside the convex hull of $S(t_q)$.

Previous results. Range trees and kd -trees are two widely used data structures for orthogonal range queries; both of these structures have been generalized to the kinetic setting. Agarwal *et al.* [1] developed a two-dimensional kinetic range tree that requires $O(n \log n / \log \log n)$ space and answers queries in $O(\log n + k)$ time.¹ Agarwal *et al.* [4] developed kinetic data structures for two variants of the standard kd -tree that can answer queries in $O(n^{1/2+\varepsilon} + k)$ time. These kinetic data structures can only answer queries about the current configuration of points, that is, where t_q is the current time.

Kollios *et al.* [17] proposed a linear-size data structure based on partition trees [18] for answering interval queries with arbitrary time stamps t_q , in time $O(n^{1/2+\varepsilon} + k)$. This result was later extended to \mathbb{R}^2 by Agarwal *et al.* [1] with the same space and query time bounds.

Agarwal *et al.* [1] were the first to propose *time-responsive* data structures, which answer queries about the current configuration of points more quickly than queries about the distant past or future. Agarwal *et al.* [1] developed a time-responsive data structure for interval and rectangle queries where the cost of any query is a monotonically increasing function of the difference between the query’s *time stamp* t_q and the current time; the worst-case query cost is $O(n^{1/2+\varepsilon} + k)$. However, no exact bounds on the query time were known except for a few special cases.

Fairly complicated time-responsive data structures with provable query time bounds were developed later in [2]. The query times for these structures are expressed in terms of *kinetic events*; a kinetic event occurs whenever two moving points in S momentarily share the same x -coordinate or the same y -coordinate. For a query with time stamp t_q , let $\varphi(t_q)$ denote the number kinetic events between t_q and the current time. The data structure uses $O(n \log n)$ space, answers interval queries in $O(\varphi(t_q)/n + \log n + k)$ time, and answers rectangle queries in $O(\sqrt{\varphi(t_q)} + (\sqrt{n}/\sqrt{\lceil \varphi(t_q)/n \rceil}) \log n + k)$ time. In the worst case, when $\varphi(t_q) = \Theta(n^2)$, these query times are no better than brute force.

¹Some of the previous work described in this section was actually done in the standard two-level I/O model aiming to minimize the number of I/Os. Here we interpret and state their results in the standard internal-memory setting.

Kollios *et al.* [16] described a data structure for one-dimensional nearest-neighbor queries, but did not give any bound on the query time. Later, an efficient structure was given by Agarwal *et al.* [1] to answer δ -approximate nearest-neighbor queries in $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ time using $O(n/\sqrt{\delta})$ space. Their data structure can also be used to answer δ -approximate farthest-neighbor queries; however, in both cases, the approximation parameter δ must be fixed at preprocessing time. More practical data structures to compute exact and approximate k -nearest-neighbors were proposed by Procopiuc *et al.* [20].

Basch *et al.* [9] described how to maintain the convex hull of a set of moving points in \mathbb{R}^2 under the KDS framework. They showed that each kinetic event can be handled in logarithmic time, and the total number of kinetic events is $O(n^{2+\varepsilon})$ if the trajectories are polynomials of fixed degree; the number of events is only $O(n^2)$ if points move linearly [5]. With a kinetic convex hull at hand, one can easily answer a convex-hull query in $O(\log n)$ time. However, as noted earlier, the kinetic convex hull can only answer queries on current configurations.

Our results. In this paper we present data structures for answering the five types of queries listed above. A main theme of our results is the existence of various tradeoffs: time-responsive data structures for which the query time depends on the value of t_q ; tradeoffs between query time and the accuracy of results; and tradeoffs between query time and the size of the data structure.

In Section 2, we describe a new time-responsive data structure for interval queries in \mathbb{R}^1 and rectangle queries in \mathbb{R}^2 . It uses $O(n^{1+\varepsilon})$ space and can answer queries in $O((\varphi(t_q)/n)^{1/2} + \text{polylog } n + k)$ time, where $\varphi(t_q)$ is the number of events between t_q and the current time. To appreciate how this new query time improves over those of previously known data structures, let us examine two extreme cases. For near-future or recent-past queries, where $\varphi(t_q) = O(n)$, previous structures answer interval and rectangle queries in $O(\log n)$ and $O(\sqrt{n} \log n)$ time respectively, while our structures answer both queries in $O(\text{polylog } n)$ time. In the worst case, where $\varphi(t_q) = \Omega(n^2)$, previous structures answer queries in $O(n)$ time, which is no better than brute force, while our structures provide a query time of roughly $O(\sqrt{n})$, which is the best known for any structure of near-linear size. Using this result, we also obtain the first time-responsive structures for approximate nearest- and farthest-neighbor queries.

In Section 3, we present a data structure for answering δ -approximate farthest-neighbor queries, which provides a tradeoff between the query time and the accuracy of the result. In particular, for any fixed parameter $m > 1$, we build a data structure of size $O(m)$ in $O(n + m^{7/4})$ time so that a δ -approximate farthest-neighbor query can be answered in $O(1/\delta^{9/4+\varepsilon})$ time, given a parameter $\delta \geq 1/m^{1/4}$ as a part of the query. Note that these bounds are all independent of the number of points in S .

In Section 4, we present data structures for answering convex-hull queries. We first describe data structures that produce a continuous tradeoff between space and query time, by interpolating between one data structure of linear size and another structure with logarithmic query time. We then present a data structure that provides a tradeoff between efficiency and accuracy for approximate convex-hull queries.

2 Time-Responsive Data Structures

2.1 One-dimensional interval queries

Preliminaries. Let L be a set of n lines in \mathbb{R}^2 , let Δ be a triangle, and let r be a parameter between 1 and n . A $(1/r)$ -cutting of L within Δ is a triangulation Ξ of Δ such that each triangle of Ξ intersects at most n/r lines in L . Chazelle [13] described an efficient hierarchical method to construct $1/r$ -

cuttings. In this approach, one chooses a sufficiently large constant ρ and sets $h = \lceil \log_\rho r \rceil$. One then constructs a sequence of cuttings $\Delta = \Xi_0, \Xi_1, \dots, \Xi_h = \Xi$, where Ξ_i is a $(1/\rho^i)$ -cutting of L . Ξ_i is obtained from Ξ_{i-1} by computing, for each triangle $\tau \in \Xi_{i-1}$, a $(1/\rho)$ -cutting Ξ_i^τ of L_τ within τ , where $L_\tau \subseteq L$ is the set of lines intersecting τ . Chazelle [13] proved that $|\Xi_i| \leq (c_1\rho)^i + c_2\mu\rho^{2i}/n^2$ for each i , where c_1, c_2 are constants and μ is the number of vertices of the arrangement of L inside Δ . Hence $|\Xi| = O(r^{1+\varepsilon} + \mu r^2/n^2)$ for any $\varepsilon > 0$, provided that ρ is sufficiently large. This hierarchy of cuttings can be represented as a *cutting tree* \mathcal{T} , where each node v is associated with a triangle Δ_v and the subset $L_v \subseteq L$ of lines that intersect Δ_v . The final cutting Ξ is the set of triangles associated with the leaves of \mathcal{T} . Ξ and \mathcal{T} can be constructed in time $O(nr^\varepsilon + \mu r/n)$.

High-level structure. Let $S = \{p_1, \dots, p_n\}$ be a set of n points in \mathbb{R}^1 , each moving linearly. We regard time t as an additional dimension; each moving point p_i traces a linear trajectory in the ty -plane, which we denote ℓ_i . Let $L = \{\ell_1, \dots, \ell_n\}$ be the set of all n trajectories, and let $\mathcal{A}(L)$ denote the arrangement of lines in L . Answering an interval query for interval $[y_1, y_2]$ and time stamp t_q is equivalent to reporting all lines in L that intersect the vertical line segment $\{t_q\} \times [y_1, y_2]$. We refer to this query as a *stabbing query*.

Each vertex in the arrangement $\mathcal{A}(L)$ corresponds to a change of y -ordering for a pair of points in S , or in other words, a *kinetic event*. As in [2], we divide the ty -plane into $O(\log n)$ vertical slabs as follows. Without loss of generality, suppose $t = 0$ is the current time. Consider the partial arrangement of $\mathcal{A}(L)$ that lies in the halfplane $t \geq 0$; the other side of the arrangement is handled symmetrically. For each $1 \leq i \leq \lceil \log_2 n \rceil$, let τ_i denote the t -coordinate of the $(2^i n)$ th vertex to the right of the line $t = 0$, and set $\tau_0 = 0$. Using an algorithm of Brönnimann and Chazelle [10], each τ_i can be computed in $O(n \log n)$ time. For each i , let \mathcal{W}_i denote the half-open vertical slab $[\tau_{i-1}, \tau_i) \times \mathbb{R}$. By construction, there are at most $2^i n$ kinetic events inside $\bigcup_{j \leq i} \mathcal{W}_j$. For each slab \mathcal{W}_i , we construct a separate *window data structure* \mathbb{W}_i for answering stabbing queries within \mathcal{W}_i . Our overall one-dimensional data structure consists of these $O(\log n)$ window data structures.

Window data structure for one-sided queries. Before we describe the window data structure \mathbb{W}_i for answering general stabbing queries in \mathcal{W}_i , we first describe a simpler structure \mathbb{W}_i^- to handle half-infinite query segments of the form $\{t_q\} \times (-\infty, y]$. Equivalently, we want to report the lines of L that lie below the query point (t_q, y) .

Set $r = n/2^i$. We construct a hierarchical $(1/r)$ -cutting Ξ inside \mathcal{W}_i , as described at the beginning of this section. The cutting tree \mathcal{T}_Ξ of Ξ forms the top part of our data structure \mathbb{W}_i^- ; see Figure 1. Recall that any node u in \mathcal{T}_Ξ is associated with a triangle Δ_u and the subset $L_u \subseteq L$ of lines that intersect Δ_u . For each leaf v of \mathcal{T}_Ξ , we build in time $O(n^{1+\varepsilon})$ a partition tree [18] for the set of points dual to the lines in L_v . For a query point $q = (t_q, y)$, all k lines of L_v lying below q can be reported in time $O((n/r)^{1/2} + k)$. These partition trees form the bottom part of \mathbb{W}_i^- . Let $p(u)$ denote the parent of a node u in \mathcal{T}_Ξ . At each node u of \mathcal{T}_Ξ except the root, we store an additional set $J_u^- \subseteq L_{p(u)}$ of lines that cross $\Delta_{p(u)}$ but lie below Δ_u . If u is at level j in the cutting tree, then $|J_u^-| \leq |L_{p(u)}| \leq n/\rho^{j-1}$. This completes the construction of \mathbb{W}_i^- .

Let $\mu \leq 2^i n$ denote the number of vertices of the arrangement $\mathcal{A}(L)$ within the slab \mathcal{W}_i . The total space required by \mathcal{T}_Ξ is

$$O\left(\sum_{u \in \mathcal{T}_\Xi} |J_u^-|\right) = O\left(\sum_{j=1}^{\log_\rho r} ((c_1\rho)^j + c_2\mu\rho^{2j}/n^2) \cdot (n/\rho^{j-1})\right) = O(nr^\varepsilon + \mu r/n),$$

for any constant $\varepsilon > 0$, provided that the value of ρ is chosen sufficiently large. Each partition

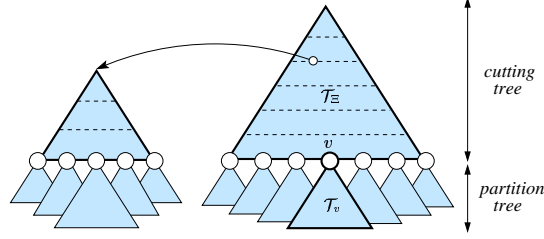


Figure 1. One window structure in our one-dimensional time-responsive data structure.

tree stored at a leaf of \mathcal{T}_Ξ requires $O((n/r)^{1+\varepsilon})$ space, and there are $O(r^{1+\varepsilon} + \mu r^2/n^2)$ such leaves. Substituting the values of μ and r , we conclude that the total space required by \mathbb{W}_i^- is $O(n^{1+\varepsilon})$. Similar arguments imply that \mathbb{W}_i^- can be constructed in $O(n^{1+\varepsilon})$ time.

Given a query point $q = (t_q, y)$ that lies in slab W_i , we report the lines of L below q as follows. We search down the tree \mathbb{W}_i^- to locate the leaf node v of \mathcal{T}_Ξ whose associated triangle Δ_v contains q . For each ancestor u of v , we report all lines in J_u^- . At the leaf node v , we use the partition tree of v to report all lines in L_v that lie below q . The correctness of the procedure is obvious. The total query time is $O(\log_\rho r + (n/r)^{1/2} + k) = O(\log n + 2^{i/2} + k)$, where k is the number of lines reported.

Our overall data structure consists of $O(\log n)$ of these window structures. The total space and preprocessing time is $O(n^{1+\varepsilon})$; the additional time required to locate the slab containing the query point is negligible. By construction, if the query point q lies in slab W_i , then $2^{i-1}n \leq \varphi(t_q) < 2^i n$.

Lemma 1 *Let S be a set of linearly moving points in \mathbb{R} . We can preprocess S in $O(n^{1+\varepsilon})$ time into a data structure of size $O(n^{1+\varepsilon})$ so that an interval query of the form $(t_q; -\infty, y)$ can be answered in $O(\log n + (\varphi(t_q)/n)^{1/2} + k)$ time, where k is the output size and $\varphi(t_q)$ is the number of kinetic events between t_q and the current time.*

A symmetric structure can clearly be used to answer interval queries of the form $(t_q; y, +\infty)$.

Window structure for query segments. We now describe the window data structure \mathbb{W}_i for answering arbitrary interval queries. Our structure is based on the simple observation that a line $\ell \in L$ intersects a segment $\gamma = \{t_q\} \times [y_1, y_2]$ if and only if ℓ intersects both $\gamma^- = \{t_q\} \times (-\infty, y_2]$ and $\gamma^+ = \{t_q\} \times [y_1, +\infty)$. Our structure \mathbb{W}_i consists of two levels. The first level finds all lines of L that intersect the downward ray γ^- as a union of a few “canonical” subsets, and the second level reports the lines in each canonical subset that intersect the upward ray γ^+ . See Figure 1.

We proceed as follows. Within each window W_i , we construct a cutting tree \mathcal{T}_Ξ on the trajectories L as before. For each node $u \in \mathcal{T}_\Xi$, let $J_u^- \subseteq L$ be defined as above. We construct a secondary structure \mathbb{W}_i^+ for J_u^- so that all k lines of J_u^- that lie above any query point can be reported in time $O(\log n + 2^{i/2} + k)$. Finally, for each leaf v of \mathcal{T}_Ξ , we construct in $O(2^i \log n)$ time a partition tree on the points dual to L_v , so that all k lines of L_v intersected by a vertical segment can be reported in time $O(2^{i/2} + k)$.

For a query segment $\gamma = \{t_q\} \times [y_1, y_2]$, we find the slab W_i that contains γ and search \mathcal{T}_Ξ of \mathbb{W}_i with the endpoint (t_q, y_2) . Let v be the leaf of \mathcal{T}_Ξ whose triangle Δ_v contains (t_q, y_2) . For each ancestor u of v , all the lines in J_u^- intersect the ray γ^- , so we query the secondary structure at u to report the lines in J_u^- that also intersect the ray γ^+ . The total query time is $O(\log^2 n + 2^{i/2} \log n + k)$. By being more careful, we can improve the query time to $O(\log^2 n + 2^{i/2} + k)$ without increasing the preprocessing time. If the query segment γ lies in slab W_i , then $2^{i-1}n \leq \varphi(t_q) < 2^i n$. Thus, the query time can be rewritten as $O(\log^2 n + (\varphi(t_q)/n)^{1/2} + k)$.

Our analysis assumes that there are $\Theta(2^i n)$ events between the current time and the right boundary of each slab \mathcal{W}_i . Thus, as time passes, we must occasionally update our data structure to maintain the same query times. Rather than making our structures kinetic, as in [1], we simply rebuild the entire structure after every $\Theta(n)$ kinetic events. This approach immediately gives us an amortized update time of $O(n^\varepsilon)$ per kinetic event; the update time can be made worst-case using standard lazy rebuilding techniques.

Putting everything together and choosing the value of ε carefully, we obtain the following theorem.

Theorem 1 *Let S be a set of n linearly moving points in \mathbb{R} . We can preprocess S in time $O(n^{1+\varepsilon})$ into a data structure of size $O(n^{1+\varepsilon})$, such that an interval query on S can be answered in $O(\log^2 n + (\varphi(t_q)/n)^{1/2} + k)$ time, where k is the output size and $\varphi(t_q)$ is the number of kinetic events between the query's time stamp t_q and the current time. Moreover, we can maintain this data structure in $O(n^\varepsilon)$ time per kinetic event.*

2.2 Two-dimensional rectangle queries

We now describe how to generalize our interval-query data structure to handle two-dimensional rectangle queries. Let $S = \{p_1, \dots, p_n\}$ be a set of n points in the plane, each moving with a fixed velocity. We again regard time t as an additional dimension. Let $L = \{\ell_1, \ell_2, \dots, \ell_n\}$ be the set of linear trajectories of points of S in txy -space. Given a time stamp t_q and a rectangle $R = R_x \times R_y$, where $R_x = [x_1, x_2]$ and $R_y = [y_1, y_2]$, a rectangle query is equivalent to reporting all lines in L that intersect the rectangle $\{t_q\} \times R$ in txy -space. For any line ℓ , let ℓ^x and ℓ^y denote the projections of ℓ onto the tx -plane and ty -plane, respectively, and define $L_x = \{\ell_1^x, \ell_2^x, \dots, \ell_n^x\}$ and $L_y = \{\ell_1^y, \ell_2^y, \dots, \ell_n^y\}$. Each vertex of $\mathcal{A}(L_x)$ or $\mathcal{A}(L_y)$ corresponds to a kinetic event. As for our one-dimensional structure, we divide txy -space into $O(\log n)$ slabs along the t -axis such that the number of kinetic events inside the i th slab \mathcal{W}_i is $O(2^i n)$, and build a window data structure \mathbb{W}_i for each \mathcal{W}_i .

Observe that a line ℓ intersects $\{t_q\} \times R$ if and only if ℓ^x intersects $\{t_q\} \times R_x$ and ℓ^y intersects $\{t_q\} \times R_y$. Thus we build a four-level data structure consisting of two one-dimensional structures described above, one on top of the other. Specifically, we first build a two-level one-dimensional window data structure \mathbb{W}_i^x for L_x as described in Section 2.1, with one minor modification: all partition trees in the structure are replaced by two-level partition trees so that a rectangle query can be answered in time $O(m^{1/2} + k)$, where m is the number of lines on which the partition tree is built and k is the output size, as described in [1]. For each node u of the cutting tree in the secondary data structure of \mathbb{W}_i^x , let J_u^x be the canonical subset of lines stored at u . Set $J_u^y = \{\ell_i^y \mid \ell_i^x \in J_u^x\}$. We then build a one-dimensional window structure $\mathbb{W}_{i,u}^y$ on J_u^y and store this structure at u .

A rectangle query can be answered by first finding the time interval \mathcal{W}_i containing the query rectangle $\{t_q\} \times R$. We then find the lines of L_x that intersect the segment R_x , as a union of canonical subsets. For each node u of \mathbb{W}_i^x in the output canonical subsets, we report all lines of J_u^y that intersect the segment R_y using the structure $\mathbb{W}_{i,u}^y$. Following standard analysis for multilevel data structures [3], we conclude the following.

Theorem 2 *Let S be a set of n linearly moving points in \mathbb{R}^2 . We can preprocess S in time $O(n^{1+\varepsilon})$ into a data structure of size $O(n^{1+\varepsilon})$, such that a rectangle query on S can be answered in $O(\text{polylog } n + (\varphi(t_q)/n)^{1/2} + k)$ time, where k is the output size and $\varphi(t_q)$ is the number of kinetic events between the query's time stamp t_q and the current time. Moreover, we can maintain this data structure in $O(n^\varepsilon)$ time per kinetic event.*

Combining this result with techniques developed in [1], we can construct time-responsive data

structures for δ -approximate nearest- or farthest-neighbor queries, for any fixed $\delta > 0$. Specifically, we can prove the following theorem.

Theorem 3 *Let S be a set of n linearly moving points in \mathbb{R}^2 . Given a parameter $\delta > 0$, we can preprocess S in time $O(n^{1+\varepsilon}/\sqrt{\delta})$ into a data structure of size $O(n^{1+\varepsilon}/\sqrt{\delta})$, such that an approximate nearest- or farthest-neighbor query on S can be answered in $O((\text{polylog } n + (\varphi(t_q)/n)^{1/2})/\sqrt{\delta})$ time, where $\varphi(t_q)$ is the number of kinetic events between the query's time stamp t_q and the current time. Moreover, we can maintain this structure in $O(n^\varepsilon/\sqrt{\delta})$ time per kinetic event.*

3 Approximate Farthest-Neighbor Queries

In this section, we describe a data structure for answering approximate farthest-neighbor queries that achieves a tradeoff between efficiency and accuracy. Unlike the approximate neighbor structure described in Theorem 3, the approximation parameter δ can be specified at query time, and the cost of a query depends only on δ , not on n . The general scheme is simple, but it crucially relies on the notion of ε -kernels introduced by Agarwal *et al.* [6].

For a unit vector $u \in \mathbb{S}^{d-1}$, the *directional width* of a point set $P \subset \mathbb{R}^d$ along u is defined to be $w_u(P) = \max_{p \in P} \langle u, p \rangle - \min_{p \in P} \langle u, p \rangle$. A subset $Q \subseteq P$ is called a δ -kernel of P if $w_u(Q) \geq (1 - \delta) \cdot w_u(P)$ for any unit vector $u \in \mathbb{S}^{d-1}$. A δ -kernel of size $O(1/\delta^{(d-1)/2})$ can be computed in $O(n + 1/\delta^{d-1})$ time [12, 24]. Using this general result and a result by Agarwal *et al.* on δ -kernels for moving points [6, Theorem 6.2], we can prove the following lemma.

Lemma 2 *Let $S = \{p_1, \dots, p_n\}$ be a set of n linearly moving points in \mathbb{R}^d . For any $0 < \delta < 1$, a subset $Q \subseteq S$ of size $O(1/\delta^{d+3/2})$ can be computed in $O(n + 1/\delta^{2d+3})$ time, so that for any time $t \in \mathbb{R}$ and any point $q \in \mathbb{R}^d$, we have*

$$(1 - \delta) \cdot \max_{p \in S} d(p(t), q) \leq \max_{p \in Q} d(p(t), q) \leq \max_{p \in S} d(p(t), q).$$

The subset Q in this lemma is referred to as a δ -kernel for δ -approximate farthest neighbors.

As we mentioned in Section 1, for a set of n linearly moving points in \mathbb{R}^2 , one can build in $O((n \log n)/\sqrt{\delta})$ time a data structure of size $O(n/\sqrt{\delta})$ that can answer δ -approximate farthest-neighbor queries in $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ time, for any fixed δ . By combining this result with Lemma 2 (for $d = 2$), we can construct in time $O(n + 1/\delta^7)$ a data structure of size $O(1/\delta^4)$ so that approximate farthest neighbors can be found in time $O(1/\delta^{9/4+\varepsilon})$ for any $\delta > 0$.

In order to allow δ to be specified at query time, we build data structures for several different values of δ . More precisely, fix a parameter $m > 1$. For each i between 0 and $\lg m$, let S_i be a δ_i -kernel of S , where $\delta_i = (2^i/m)^{1/4}$. Lemma 2 immediately implies that all $\lg m$ kernels S_i can be computed in $O(n \log m + m^{7/4})$ time. We can reduce the computation time to $O(n + m^{7/4})$ by constructing the kernels hierarchically: let S_0 be a δ_0 -kernel S , and for each $i > 0$, let S_i be a $O(\delta_i)$ -kernel of S_{i-1} . For each i , we build a data structure of size $O(|S_i|/\sqrt{\delta_i}) = O(m/2^i)$ for finding δ_i -approximate farthest neighbors in the kernel S_i . The total size of all $O(\log m)$ data structures is $O(m)$. Given a δ -approximate farthest-neighbor query, where $(1/m)^{1/4} \leq \delta < 1$, we first find an index i such that $2\delta_i \leq \delta < 2\delta_{i+1}$, and then query the data structure for the corresponding S_i . The returned point is a $1 - (1 - \delta_i)^2 < 2\delta_i \leq \delta$ approximate farthest neighbor of the query point. The overall query time is $O(|S_i|^{1/2+\varepsilon}/\sqrt{\delta_i}) = O((1/\delta)^{9/4+\varepsilon})$.

Theorem 4 *Let S be a set of n linearly moving points in \mathbb{R}^2 . For any parameter $m > 1$, a data structure of size $O(m)$ can be built in $O(n + m^{7/4})$ time so that for any $(1/m)^{1/4} \leq \delta < 1$, a δ -approximate farthest-neighbor query on S can be answered in $O((1/\delta)^{9/4+\varepsilon})$ time.*

4 Convex-Hull Queries

In this section, we describe data structures for convex-hull queries for linearly moving points in the plane that provide various tradeoffs. We begin with data structures that provide tradeoffs between space and query time by interpolating between two base structures—one with linear size, the other with logarithmic query time. We then show how to achieve a tradeoff between efficiency and accuracy, following the general spirit of Section 3.

Trading off space for query time. As in the previous section, let ℓ_i be the line traced by p_i through xyt -space and let $L = \{\ell_1, \ell_2, \dots, \ell_n\}$. We orient each line in the positive t -direction. Given a time stamp t_q and a query point $p \in \mathbb{R}^2$, let \bar{p} denote the point $(p, t_q) \in \mathbb{R}^3$ and let γ denote the plane $t = t_q$ in \mathbb{R}^3 . The convex-hull query now asks whether $\bar{p} \in \text{conv}(\gamma \cap L)$. Observe that $\bar{p} \notin \text{conv}(\gamma \cap L)$ if and only if some line $\ell \subset \gamma$ passes through \bar{p} and lies outside $\text{conv}(\gamma \cap L)$. Each of the two orientations of such a line ℓ has the same relative orientation² with respect to every line in L ; see Figure 2.

We use Plücker coordinates to detect the existence of such a line. Plücker coordinates map an oriented line ℓ in \mathbb{R}^3 to either a point $\pi(\ell)$ in \mathbb{R}^5 (referred to as the *Plücker point* of ℓ) or a hyperplane $\varpi(\ell)$ in \mathbb{R}^5 (referred to as the *Plücker hyperplane* of ℓ). Furthermore, one oriented line ℓ_1 has positive orientation with respect to another oriented line ℓ_2 if and only if $\pi(\ell_1)$ lies above $\varpi(\ell_2)$ [14]. The following lemma is straightforward.

Lemma 3 *For any plane γ in \mathbb{R}^3 and any point $\bar{p} \in \gamma$, the set of Plücker points of all oriented lines that lie in γ and pass through \bar{p} forms a line in \mathbb{R}^5 .*

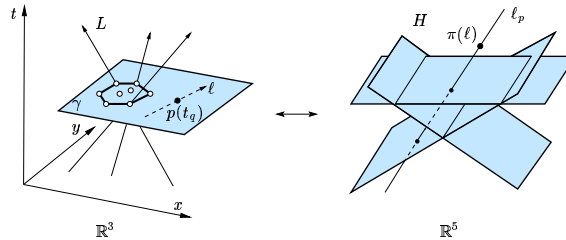


Figure 2. Plücker coordinates and convex-hull queries on moving points in \mathbb{R}^2 .

Let $H = \{\varpi(\ell_i) \mid \ell_i \in L\}$ be the set of Plücker hyperplanes in \mathbb{R}^5 of the oriented lines in L . For a query point $\bar{p} = (t_q, p)$, let Λ denote the line in \mathbb{R}^5 of Plücker points of oriented lines that lie in the plane γ and pass through \bar{p} . A line ℓ has the same relative orientation with respect to all lines in L if and only if $\pi(\ell)$ lies either above every hyperplane in H or below every hyperplane in H . Thus, our goal is to determine whether there is any Plücker point on the line Λ that lies on the same side of every Plücker hyperplane in H . This can be formulated as a 5-dimensional linear-programming query, which can be solved directly using a data structure of Matoušek [19] to obtain the following result.³

Theorem 5 *Let S be a set of n linearly moving points in \mathbb{R}^2 . Given a parameter $n \leq s \leq n^2$, a data structure of size $O(s^{1+\epsilon})$ can be built in $O(s^{1+\epsilon})$ time so that a convex-hull query on S can be answered in $O((n/\sqrt{s}) \text{polylog } n)$ time.*

²For any four points $a, b, c, d \in \mathbb{R}^3$, the *relative orientation* of the oriented line from a to b with respect to the oriented line from c to d is defined by the sign of the determinant of the 4-by-4 matrix $\begin{pmatrix} a & b & c & d \\ 1 & 1 & 1 & 1 \end{pmatrix}$.

³Data structures of Chan [11] and Ramos [21] can also be used for this purpose.

Logarithmic query time. If we are willing to use quadratic space, we can achieve $O(\log n)$ query time without using the complicated linear-programming query structure. Basch *et al.* [9] described how to kinetically maintain $\text{conv}(S(t))$ as t changes continuously, in $O(\log^2 n)$ time per kinetic event. Let μ denote the number of combinatorial changes to $\text{conv}(S(t))$; since the points in S are moving linearly, we have $\mu = O(n^2)$. The kinetic convex hull may undergo some additional *internal* events, but the number of internal events is also $O(n^2)$. Thus, the entire kinetic simulation takes $O(n^2 \log^2 n)$ time. Applying standard persistence techniques [23], we can record the combinatorial changes in $\text{conv}(S(t))$ in $O(n + \mu)$ space, so that for any time t_q , the combinatorial structure of $\text{conv}(S(t_q))$ can be accessed in $O(\log \mu) = O(\log n)$ time. With this combinatorial structure in hand, we can easily determine whether a point p lies inside or outside $\text{conv}(S(t_q))$ in $O(\log n)$ time.

Theorem 6 *Let S be a set of n linearly moving points in \mathbb{R}^2 , and let $\mu = O(n^2)$ be the total number of combinatorial changes to the convex hull of S over time. A data structure of size $O(n + \mu)$ can be built in $O(n^2 \log^2 n)$ time so that a convex-hull query on S can be answered in $O(\log n)$ time.*

Trading efficiency for accuracy. Using ideas from the previous section, we can also build a data structure for *approximate* convex-hull queries. Recall that a subset $Q \subseteq S$ is a δ -kernel of a set S of moving points in \mathbb{R}^d if $w_u(Q(t)) \geq (1 - \delta) \cdot w_u(S(t))$ for any time t and any unit vector $u \in \mathbb{S}^{d-1}$, where $w_u(\cdot)$ is the directional width function defined in Section 3. If the points in S are moving linearly, then for any δ , one can compute a δ -kernel of S of size $O(1/\delta^{3/2})$ in $O(n + 1/\delta^3)$ time [6, 12, 24]. To establish a tradeoff between efficiency and accuracy for convex-hull queries, we proceed as in Section 3.

Let $m > 1$ be a fixed parameter. For all i between 0 and $\lg m$, let $\delta_i = (2^i/m)^{1/3}$, and let S_i be a δ_i -kernel of S of size $O(1/\delta_i^{3/2}) = O((m/2^i)^{1/2})$. We first compute all S_i 's in $O(n + m)$ time as in Section 3, and then we build the logarithmic-query structure of Theorem 6 for each S_i . The size of one such data structure is $O(|S_i|^2) = O(m/2^i)$, so altogether these structures use $O(m)$ space. To answer a δ -approximate convex-hull query, where $(1/m)^{1/3} \leq \delta < 1$, we first find an index i such that $\delta_i \leq \delta < \delta_{i+1}$, and then query the corresponding data structure for S_i . The query time is $O(\log |S_i|) = O(\log(1/\delta))$.

Theorem 7 *Let S be a set of n linearly moving points in \mathbb{R}^2 . For any parameter $m > 1$, a data structure of size $O(m)$ can be built in $O(n + m^{2+\varepsilon})$ time so that for any $(1/m)^{1/3} \leq \delta < 1$, a δ -approximate convex-hull query on S can be answered in $O(\log(1/\delta))$ time.*

References

- [1] P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *J. Comput. Syst. Sci.*, 66 (2003), 207–243.
- [2] P. K. Agarwal, L. Arge, and J. Vahrenhold, Time responsive external data structures for moving points, *Proc. 7th Workshop on Algorithms and Data Structures*, 2001, pp. 50–61.
- [3] P. K. Agarwal, J. Erickson, Geometric range searching and its relatives, in *Advances in Discrete and Computational Geometry* (B. Chazelle, J. Goodman, R. Pollack, eds.), American Mathematical Society, Providence, RI, 1999, 1–56.
- [4] P. K. Agarwal, J. Gao, and L. Guibas, Kinetic medians and kd -trees, *Proc. 10th European Symposium on Algorithms*, 2002, pp. 5–16.
- [5] P. K. Agarwal, L. Guibas, J. Hershberg, and E. Veach, Maintaining the extent of a moving point set, *Discrete Comput. Geom.*, 26 (2001), 253–274.
- [6] P. K. Agarwal, S. Har-Peled, and K. Varadarajan. Approximating extent measure of points. *Journal of the ACM*, to appear.

- [7] P. K. Agarwal and C. M. Procopiuc, Advances in indexing for moving objects, *IEEE Bulletin of Data Engineering*, 25 (2002), 25–34.
- [8] P. K. Agarwal and M. Sharir, Arrangements and their applications, in *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), Elsevier Science Publishers, North-Holland, Amsterdam, 2000, 49–119.
- [9] J. Basch, L. Guibas, and J. Hershberger, Data structures for mobile data, *J. Algorithms*, 31(1) (1999), 1–28.
- [10] H. Brönnimann and B. Chazelle, Optimal slope selection via cuttings, *Comp. Geom. Theory & Appl.*, 10(1) (1998), 23–29.
- [11] T. M. Chan, Fixed-dimensional linear programming queries made easy, *Proc. 12th Annu. Sympos. Comput. Geom.*, 1996, pp. 284–290.
- [12] T. M. Chan, Faster core-set constructions and data stream algorithms in fixed dimensions, *Proc. 20th Annu. Sympos. Comput. Geom.*, 2004, pp. 152–159.
- [13] B. Chazelle, Cutting hyperplanes for divide-and-conquer, *Discrete Comput. Geom.*, 9 (1993), 145–158.
- [14] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Stolfi, Lines in space: Combinatorics and algorithms, *Algorithmica*, 15 (1996), 428–447.
- [15] A. Czumaj and C. Sohler, Soft kinetic data structures, *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001, pp. 865–872.
- [16] G. Kollios, D. Gunopulos, and V. Tsotras, Nearest neighbor queries in a mobile environment, *Proc. Intl. Workshop on Spatiotemporal Database Management*, 1999, pp. 119–134.
- [17] G. Kollios, D. Gunopulos, and V. Tsotras, On indexing mobile objects, *Proc. ACM Sympos. Principles Database Syst.*, 1999, pp. 261–272.
- [18] J. Matoušek, Efficient partition trees, *Discrete Comput. Geom.*, 8 (1992), 315–334.
- [19] J. Matoušek, Linear optimization queries, *J. Algorithms*, 14 (1993), 432–448.
- [20] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled, STAR-tree: an efficient self-adjusting index for moving points, *Proc 4th Workshop on Algorithms Engineering and Experiments*, 2002, pp. 178–193.
- [21] E. Ramos, Linear programming queries revisited, *Proc. 16th Annu. Sympos. Comput. Geom.*, 2000, pp. 176–181.
- [22] S. Saltenis, C. Jensen, S. Leutenegger, and M. López, Indexing the positions of continuously moving objects, *Proc. SIGMOD Intl. Conf. Management of Data*, 2000, pp. 331–342.
- [23] N. Sarnak and R. Tarjan, Planar point location using persistent search trees, *Communications of the ACM*, 29(7) (1986), 669–679.
- [24] H. Yu, P. K. Agarwal, R. Poreddy, and K. R. Varadarajan, Practical methods for shape fitting and kinetic data structures using core sets, *Proc. 20th Annu. Sympos. Comput. Geom.*, 2004, pp. 263–272.